# Lab #12 and #13
# Acoustic Communications and Networking

2.S998 Unmanned Marine Vehicle Autonomy, Sensing and Communications

**Mar 20, 2012 and Mar 22, 2012**

Guest lecturer: Toby Schneider
Michael Benjamin,
Henrik Schmidt
Department of Mechanical Engineering
MIT, Cambridge MA 02139

## Contents

# 1 Overview and Objectives

## 1.1 Structure of Today's Lab and Goals

Today's and Thursday's labs are combined into a single longer lab focused on acoustic communications and networking. By the end, you will have a good understanding of the difficulties AUV researchers face when attempting to communicate over the very low throughput / high latency acoustic link.

In this lab, we will make use of the Goby Underwater Autonomy Project to provide us with the backbone networking software. We will be both simulating the acoustic link using the uField Toolbox and using real acoustic modems connected via coaxial cables instead of electro-acoustic transducers (No, I'm not bringing a bathtub or the ocean into the classroom).

There are three parts to this lab:

1. Understand the process of creating a message suitable for sending on an acoustic modem using Goby and MOOS-IvP. (Section 2).

2. Deploy your lab partner's surface vehicle using an acoustic minipacket using the Micro-Modem hardware. (Section 3).

3. Design an basic arithmetic coder suitable for encoding temperature data generated by a simulated AUV's sensor. (Section 4).

**Important**: The second task can be done at any point before or after the other parts of the lab. I only have enough hardware for three lab groups to work at one time, so budget your time to do this task (1/2 hour or so) during lab time when one of the Micro-Modem boxes are free. Find a partner and sign up on the blackboard for your turn.

While the other sections are primarily individual assignments, you are welcome to troubleshoot and ask questions of your fellow students for all parts of this lab.

## 1.2 Preliminaries - Pre-lab assignment

In addition to the working `MOOS-IvP` setup required for every lab, we have a few more preliminaries for this lab. In order to maximize useful lab time this week, and hopefully *remove the need for any work to be done over Spring Break*, we ask you to complete the following setup and background pre-lab assignment and hand it on (on Stellar) *prior* to lab start on Tuesday March 20, 2012:

1. Download and compile Goby (version 2.0) per the instructions listed here: http://gobysoft.com/wiki/InstallFor2S998.

2. Read the first two sections (*Data Compression and Arithmetic Coding* and *Tutorial on Arithmetic Coding*) from the paper *Practical Implementations of Arithmetic Coding* by P. Howard and J. Vitter [1], available at http://www.gvu.gatech.edu/~jarek/courses/7491/Arithmetic2.pdf. Please answer the following questions:

   (a) State the primary advantage(s) of arithmetic source encoding.

   (b) State the primary disadvantage(s) of arithmetic coding.

   (c) Why might this type of coding make sense for marine vehicle communications?

3. We will be using Google Protocol Buffers ("protobuf") as the definition language for our acoustic datagrams (though we will *not* be using their encoding - it's not efficient enough!). Read through these pages to get a sense of what Google Protocol Buffers is and how it works with C++, and answer these quick questions:

   (a) Overview: http://code.google.com/apis/protocolbuffers/docs/overview.html

   (b) C++ Tutorial: http://code.google.com/apis/protocolbuffers/docs/cpptutorial.html

      i. What is the relationship between a Protobuf "Message" and a C++ "class"?

      ii. Given this .proto file, how would I (in C++) set the value of the "temperature" field to 21.435?

```
message CTDSample
{
  optional double temperature = 1;
  optional double salinity = 2;
}
```

### 1.3 Preliminaries - Start of lab

1. Update and compile any changes to Goby since you did the pre-lab checkout:

```
cd goby
bzr up
./build.sh
```

2. You will need to check out a branch of moos-ivp-extend that contains the code and missions for today's lab. The structure of the repository should be familiar from previous labs. Use

```
bzr co lp:~tes/junk/moos-ivp-extend-acomms
```

   Build the repository using the include script:

```
cd moos-ivp-extend-acomms
./build.sh
```

   Add `moos-ivp-extend-acomms/bin` to your shell path. Open a new terminal window and check that you have correctly set your path using

```
which ctd_tester
```

   which should give something like

```
/home/toby/moos-ivp-extend-acomms/bin/ctd_tester
```

3. If you're using a Mac (OS X Lion), you'll need to install the PL2303 driver as explained on this blog post http://xbsd.nl/2011/07/pl2303-serial-usb-on-osx-lion.html Use "sudo bash" or "sudo tcsh" to get a root shell before following their instructions.

### 1.4 What you will hand in

By the end of lab Thursday Mar 22, 2012, you will hand in answers to the questions of sections 2 and 3. The arithmetic coder (Section 4) and related questions will be due Thursday April 5, but you should be able to complete most or all of it in the lab time provided. Please produce a PDF document with the answer to your questions, and hand in your source code as usual by zip'ing or tar'ing the `moos-ivp-extend-acomms` tree, renamed to `moos-ivp-yourname`. All of these should be uploaded to Stellar.

### 1.5 Glossary

We'll be doing some bit math and such, and since you all have a varieties of backgrounds, we've made a quick glossary to try to address as many foreign terms / code snippets as possible:

- Binary (base 2): A number system where there are only two numerals: 0 and 1. It works just like base 10 (decimal) in other regards, just use 2 instead of 10. Binary numbers are generally easy to identify as they only contain 0 and 1. Where ambiguous, I will use a prefix letter 'd' to mean decimal, and 'b' for binary. '0x' is the prefix commonly used for hexadecimal (this works in C++). You may find a calculator that can do binary to decimal conversions useful for this lab. The default Ubuntu calculator in Programming Mode will do that for you. You can also use Python:

  ```
  $ python
  >>> bin(200)[2:]
  '11001000'
  >>> int('11001000', 2)
  200
  ```

- Bit shifting (`operator<<` and `operator>>`): Moves the bits to the left or right by the number of places on the right side of the operator. This effectively multiples or divides the number by 2 to the number of bits shifted. For example:

  ```
  int i = 5; // binary 101
  int j = i << 2; // binary 10100
  assert(j == 20); // true
  ```

- Codec (**co**der/**dec**oder), coder, and encoder/decoder are all used interchangeable in this lab to mean a program that takes a high-level uncompressed input, produces a set of bits suitable for transmission across a medium (encode), and can reproduce the input on the other end (decode). This process is somewhat synonymous with "data marshalling." In certain contexts this is referred to as "source coding" to disambiguate it from "channel coding," which is interested in making the best use of a particular communications channel (and provides error correction).

# 2 Using Goby (pAcommsHandler) with MOOS-IvP

## 2.1 Launch a one-way point-to-point network

We're going to start with a basic two community network that is a simple version of a common underwater scenario - a "sender" community (often a subsea node collecting data), and a "receiver" (typically a topside on a ship). In reality, communications are bi-directional as we want to command the vehicle from the topside as well, but typically the data flow is lopsided, weighted heavily towards the data-collecting node.

Find the mission named "acomms1":

```
cd moos-ivp-extend-acomms/missions/acomms1
```

Launch it:

```
./launch.sh
```

You will see two communities launch: the **sender** on the left side of the screen and the **receiver** on the right side. Both communities have a window for pAcommsHandler (the MOOS interface to the `goby_acomms` library) in `show_gui: true` mode that provides a large amount of diagnostic information. The other applications include:

- pAcommsExample: This application, running on the sender community, is continuously publishing a message containing an index, a random number (placeholder for real data), and the sender of the message (currently my MIT id, `tes`).
- uXMS: You are familiar with this application. On the sender community, it is showing the output of pAcommsExample (MOOS variable `ACOMMS_EXAMPLE_OUT`). On the receiver community, shows the equivalent received variable after sending through the (simulated) acoustic link.
- pCoroner: Similar to uProcessWatch, shows if any applications known to pAntler are not currently running.

Some things to note before we continue:

- Vehicles are identified by an unsigned integer (or "modem id") that is much like a MAC address on your computer's network card. String names like "gilda" are too costly in message space to use. We can use lookup tables, however, to tie modem ids to names. The modem id "0" is reserved to indicate "broadcast" in Goby. All vehicles will decode messages sent to "0".

## 2.2 pAcommsHandler terminal output

pAcommsHandler is the focus here, so watch the text in the terminal window to get a sense of what is going on. Maximize it as needed. From the top left, going to the bottom left, and over to the right column, the diagnostic displays indicate the modules of `goby_acomms` in order from application layer to link layer (see Fig. 1):

- **Ungrouped messages**: messages that don't have a clear home.

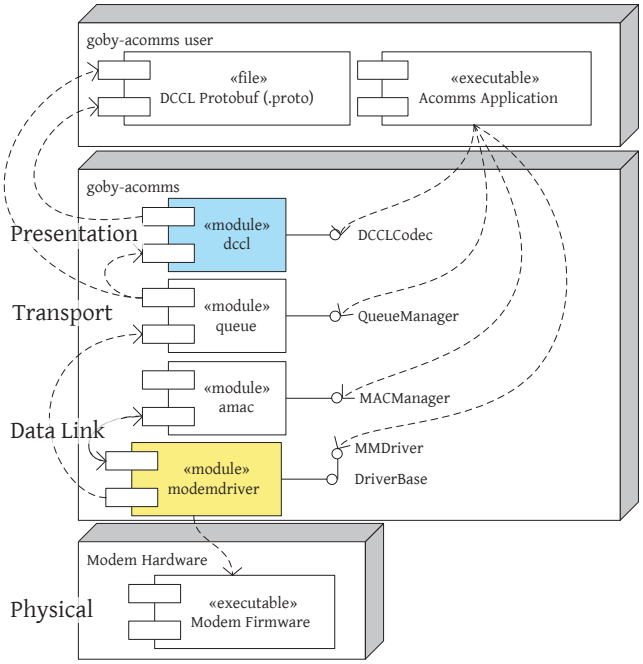


Figure 1: The software structure of Goby, showing the approximate OSI network layer on the left. In this lab, we will focus mostly on the DCCL and ModemDriver components.

- **pAcommsHandler**: messages specific to the interaction between Goby and MOOS. In this window you will basically see a copy of what pAcommsExample is publishing.
- **goby::acomms::dccl::encode**: shows the Dynamic Compact Control Language (DCCL) encoder output. When a message is encoded for sending, a line is printed here indicating that.
- **goby::acomms::dccl::decode**: shows the Dynamic Compact Control Language (DCCL) decoder output. When a message is decoded after receiving, a line is printed here indicating that.
- **goby::acomms::queue::push**: shows when a message is pushed (added) to the Goby dynamic priority queue.
- **goby::acomms::queue::pop**: shows when a message is popped (removed) from the Goby dynamic priority queue.
- **goby::acomms::priority**: shows the results of the priority contest that is held between all queues when it is time to send a message. Since there is only one message loaded (AcommsExampleMessage), it always wins.
- **goby::acomms::queue::out**: Messages showing the output of the queue data to the ModemDriver (link layer).

- **goby::acomms::queue::in**: Messages showing the input of the ModemDriver to the queuing layers. Messages are not queued upon receipt, but are rather immediately decoded.
- **goby::acomms::queue::amac**: Acoustic Medium Access Control (MAC) unit. This governs the time schedule when a given node is allowed to send. In this example, we have a single slot: vehicle 1 sends to the destination of the next message.
- **goby::acomms:modemdriver::out**: Raw outgoing messages to the modem hardware. In this case of this example, we are using a driver that passes messages through MOOS (using the uField `NODE_MESSAGE` variables). Switching to a real modem such as the WHOI Micro-Modem is only a matter of changing a few configuration variables.
- **goby::acomms:modemdriver::in**: Raw incoming messages from the modem hardware.

### 2.3 pAcommsHandler MOOS API

pAcommsHandler writes a variety of messages to MOOS variables that by default begin with `ACOMMS_`. The contents of all these messages are human-readable representations of Protobuf messages. Open uMS or uXMS and take a look at these variables (these are the default names, all of which can be configured in the MOOS file if desired). Some will only be published on one community or the other:

- `ACOMMS_RAW_INCOMING`: Shows the serial or TCP feed **to** the current acoustic modem. The uField Driver uses a hexadecimal representation of the Protobuf encoding of goby::acomms::protobuf::ModemTransmission. The WHOI Micro-Modem uses an NMEA-0183 interface as you'll see in the next section.
- `ACOMMS_RAW_OUTGOING`: Shows the serial or TCP feed **from** the current acoustic modem.
- `ACOMMS_MODEM_RECEIVE`: Abstracted message from the Modem driver. This message will look the same regardless of which low-level driver is being used. The `frame` field contains a DCCL encoded message.
- `ACOMMS_MODEM_TRANSMIT`: Abstracted message to the Modem driver. This message will look the same regardless of which low-level driver is being used. The `frame` field contains a DCCL encoded message.
- `ACOMMS_QUEUE_RECEIVE`: Decoded received message. This should look very much like the final published message.
- `ACOMMS_QUEUE_TRANSMIT`: Unencoded sent message.
- `ACOMMS_QSIZE`: Indicates whenever a queue changes size (due to a message being pushed or popped).
- `ACOMMS_MAC_INITIATE_TRANSMISSION`: Indicates the beginning of a transmission.

The uField Driver also uses a few more MOOS variables to do its work:

- `NODE_MESSAGE_LOCAL`: posted to allow the uField tools to route the message via the shoreside to another vehicle. This gives us some rudimentary network simulation capabilities (primarily range-dependent packet dropping).
- `ACOMMS_UFIELD_DRIVER_OUT`: copy of the `string_val` field of the message posted `NODE_MESSAGE_LOCAL` in the same encoded format as will be published to `ACOMMS_UFIELD_DRIVER_IN` on the receiving end.
- `ACOMMS_UFIELD_DRIVER_IN`: posted by uFieldMessageHandler after routing the message originally posted to `NODE_MESSAGE_LOCAL`. This is subscribed to by the Goby uField Driver.

## 2.4 Comparing encoding representations

Look at the terminal output of `pAcommsExample`. After each message, the size is written if the message were encoded using a std::string ("TextFormat"), the built-in Protobuf encoder, and DCCL. It should be clear from this that DCCL offers the smallest packet size. The drawback is the extra work precisely defining the origin of our messages' data. We will look at this briefly in the next subsection (2.5), and substantially more using a probabilistic framework in Section 4.

We're done with running the `acomms1` example, but before killing MOOS (`mykill`), answer the first question below in Section 2.7.

## 2.5 DCCL Message: AcommsExampleMessage

The message AcommsExampleMessage is a Google Protocol Buffers message that is defined in

`moos-ivp-extend-acomms/src/protobuf/acomms_example.proto`

To allow DCCL to encode this message, some custom message (`goby.msg`) and field extensions (`goby.field`) are defined to allow us to attach further meta-data to the message definition. This additional information is used to produce an even smaller encoded message than the default Protobuf technique. Only a subset of the extensions are used here. The full set is given in http://gobysoft.org/doc/2.0/index.html., and the requirements of the various built-in encoders/decoders ("codecs") is given on this page:
http://gobysoft.org/doc/2.0/acomms_dccl.html#dccl_options

Normally, integers are stored in 32 or 64 bits, but most of the time we're actually sending messages from a small subset of the available $2^{32}$ or $2^{64}$ integers. Thus, the built-in DCCL codecs use arbitrarily bounded integers to send data[1]. For example, if you set

```
(goby.field).dccl.max = 99, (goby.field).dccl.min = 0
```

the default DCCL codec for `int32` creates a 7-bit integer[2] to store it, a savings of 25 bits or 78% over using a 32-bit integer. This is how the field named `random` in `AcommsExampleMessage` is being encoded.

[1] While slightly out of date, this paper gives the algorithms for the built-in codecs: http://gobysoft.com/dl/dccl_oceans10.pdf

[2] $\text{ceil}(\log 2(99 - 0 + 1)) = 7$

## 2.6 Examine a special purpose encoder: VarIntCodec

While this simple technique for reducing the size of our messages is a useful first attempt to send tightly compressed data, we can often do better than this. The key to this improvement is to understand the underlying *probability distribution* of the data (or "symbols") to be sent. Let's assume that the small numbers (say 0-127) are sent much more frequently than large numbers, but we still want to be able to handle larger numbers as the need arises. One way to do this is with a variable integer ("varint") encoder[3], which uses one byte for the values 0-127 and two bytes for 128-32768. The final bit is used for indicating the size of the range (large or small). For example, the small number 50 would be encoded as:

```
0110010 0
   ^    ^
   |    |
  d50   0 means short form (1 byte = 8 bits)
```

On the other hand, the large number 500 would be

```
000000111110100 1
       ^        ^
       |        |
     d500       1 means long form (2 bytes = 16 bits)
```

Such a codec can be written outside the Goby project and imported as a plugin. To do so, we define some name for our codec and write it into the Protobuf message:

```
(goby.field).dccl.codec="lab12_varint"
```

To write our codec, we subclass DCCLTypedFieldCodec[4] for singular fields (`optional` or `required` in Protobuf parlance) or DCCLRepeatedTypedFieldCodec[5] for `repeated` fields (which are basically vectors). This is done for the varint codec in

```
moos-ivp-extend-acomms/src/codecs/varint.h
moos-ivp-extend-acomms/src/codecs/varint.cpp
```

You will want to look at this code to get an idea of how a simple variable-length codec looks in advance of writing a slightly more complicated one in section 4.

Finally, we must load our codecs using the code given in

```
moos-ivp-extend-acomms/src/codecs/codec_load.cpp
```

All this code is compiled into a dynamic library (`lab12codecs.so/dylib`) that is loaded by pAcommsHandler[6], which in turn loads the codecs into the Goby DCCL module. This extensibility allows for tailoring acoustic messaging to specific mission needs, *without needing to modify the Goby source code*. However, this level of effort isn't required, as the built-in codecs provide useful functionality for deployments with slightly less rigorous data compression requirements.

[3] as an interesting side note, variable integer encoding is how Google Protocol buffers default encoding works: see http://code.google.com/apis/protocolbuffers/docs/encoding.html

[4] http://gobysoft.org/doc/2.0/classgoby_1_1acomms_1_1DCCLTypedFieldCodec.html

[5] http://gobysoft.com/doc/2.0/classgoby_1_1acomms_1_1DCCLRepeatedTypedFieldCodec.html

[6] using dlopen

## 2.7 Questions

Some quick questions for you to consider and hand in by the end of the lab on Thursday.

1. When you examine the output of pAcommsExample, you will notice that more messages are being sent then are able to make it through the simulated acoustic link (which has the bitrate of 32 bytes/10 seconds = 25.6 bps). The reality here is that *total throughput is usually unattainable on acoustic links*. This is a foreign concept to us on terrestrial links (e.g. my emails will be sent faster than I create them). List two ways you could deal with this buffer overflow problem.

2. If we were using the varint codec given in section 2.6 and our numbers were almost always $large$, we would use 16 bits to encode symbols that only require 15 bits to encode (because we use 1 bit to indicate the size domain). Thus, there is some probability, $p_0$, where $P(small) >= p_0$ that makes use of the varint codec worthwhile (i.e. it will use fewer bits *on average*). What is that value $p_0$? As before, $large$ is defined as values [128, 32768) and $small$ is [0, 128).

# 3 Using an acoustic modem to deploy an underwater vehicle

This section is done on limited modem hardware (up to three teams of two at one time). Sign up on the blackboard and we'll go in that order.

## 3.1 The WHOI Micro-Modem

The WHOI Micro-Modem is one of only a few commercially available acoustic modems. It uses two types of modulation (Frequency-Hopping Frequency Shift Keying (FH-FSK) and Phase Shift Keying (PSK)) to achieve a range of packet sizes from 32 bytes / packet to 2048 bytes / packet. Higher rates are more susceptible to bit errors and thus packet loss. Realistically, a single packet can be sent about once every 10-15 seconds after accounting for signal length (3.5 seconds), time of flight (100 to 10000 meters range over 1500 m/s nominal speed of sound), and potentially an acoustic acknowledgment. Thus, realistic data rates range from 20 bps to to 200 bps *before* accounting for packet loss (which can be easily greater than fifty percent). This is why we spend so much time in the rest of this lab focusing on data compression for very small packets.

### 3.1.1 Serial Data Protocol

The WHOI Micro-Modem connects over a serial line (RS-232) at a default 19200 baud rate. It uses a data protocol roughly based on the NMEA 0183 standard, which is widely used in marine electronics (e.g. GPS units)[7]. Two primary "talker" ids are employed: $CC for messages from the computer and $CA for messages from the modem. A full PSK transaction including acknowledgment from modem 1 to modem 2 looks like this and is rather complicated:

```
{goby::acomms::modemdriver::out::1}: $CCCYC,0,1,2,2,1,3*5A
{goby::acomms::modemdriver::out::1}: ^ Network Cycle Initialization Command
{goby::acomms::modemdriver::in::1}: $CACYC,0,1,2,2,1,3*58
{goby::acomms::modemdriver::in::1}: ^ Echo of Network Cycle Initialization command
{goby::acomms::modemdriver::in::1}: $CADRQ,025728,1,2,1,64,1*4E
{goby::acomms::modemdriver::in::1}: ^ Data request message, modem to host
{goby::acomms::modemdriver::out::1}: $CCTXD,1,2,1,3131313131313131313131313131313131...
{goby::acomms::modemdriver::out::1}: ^ Transmit binary data message, host to modem
{goby::acomms::modemdriver::in::1}: $CATXD,1,2,1,64*7A
{goby::acomms::modemdriver::in::1}: ^ Echo back of transmit binary data message
{goby::acomms::modemdriver::in::1}: $CADRQ,025729,1,2,1,64,2*4C
{goby::acomms::modemdriver::in::1}: ^ Data request message, modem to host
{goby::acomms::modemdriver::out::1}: $CCTXD,1,2,1,3232323232323232323232323232323232...
{goby::acomms::modemdriver::out::1}: ^ Transmit binary data message, host to modem
{goby::acomms::modemdriver::in::1}: $CATXD,1,2,1,64*7A
{goby::acomms::modemdriver::in::1}: ^ Echo back of transmit binary data message
{goby::acomms::modemdriver::in::1}: $CADRQ,025729,1,2,1,64,3*4D
{goby::acomms::modemdriver::in::1}: ^ Data request message, modem to host
{goby::acomms::modemdriver::out::1}: $CCTXD,1,2,1,*7A
{goby::acomms::modemdriver::out::1}: ^ Transmit binary data message, host to modem
{goby::acomms::modemdriver::in::1}: $CATXD,1,2,1,0*48
{goby::acomms::modemdriver::in::1}: ^ Echo back of transmit binary data message
{goby::acomms::modemdriver::in::1}: $CATXP,128*49
```

[7]The entire protocol is posted at http://acomms.whoi.edu/umodem/documentation.html under "Micro-Modem Software Interface Guide"

```
{goby::acomms::modemdriver::in::1}: ^ Start of packet transmission, modem to host
{goby::acomms::modemdriver::in::2}: $CARXP,1*45
{goby::acomms::modemdriver::in::2}: ^ Incoming packet detected, modem to host
{goby::acomms::modemdriver::in::1}: $CATXF,128*5F
{goby::acomms::modemdriver::in::1}: ^ End of packet transmission, modem to host
{goby::acomms::modemdriver::in::1}: $CAREV,025733,AUV,0.94.0.00*0E
{goby::acomms::modemdriver::in::1}: ^ Software revision message, modem to host
{goby::acomms::modemdriver::in::1}: $CAREV,025733,COPROC,0.10.0.46*40
{goby::acomms::modemdriver::in::1}: ^ Software revision message, modem to host
{goby::acomms::modemdriver::in::2}: $CACYC,1,1,2,2,1,3*59
{goby::acomms::modemdriver::in::2}: ^ Echo of Network Cycle Initialization command
{goby::acomms::modemdriver::in::2}: $CARXD,1,2,1,1,3131313131313131313131313131313...
{goby::acomms::modemdriver::in::2}: ^ Received binary message, modem to host
{goby::acomms::modemdriver::in::2}: $CARXD,1,2,1,2,3232323232323232323232323232323...
{goby::acomms::modemdriver::in::2}: ^ Received binary message, modem to host
{goby::acomms::modemdriver::in::2}: $CARXD,1,2,1,3,*61
{goby::acomms::modemdriver::in::2}: ^ Received binary message, modem to host
{goby::acomms::modemdriver::in::2}: $CACST,0,025729.0000,1,1903,20,0410,0167,00,01,...
{goby::acomms::modemdriver::in::2}: ^ Communication cycle receive statistics
{goby::acomms::modemdriver::in::2}: $CAREV,025733,AUV,0.94.0.00*0E
{goby::acomms::modemdriver::in::2}: ^ Software revision message, modem to host
{goby::acomms::modemdriver::in::2}: $CAREV,025733,COPROC,0.10.0.46*40
{goby::acomms::modemdriver::in::2}: ^ Software revision message, modem to host
{goby::acomms::modemdriver::in::1}: $CARXP,1*45
{goby::acomms::modemdriver::in::1}: ^ Incoming packet detected, modem to host
{goby::acomms::modemdriver::in::2}: $CATXP,0*42
{goby::acomms::modemdriver::in::2}: ^ Start of packet transmission, modem to host
{goby::acomms::modemdriver::in::1}: $CAACK,2,1,1,1*48
{goby::acomms::modemdriver::in::1}: ^ Acknowledgment of a transmitted packet
{goby::acomms::modemdriver::in::1}: $CAACK,2,1,2,1*4B
{goby::acomms::modemdriver::in::1}: ^ Acknowledgment of a transmitted packet
{goby::acomms::modemdriver::in::1}: $CAACK,2,1,3,1*4A
{goby::acomms::modemdriver::in::1}: ^ Acknowledgment of a transmitted packet
{goby::acomms::modemdriver::in::1}: $CACST,0,025734.0000,1,1608,20,0295,0165,00,01,01,...
```

The Micro-Modem also has several other features, one of which is the handy ability to send 13 bits in the mini-packet (normally used for control messages such as the acknowledgment). This is pretty much the acoustic modem equivalent of the SMS. While very small in size, the packet is received with high reliability, and the transaction is simpler (see also Fig. 2) (sending hexadecimal data `0x1234`):

```
{goby::acomms::modemdriver::out::1}: $CCMUC,1,2,1234
{goby::acomms::modemdriver::in::1}: $CAMUC,1,2,1234*72
{goby::acomms::modemdriver::in::1}: $CATXP,0*42
{goby::acomms::modemdriver::in::1}: $CATXF,0*54
{goby::acomms::modemdriver::in::2}: $CARXP,1*45
{goby::acomms::modemdriver::in::2}: $CAMUA,1,2,1234*70
```

You should try this now. Connect one modem to each of you and your lab partner's machines. Both of you should identify the serial port created by the connection. Use `ls /dev/tty*` to see what device is added when you plug in the USB-serial dongle. On Linux it's probably `/dev/ttyUSB0`; on Mac OS X it's probably `/dev/tty.usbserial` or `/dev/tty.PL2303...`). Replace all usages of `/dev/ttyUSB0` below with the correct device name for your machine.

Open picocom

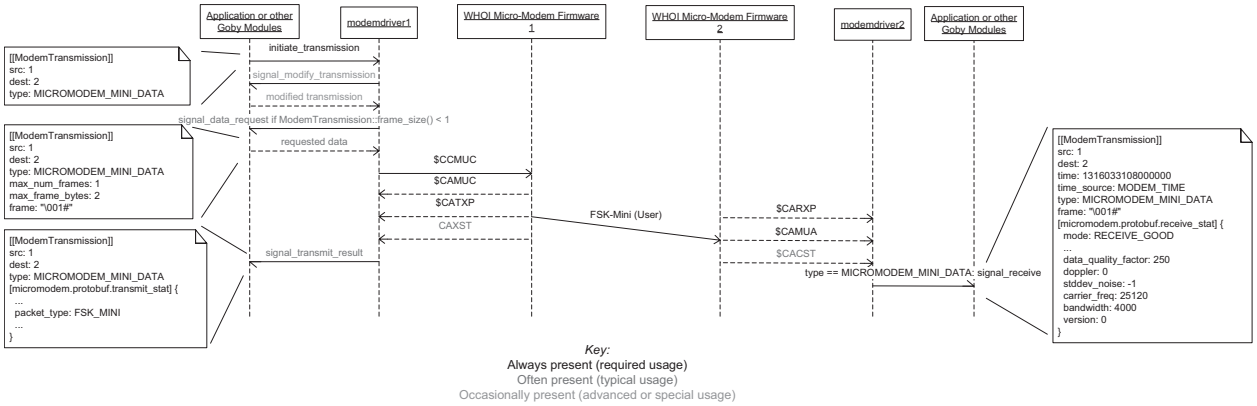


Figure 2: WHOI Micro-Modem User mini-packet 13 bit data transmission using Goby. You should see something like the far right ModemTransmission message if you open uXMS on `ACOMMS_MODEM_RECEIVE` on the receiving end (vehicle). The actual acoustic transmission occurs where it says "FSK-Mini" in the center.

```
picocom /dev/ttyUSB0 -b 19200
```

and examine the output

```
$CAREV,025733,AUV,0.94.0.00*0E
$CAREV,025733,COPROC,0.10.0.46*40
```

`$CAREV` is the software revision number and serves as a heartbeat. Configure one modem to be modem id 1 (the modem id is called SRC on the WHOI Micro-Modem)

```
printf '$CCCFG,SRC,1\r\n' > /dev/ttyUSB0
```

and the other 2

```
printf '$CCCFG,SRC,2\r\n' > /dev/ttyUSB0
```

Try sending each other minipackets. For the person connected to modem 1:

```
printf '$CCMUC,1,2,1234\r\n' > /dev/ttyUSB0
```

or for the person on modem 2:

```
printf '$CCMUC,2,1,1234\r\n' > /dev/ttyUSB0
```

On the receiving end you will see the received data

```
$CAMUC,1,2,1234*72
```

Feel free to replace `1234` with any 13-bit hex number (`0000` to `1fff`). Close picocom (CTRL-A CTRL-X) on both machines in preparation for the next section.

### 3.2 Sending a Mini-Packet command using pAcommsHandler

One of you will act the role of the shoreside, the other the vehicle. It doesn't matter which person was id 1 last time, pAcommsHandler will change it to the correct value.

Find the mission named "acomms2":

```
cd moos-ivp-extend-acomms/missions/acomms2
```

If you are using a Linux machine, change the line in `goby_liaison.pb.cfg`

```
load_shared_library: "../../lib/liblab12messages.dylib"
```

to

```
load_shared_library: "../../lib/liblab12messages.so"
```

Change the `MICROMODEM_PORT` definition in `meta_shoreside.moos` to correspond to the correct serial port and launch the shoreside on one machine:

```
./launch_shoreside.sh
```

Similarly, change the `MICROMODEM_PORT` definition in `meta_vehicle.moos` and launch the vehicle on the other machine:

```
./launch_vehicle.sh
```

Open a web browser of your choice and point it to http://localhost:50001/#/commander. Choose the `MiniCommand` message and fill in the fields (e.g. destination: 2, state: DEPLOY, manual_override: false, speed: 3). Click "Send" and the vehicle will receive the message and publish the variables specified under the `translator_entry { publish {} }` in `plug_pAcommsHandler.moos`. Once you have this working properly, kill the vehicle community and open picocom again

```
picocom /dev/ttyUSB0 -b 19200
```

Examine the raw modem feed when you send the MiniCommand message. Note the `$CAMUA` when the shoreside pushes "Send" on Goby Liaison Commander. Now, restart the vehicle. Examine the MOOS variables `ACOMMS_RAW_INCOMING` and `ACOMMS_MODEM_RECEIVE`. In addition to the `src`, `dest`, `type`, `frame` fields you saw when you looked at `ACOMMS_MODEM_RECEIVE` in section 2.3, you'll also notice a large amount of diagnostic information about the transmission. This is extracted from the `$CACST` message from the WHOI Micro-Modem.

The interaction with the Goby ModemDriver and the rest of Goby (and MOOS via pAcommsHandler) is diagrammed in Fig. 2.

### 3.3 Questions

Please hand these in with the questions from the other sections by the end of lab Thursday.

1. What is the raw mini-packet NMEA-0183 message received by the "vehicle" computer when you send "destination: 2 state: DEPLOY manual_override: false"?
2. Besides sending small commands from a shoreside computer as we did in this section, what is another possible use for a 13-bit packet?

Extra credit Once you are able to command the vehicle successfully, disconnect both the coaxial cables from either the top or bottom modem. Now try to send a RETURN command to the vehicle. What happens? Why is this happening?

# 4 Encoding a sample from a Conductivity-Temperature-Depth (CTD) sensor

In section 2.6, we explored a variable integer encoder for encoding single integer (int32) fields. In this section, your goal will be to write an entropy encoder (specifically, an arithmetic encoder) for encoding temperature data from a Conductivity-Temperature-Depth (CTD) sensor (pictured in Fig. 3), which measures salinity (from conductivity), temperature, and depth (from pressure). CTD data are useful for vehicles performing collaborative environmental missions. They can also used to calculate the speed of sound using various empirical formulas [2], so they are also invaluable for acoustic modeling in sonar-based applications.


Figure 3: A typical vehicle CTD: the Sea-Bird SBE49 [3]

A traditional ship-based use of a CTD involves casting the instrument to a certain depth at a fixed location in the ocean (a "station"). When the cast is complete, the ship can move to another station, and another CTD cast is made. In this way, one can gather (and interpolate) a three-dimensional picture of the salinity and temperature of the ocean. Since typically AUVs do not move directly vertically, a CTD profile is often made by "yoyoing" the vehicle; that is, the vehicle performs a series of periodic oscillations in depth while still covering ground.

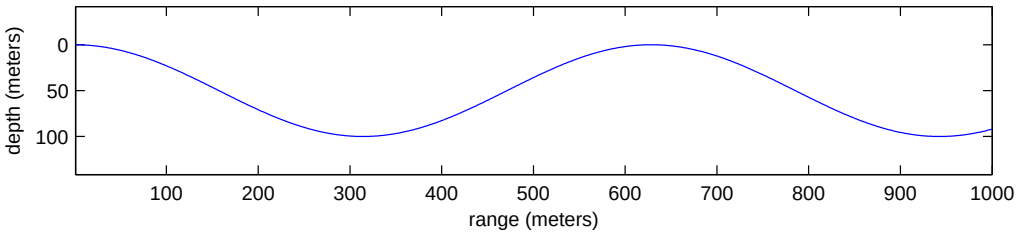


Figure 4: The path of a vehicle performing a sinusoidal yoyo for the purpose of collecting CTD data.

In this section, we will be assuming the vehicle is continuously sampling the ocean between 0 and 100 meters as illustrated in Fig. 4. *A priori* (perhaps from an earlier data set or simulation),

we expect the temperature profile (in depth) to be that given in Fig. 5, which is defined by

$$t(d) = \begin{cases} 20 & 0 <= d < 10 \\ 10 + \frac{30-d}{2} & 10 <= d < 30 \\ 10 & d >= 30 \end{cases} \quad (1)$$

where $t$ is the current temperature in $^{\circ}C$, and $d$ is depth in $m$. Given this information, it is possible for us to determine a probability mass function of the measured temperature. This will provide us with the ability to predict the samples we will obtain and thus improve the performance of the encoder used to prepare them for transmission.

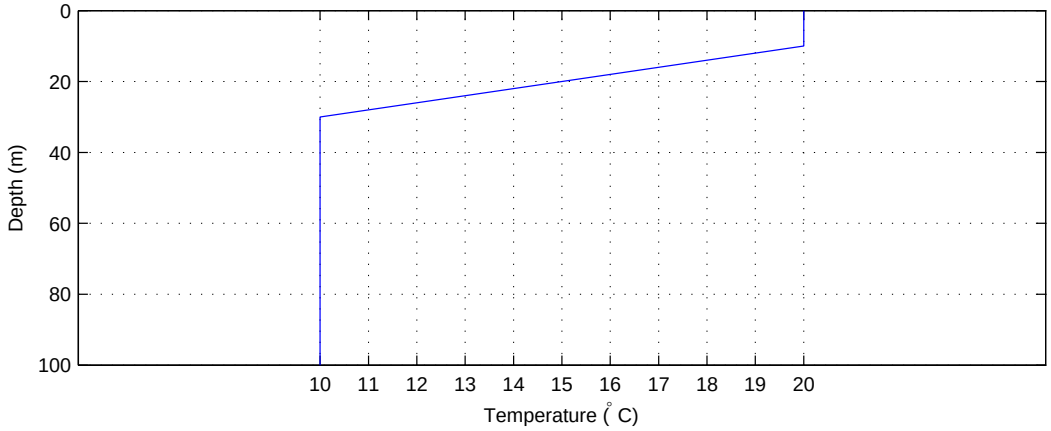


Figure 5: Assumed temperature profile (based off a typical Mediterranean summer profile)

### 4.1 Your Task

Your goal in this section is to finish an arithmetic coder that will encode nearly optimally [1] the CTD samples coming from this hypothetical AUV sampling the profile given in Fig. 5. Specifically, we will make some simplifying assumptions to make your task manageable in the time allotted:

- We will use arithmetic encoding only for the temperature; we will send depth and salinity values using the default DCCL codecs (bounded real numbers).
- We will send the temperature values (in $^{\circ}C$) rounded to the nearest integer. This means we will be transmitting from a set of eleven symbols: {10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}.
- We will send exactly five samples per transmission (which means encoding five samples at a time). This means you will not run into trouble using the double precision floating point type (`double`) to hold the entire encoded answer range. For this assignment you do not need to worry about a variable number of samples per transmission (though in general you would want to consider this using an end-of-file or EOF symbol). This all means you can use the basic approach diagrammed in the table at the top of page 4 in Howard and Vitter's paper, and not worry about using integer arithmetic or interval expansion.

Also, to avoid spending all the time you have working out the necessary software "plumbing", we've provided you with a partially implemented codec:

```
moos-ivp-extend-acomms/src/codecs/arithmetic.cpp
moos-ivp-extend-acomms/src/codecs/arithmetic.h
```

The DCCL message `CTDMessage` you will be sending is defined in

```
moos-ivp-extend-acomms/src/protobuf/ctd.proto
```

A very similar message `CTDMessage` using the DCCL default codecs is defined in

```
moos-ivp-extend-acomms/src/protobuf/ctd_default.proto
```

and will form the basis of comparison with the arithmetic coder.

Finally, we have provided a unit test `ctd_tester` in

```
moos-ivp-extend-acomms/src/ctd_tester/main.cpp
```

that runs a number of encode/decode tests on your arithmetic coder. It checks two deterministic cases: the "best case" (temperature = {10, 10, 10, 10, 10}), and a "worst case" (temperature = {19, 19, 18, 17, 17}). Then it checks ten random sets of temperatures sampled by uniformly placing the vehicle in [0, 100] meters in depth, then reading the value off Fig. 5. Obviously a real vehicle couldn't do this, but it serves to check our coder's functionality and performance. To run this unit test:

```
cd moos-ivp-extend-acomms
./build.sh
ctd_tester
```

Until you finish the ArithmeticCoder class, `ctd_tester` will assert-fail[8] with a message like:

```
ctd_tester: /home/toby/moos-ivp-extend-acomms/src/ctd_tester/main.cpp:97:
void run_test(const std::vector<double>&): Assertion
'msg_in1.SerializeAsString() == msg_out1.SerializeAsString()' failed.
```

## 4.2 Approach

This section outlines a step-by-step approach to this problem:

1. Ensure you understand the basics of arithmetic encoding by reading the first two sections of *Practical Implementations of Arithmetic Coding* by P. Howard and J. Vitter [1]. This was the pre-lab assignment, but it may be helpful to have the paper handy.

2. Given that the vehicle is assumed be uniformly distributed between 0 and 100 meters, as given by

$$p_D(d) = \begin{cases} 0.01 & 0 <= d < 100 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

   where $D$ is the random variable indicating the vehicle's depth (in meters) below the sea surface. Combining this with the (assumed) deterministic model for temperature over depth

[8] Assertions are debugging / testing tool that cause the program to exit with an error message if the contents of the `assert` macro is false.

Table 1: Probability mass function of sampled temperature.

| Symbol $t$ (temperature) | $p_T(t)$ |
|---|---|
| 10 | |
| 11 | |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| 16 | |
| 17 | |
| 18 | |
| 19 | |
| 20 | |

given in Fig. 5 and Eq. 1 you should fill in Table 1 for the probability mass function for the temperature symbols (random variable $T$). Make sure your probabilities sum to one:

$$\sum_{t=10}^{20} p_T(t) = 1 \tag{3}$$

You may wish to check your values with me before implementing any code.

3. Implement the remaining virtual methods in `arithmetic.cpp` to finish defining our custom codec `ArithmeticCodec`:

```
Bitset encode_repeated(const std::vector<goby::int32>& wire_values);
std::vector<goby::int32> decode_repeated(const Bitset& bits);
unsigned max_size_repeated();
unsigned min_size_repeated();
```

These functions should be relatively well explained in `arithmetic.h`. If you are interested, the Doxygen documentation for DCCLRepeatedTypedFieldCodec, giving all the member functions and some brief explanation, is available at http://gobysoft.com/doc/2.0/classgoby_1_1acomms_1_1DCCLRepeatedTypedFieldCodec.html. A few notes to avoid unnecessary difficulty or confusion:

- The type `goby::int32` is used to make all the behind-the-scenes templating work correctly (it's a `typedef` for `google::protobuf::int32` which corresponds to the `int32` type in the protobuf file). You can use it like a normal `int`, which it probably is on your system anyway.
- All sizes are in bits (e.g. return value in `unsigned max_size_repeated();`). There are eight bits to a byte.

- `unsigned max_size_repeated();` must return an upper bound $U$ (in bits) for the worst case number of bits the encoder will use. In this implementation of the arithmetic encoder,

$$U = \left\lceil \frac{1}{log_2(Np_{min})} \right\rceil + 1 \tag{4}$$

  where $N$ is the number of symbols (five in this case), $p_{min}$ is the minimum probability of any of the symbols. You may notice that this is almost the number of bits given by the entropy (in an information content sense) of this probability distribution. The lower bound case, `unsigned min_size_repeated();`, is very similar, except of course it is governed by the most likely outcome.

- The type `goby::acomms::Bitset` is a `typedef`[9] for `boost::dynamic_bitset<unsigned char>`. The documentation is listed here: http://www.boost.org/doc/libs/1_49_0/libs/dynamic_bitset/dynamic_bitset.html. To avoid some of the nuisance of learning `dynamic_bitset`, we've provided two helper functions: `range_to_bits` for encoding the decimal range you will provide and `bits_to_range` for decoding.

- `range_to_bits`, which is provided for you, converts a double precision `std::pair` decimal range that is a subset of [0, 1) such as [0.25,0.265625) into the shortest binary fraction that uniquely identifies the range, e.g. 000010. The `Bitset` output (e.g. 000010) has the least significant bit as the largest fraction ($2^{-1}$) and the most significant bit as the smallest fraction ($2^{-6}$). In regular mathematical notation, this would be written as 0.01000, and means in decimal:

```
000010
|||||->0*2^-1 +
||||->1*2^-2 +
|||->0*2^-3 +
||->0*2^-4 +
|->0*2^-5 +
->0*2^-6
---------------
0.25
```

- `bits_to_range` takes a `Bitset` generated by `range_to_bits` and returns the range uniquely identified by that `Bitset`. Since we are forced to work in whole bits, the returned range from `bits_to_range` is generally a proper subset (a smaller range) of the original range passed to `range_to_bits`. For example, if we write the code

```
Bitset bits = range_to_bits(std::make_pair(lower, upper));
std::pair<double, double> result = bits_to_range(bits);

std::cout << i << ": " << std::setprecision(15)
          << "in: [" << lower
          << ", " << upper
          << "), out: ["
```

[9] for those new to C++, a `typedef` is essentially an alias for a type. `typedef LongNameType L` allows me to use `L` in place of `LongNameType`

```
        << result.first << "," << result.second << ")"
        << ", bits: " << bits << std::endl;
```

and run it ten times with random inputs to `double lower` and `double upper`, the result looks like:

```
1: in: [0.224450518947304, 0.272736707829282), out: [0.25,0.265625), bits: 000010
2: in: [0.343518942754492, 0.498916563810276), out: [0.375,0.4375), bits: 0110
3: in: [0.181075460361818, 0.735460476826625), out: [0.25,0.5), bits: 10
4: in: [0.169450353444298, 0.263964024960978), out: [0.1875,0.25), bits: 1100
5: in: [0.478094624578066, 0.73305528551948), out: [0.5,0.625), bits: 001
6: in: [0.796269196922085, 0.977315562766658), out: [0.875,0.9375), bits: 0111
7: in: [0.139742564009383, 0.416557414651177), out: [0.25,0.375), bits: 010
8: in: [0.219583303769856, 0.34400233828649), out: [0.25,0.3125), bits: 0010
9: in: [0.415195364698393, 0.418356456988657), out: [0.416015625,0.41796875),
 bits: 101010110
10: in: [0.350675355340669, 0.812326993705857), out: [0.5,0.75), bits: 01
```

Not surprisingly, tighter ranges take more bits to represent. More probable results lead a larger range to encode. Thus, more probable events have smaller encoded sizes, leading to the desired result.

- When your method `std::vector<goby::int32> decode_repeated(const Bitset& bits);` is called, `bits` *only* contains the $N_{min}$ number of least significant bits required to decode, where $N_{min}$ is the value returned by `unsigned min_size_repeated();`. To get more bits from your parent class (which you will need in all but the best case situation), call the function `get_more_bits(unsigned n)`, where $n$ is the number of bits requested. The requested bits are placed at the most significant end of `bits`. **You must request exactly the number of bits you provided at encode_repeated** or you will break the encoding and probably throw a `std::bad_alloc` exception as DCCL tries to provide more bits than are available. The shell version of `ArithmeticCodec` provided to you gives an example of how to request the remaining bits and where they are placed. In your final design, you will probably be requesting one bit at a time until you have reached the necessary precision to fully determine the five symbols you encoded.

4. Test your codec using the `ctd_tester` unit test previously described. The result should look something like:

```
= Begin CTDMessage =
Actual maximum size of message: ?? bytes / ??? bits [dccl.id head: 8, user head: 0, ...
Allowed maximum size of message: 32 bytes / 256 bits
== Begin Header ==
== End Header ==
== Begin Body ==
CTDMessage
 repeated int32 depth = 1;
 :: size = 50 bit(s)
 repeated int32 temperature = 2;
 :: min size = ?? bit(s)
 :: max size = ?? bit(s)
 repeated double salinity = 3;
 :: size = 55 bit(s)
:: min size = ??? bit(s)
```

```
:: max size = ??? bit(s)
== End Body ==
= End CTDMessage =
****************** BEGIN TEST 0 ******************
In: depth: [12, 13, 14, 13, 13] temperature: [19, 19, 18, 17, 17] salinity: [30.09, 30.1,...
Try encode ...
Encoded (hex): 660d38f080030e1c94154bfffc0740011210
Size (bytes): ??
Default codec size (bytes): 17
Try decode...
Out: depth: [12, 13, 14, 13, 13] temperature: [19, 19, 18, 17, 17] salinity: [30.09, 30.1, ...
****************** PASSED TEST 0 ******************

****************** BEGIN TEST 1 ******************
In: depth: [12, 13, 14, 13, 13] temperature: [10, 10, 10, 10, 10] salinity: [30.09, 30.1, ...
Try encode ...
Encoded (hex): 660d38f080030ec03fff0150800404
Size (bytes): ??
Default codec size (bytes): 17
Try decode...
Out: depth: [12, 13, 14, 13, 13] temperature: [10, 10, 10, 10, 10] salinity: [30.09, 30.1, ...
****************** PASSED TEST 1 ******************

(... and similar through TEST 11 ...)

all tests passed
```

(we've added `??` to some places which will be actual numbers in your output).

### 4.3 Questions

Some quick questions for you to consider and hand in with your completed ArithmeticCodec by Thursday April 5.

1. If you run `analyze_dccl` on `ctd_default.proto`, you find

   ```
   repeated int32 temperature = 2;
   :: size = 20 bit(s)
   ```

   How does the size of the encoded message from your arithmetic coder compare to that of the default DCCL encoder? (hint: if the return value from `encode_repeated` is called `data_bits`, use

   ```
   std::cout << data_bits.size() << std::endl;
   ```

   to show how many bits your arithmetic encoding takes) Does it always outperform or underperform the default encoder? If not, in which cases does it underperform and in which cases does it overperform?

2. How could you improve the performance of the arithmetic coder even more in this situation (hint: think about the sensor's sampling process and the AUV's realistic motion through the water)?

## References


[1] P. Howard and J. Vitter, "Practical implementations of arithmetic coding," *Image and Text compression*, pp. 85–112, 1992. [Online]. Available: http://www.gvu.gatech.edu/~jarek/courses/7491/Arithmetic2.pdf

[2] "Technical guides - speed of sound in sea-water." [Online]. Available: http://resource.npl.co.uk/acoustics/techguides/soundseawater/

[3] "FastCAT CTD Sensor SBE 49." [Online]. Available: http://www.seabird.com/products/spec_sheets/49data.htm

# 2.S998 Marine Autonomy, Sensing and Communications

Spring 2012