A Compact Control Language for Autonomous Underwater Vehicles

February 16, 2004

Roger P. Stokey
Woods Hole Oceanographic Institution
Woods Hole, MA 02543
rstokey@whoi.edu
(508) 289-3323

This document describes a communications protocol designed to allow multiple autonomous underwater vehicles (AUVs) to communicate with a central control point in an efficient and robust manner. The protocol also allows multiple AUVs to communicate with each other during cooperative missions. The protocol was developed specifically for low-bandwidth acoustic communication links and thus is extremely compact. All messages are designed to fit into small packets by using the appropriate minimum number of bits to denote each value. It is the basis for acoustic communications by the REMUS underwater vehicle for communication with both shore based personnel as well as other vehicles. Although designed for the capabilities of REMUS 100 and its derivatives (such as the REMUS 6000/SAMS system), it is sufficiently generic to have application for other autonomous underwater vehicles.

The protocol does not describe the communication transport layer, however it is designed around the capabilities of the WHOI Utility Acoustic Modem (UAM) and the WHOI Micro Modem, and thus is restricted to 32 byte packets. Thus all control, sensor, and status messages described here are sized to fit into 32 bytes. Image information (such as compressed images or sidescan data) is not included in this document, but may be added in the future to support common data formats for CAD/CAC image segments or camera images. An important by-product of the introduction of the standard is that different control and display systems will be able to talk to different types of AUVs during cooperative exercises.

The scope of the protocol includes the following status and sensor messages:

- Vehicle status (position, heading, etc.)
- Environmental sensor information (depth, temperature, salinity, etc.)
- CAD/CAC target information
- Redirection and mission modification
- Control commands
- Error messages

What the Protocol Does Not Provide

There is no provision for error detection or correction within a message. It is assumed that any errors are dealt with at the transport layer. For the WHOI utility acoustic modem (UAM) and the micro modem this is correct.

- There is no provision for dealing with dropped messages. It is assumed that all messages transmitted are received. For some messages, such as information about the vehicle status, this is acceptable. It is better to retransmit new information. For other information, such as redirection commands and their acknowledgement, this presents a problem. A future revision of the micro modem firmware is supposed to be capable of retransmitting messages that must be delivered.
- There are no fields indicating source or destination addresses. It is assumed that this information is available separate from the data content, as it is in UDP and/or TCP data transmissions. The micro modem and UAM interface protocol provide this information.
- There is no inherent priority of one data type over another. Determining what information is to be sent in what order is solely under the control of the sender by how it chooses to queue data.

Terms and Conditions

Because acoustic bandwidth is a finite resource, it is useful and necessary for vehicles that share the same operating area cooperate in their utilization of this resource. Furthermore, achieving a synoptic picture of the position and status of all autonomous assets is a necessary and worthwhile goal for autonomous underwater vehicles.

This protocol is made available to the AUV community at large with the following stipulations:

- Usage of any part of the protocol requires adherence to all parts of the protocol. End users may not pick and choose which parts with which they will and will not conform. An application or system is either fully compliant or not compliant.
- Users agree to full disclosure of all message formats they are using during any cooperative exercise. All users should be able to interpret all information transmitted by any asset.
- Users may extend the protocol as they see fit, within the provisions of items 1 and 2, and with the understanding that some governing body may be required for the assignment of message numbers.

It is freely acknowledged that at some level, AUV developers are in competition with each other. What must also be recognized is that acoustic communication standards and protocols are ***not*** a suitable area for this competition to flourish. REMUS developers and its licensees are committed to open standards for acoustic communications, and publication of our existing protocol is only the first step. We recognize that at some point the standard may need to be under the purview of an independent governing body, and at an appropriate time will relinquish control.

WHOI will make a reasonable effort within its applications to support message formats developed for other assets, particularly if these formats are provided in a reasonable and timely manner.

Message Structure

Each message is defined as a "C" structure, and includes a number of sub-elements. Developers should be aware that different compilers may pack the elements of these structures differently because they insert unseen pad bytes for data alignment purposes. Compilers generally provide a method for disabling or limiting the insertion of pad bytes. There must be no hidden pad bytes between elements.

The first byte of each message indicates the message type, and it indicates the contents of the data to follow. Developers can switch() based on this value, then within the individual case statements, cast the data pointer to the appropriate message type.

For the purposes of this document, the following sizes are assumed.:

C type	Size
char	**1 byte**
int	**(not used)**
short	**2 bytes**
long	**4 bytes**
float	**4 bytes (IEEE format)**
double	**4 bytes (IEEE format)**

All data is transmitted little endian format, i.e. the least significant byte is first (Intel x86 format). Note that communications to the UAM follows a big endian format, however it does not interpret the binary data stream that it is transmitting. The micro modem communicates with an ASCII protocol, and thus is not affected by this issue. Negative integer values use a twos complement format.

Using and Extending the Protocol

Messages are formed by reading the data and compressing it to fit into the appropriate data element by simple compression algorithms, included as Appendix A. Some messages are sufficiently generic that

different vehicles can use them as is. For example, the format for transmitting bathymetry ought to be common across all platforms: it would be advantageous for other vehicles to transmit their data using this message. Other messages tend to be more vehicle specific. For example the STATE message transmitted by the REMUS class AUVs may not be suitable for vehicles of a different architecture. It is important for developers to recognize when they should reuse an existing format, and when they should extend to include a new format. Obviously, if there need to be ***any*** changes in the format, then a new format is called for, since required changes, however minor, may break existing or archived applications or data.

Appendix A contains a number of very simple compression and decompression algorithms. A new message format may chose to use these existing algorithms, or develop new ones.

When developing a new message, the developer should consult with the author regarding message number assignment. In short, cooperation is required regarding the value of the first byte.

Message Types

There are a number of different messages.

```
typedef enum
  {
  MDAT_EMPTY                = 6,
  MDAT_REDIRECT             = 7,
  MDAT_USBL_POSITION        = 8,
  MDAT_BATHY                = 9,
  MDAT_CTD                  = 10,
  MDAT_COMMAND              = 11,
  MDAT_USBL_DATA            = 12,
  MDAT_CADCAC_DETECTION     = 13,
  MDAT_STATE                = 14,
  MDAT_ERROR                = 15,
  MDAT_RANGER               = 16,
  MDAT_MLO_REQUEST          = 17,  (proposed)
  MDAT_LAST                 = 18,
  }
  MODEM_DATA;
```

Notes:

1. WHOI has obsoleted a number of different data messages, however has not reused the numbers so as to retain compatibility with archived data.

2. The USBL position message is not documented in this reference at this time. It is used in the REMUS 6000 vehicle to transmit ship position to the vehicle.

3. The USBL data message is not documented in this reference at this time. It is used in the REMUS 6000 vehicle to transmit USBL data information from the vehicle to the ship.

State

Message Number: MDAT_STATE (14)

Purpose: This message transmits basic status information from a REMUS class vehicle.

Vehicles: REMUS, REMUS 6000

Format:

```
typedef struct
  {
  unsigned char     mode;                    // MDAT_STATE
  LATLON_COMPRESSED latitude;                // 3 bytes
  LATLON_COMPRESSED longitude;
  unsigned char     fix_age
  TIME_DATE         time_date;               // 3 bytes;
  unsigned char     heading;                 // 1.5 degree resolution
  unsigned short    mission_mode_depth;      //
  unsigned long     faults;
  unsigned char     faults_2;
  unsigned char     mission_leg;
  char              est_velocity;
  char              objective_index;
  unsigned char     watts_encoded;
  LATLON_COMPRESSED lat_goal;              // 3 bytes
  LATLON_COMPRESSED lon_goal;              // 3 bytes
  unsigned char     battery_percent;
  unsigned short    gfi_pitch_oil_encoded; // 5 bits gfi,6 bits pitch,
                                           // 5 bits oil
  }
  MODEM_MSG_DATA_STATE;
```

Parameters:

mode: This one byte value indicates what type of data is contained in the remaining 31 bytes of the message.

Latitude, longitude: These values indicate the position of the vehicle at the time the message is formed. Since this message is generally formed shortly before transmission, it is the current position of the vehicle.

Fix_age: the number of seconds divided by 4 since the vehicle obtained either an acoustic or GPS fix. If the age is greater than 255*4 seconds. If the fixes older than 17 minutes (1020 seconds) are clamped to that value.

Time_date: This 3 byte bitwise encoded value that has the month, day, hour, minute, and seconds bitwise endcoded. It does NOT encode the year. This is vehicles current time and date information, including whatever local offsets for timezone and daylight savings are appropriate. Resolution is to 4 seconds.

Heading: The vehicles current heading in degrees, scaled to fit a range of 0 to 255, thus there is a resolution of about 1.4 degrees.

Mission_mode_depth: The lower 13 bits contain the depth, encoded using the Encode_deep_depth(), as listed in appendix A. The top 3 bits contain the vehicles operational mode, encoded as follows:

```
      0x0000 = Mission completed
```

```
0x2000 = Manual mode
0x4000 = Test
0x6000 = Fault
0xA000 = Redirected mission in progress
0xC000 = Normal mission operation
```

Faults, Faults_2: This Contains bitwise coded information regarding the vehicle status. The encoding the bits is vehicle specific, however there is some continuity across platforms regarding the bits in the fault word:

```
#define ST_FAIL_VOLTAGE              0x00000001
#define ST_FAIL_CURRENT              0x00000002
#define ST_FAIL_GROUND_FAULT         0x00000004
#define ST_FAIL_COMPASS              0x00000008
#define ST_FAIL_ATTITUDE_SENSORS     0x00000010
#define ST_FAIL_PITCH_MOTOR          0x00000020
#define ST_FAIL_RUDDER_MOTOR         0x00000040
#define ST_FAIL_THRUSTER_MOTOR       0x00000080
#define ST_FAIL_ACOUSTIC_NAV         0x00000100
#define ST_FAIL_DEPTH_SENSOR         0x00000200
#define ST_FAIL_LEAK                 0x00000400
#define ST_FAIL_LEAK_CONTINUITY      0x00000800
#define ST_FAIL_ENERGY               0x00001000
#define ST_FAIL_DISK_SPACE           0x00002000
#define ST_FAIL_HARDWARE             0x00004000
#define ST_FAIL_RANGER_PING          0x00008000
```

mission_leg: The number of the leg the vehicle is currently doing. This value is modulo 256, so for very long missions, this value may wrap around. Note: the numbering of redirect legs commences with the value after the last pre-programmed leg. For example, if a mission contains 10 legs, and the vehicle is redirected during the 4^{th} leg, the first leg of the redirect will be number 11. At the completion of the redirect, the vehicle will resume leg 4.

Estimated_velocity: Contains the vehicles estimate velocity, which is derived from the ADCP data if bottom lock is available, or from prop turns if it is not.

Objective_index: Numerical value indicating what type of objective (vehicle capability) the vehicle is currently trying to accomplish. This parameter is vehicle specific.

Watts_encoded: The watts the vehicle is drawing from the power supply.

Lat_goal, lon_goal: The vehicles current destination. Generally this is the endpoint of the current leg, however that is not necessarily true.

Battery_percent: Direct readout of the percentage of the available battery capacity that remains.

Gfi_pitch_oil: This value encodes 3 parameters, the ground fault status of the vehicle (in percent, 0 to 100), the vehicles pitch attitude, and the percentage of the available capacity remaining in the oil compensation system (if any). If the vehicle is not equipped with an oil compensation system, then this value is 0.

Parameter	Encode	Decode
mode	(none)	(none)
latitude	Encode_latlon()	Decode_latlon()
longitude	Encode_latlon()	Decode_latlon()
fix_age	(none)	(none)
time_date	Encode_timedate()	Decode_timedate()
heading	Encode_heading()	Decode_heading()
mission_mode_depth	Encode_depth()	Decode_depth()
faults	(none)	(none)
faults_2	(none)	(none)
mission_leg	(none)	(none)
estimated_velocity	Encode_est_velocity()	Decode_est_velocity()
objective_index	(none)	(none)
watts_encoded	Encode_watts()	Decode_watts()
lat_goal	Encode_latlon()	Decode_latlon()
lat_goal	Encode_latlon()	Decode_latlon()
battery_percent	(none)	(none)
gfi_pitch_oil	Encode_gfi_pitch_oil()	Decode_gfi_pitch_oil()

Sample:

```
Acoustic Modem 233
 From: 0 to 1 (Ack 0)
 type:State (14)
 Lat:    25N16.945'
 Lon:    77W09.856'
 Time:  Mar 4, 17:01:44
 Fix age: 00:04
 Heading: 270
 Pitch: 6 degs
 Depth: 2323.0
 Faults:24000000 0001
 Percent batt: 90 watts 500
 Leg:     4
 Est vel:1.88 M/sec 3.65 knots
 Goal:  25N16.945'   77W09.856'
 Oil: 55
 GFI: 0

 0E86FA11AD20C901
 1B4432BF47D10000
 002401042F0E7D87
 FA111620C95A200A
```

Empty Message

Message Number: MDAT_EMPTY (6)

Purpose:
The empty message is used in those instances when a system is required to transmit information, but has no information to transmit. Generally, a vehicle always has some default information that it can send, however a control station may at times be required to transmit information when it in fact has no information to transmit.

Vehicles: REMUS 6000

Format:

```
typedef struct
  {
  unsigned char     mode;                // MDAT_EMPTY
  char              spare[31];
  }
  MODEM_MSG_DATA_EMPTY;
```

Parameters:

mode: This one byte value indicates what type of data is contained in the remaining 31 bytes of the message.

spare: Zero filled array of 31 bytes to pad the structure out to 32 bytes total.

Parameter	Encode	Decode
mode	(none)	(none)
spare	(none)	(none)

Redirect Message

Message Number: MDAT_REDIRECT (7)

Vehicles: REMUS, REMUS 6000 (others?)

Purpose:
The redirect message is used by a control station to redirect a vehicle to a new waypoint or destination. It has two modes. The first is a simple transit message, and directs the vehicle to a new position at a commanded speed and depth or altitude. The second format enhances this message by adding a survey centered about the initial destination after the initial transit.

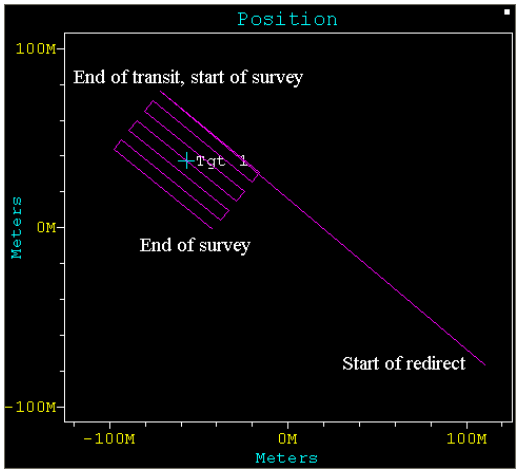

Figure 1: Redirect with Transit and Survey

An example of a redirect is shown above. The redirection starts from the lower right corner. The transit portion consists of the initial leg. The survey portion consists of the 7 rows centered about the waypoint "Tgt 1".

This message does not specify the action that should be taken at the completion of this redirect message, and may be vehicle specific. Possible actions include waiting at the completion point for additional instructions or returning to the starting point of the redirect. REMUS class vehicles take the following actions: If no additional redirects are pending, then the vehicle returns to the starting point of the redirect, and resumes the mission. If additional redirects are pending, then the vehicle prosecutes them. At the completion of all pending redirects, the vehicle returns to the starting point of the redirect and resumes the mission.

If a vehicle is given two (or more) identical redirect messages in sequence, it should ignore all but the first. However, if any parameter within the message is different the vehicle should assume that the messages are different. For example, even if only the heading is changed between two messages, the vehicle should interpret the messages as two discrete messages.

An asset may ignore or modify commands that may result in its damage or loss. This includes "goto" commands that exceed some reasonable transit range; commands to transit or survey at a depth that exceeds the assets allowed limits; or commands to survey at an altitude that exceeds the range of the onboard altimeter. An asset may, but is not required to, respond to this situation with an error message describing the error.

An asset should respond to a redirect message with a redirect acknowledge message. This is simply the redirect message that includes the starting point of the redirection.

Example redirect with survey

Format:

```
typedef struct
  {
  unsigned char     mode;               // MDAT_REDIRECT
  unsigned char     message_number;
  LATLON_COMPRESSED lat;                // Center of search area
  LATLON_COMPRESSED lon;                // Center of search area
  char              speed_depth_flags;
  unsigned short    depth_goal_encoded_transit;
  char              speed_encoded_transit;
  unsigned char     device_cmd_transit;    // Sidescan, DIDSON range.
  unsigned short    depth_goal_encoded_survey;
  char              speed_encoded_survey;
  unsigned char     device_cmd_survey;     // Sidescan, DIDSON range.
  unsigned char     num_rows;              // 0 if not rows.
  unsigned short    row_length;            // in meters
  unsigned char     spacing_0;             // in meters
  unsigned char     spacing_1;             // in meters
  char              heading_encoded;
  LATLON_COMPRESSED lat_start;      // ack only, where redirect started
  LATLON_COMPRESSED lon_start;
  char              spare[3];
  }
  MODEM_MSG_DATA_REDIRECT;
```

Parameters:

mode: This one byte value indicates what type of data is contained in the remaining 31 bytes of the message.

Message_number: (obsolete, should be 0).

Lat, lon: If this is a single “goto” waypoint instruction, these values are the latitude and longitude of the survey area. If the message also encodes a survey, then the position is the *center* of the survey area.

Speed_depth_flags: This parameter encodes the speed mode and depth mode of the vehicle during both the transit and survey phase of the redirect. The following bitwise coding is used:

Binary value	Hex value	Meaning
XXXXX000	0x00	transit in depth mode
XXXXX001	0x01	transit in altitude mode
XXXX0XXX	0x00	transit speed is in RPM
XXXX1XXX	0x08	transit speed is in meters/second
X000XXXX	0x00	survey in depth mode
X001XXXX	0x00	survey in altitude mode
0XXXXXXX	0x00	survey speed is in RPM
1XXXXXXX	0x80	survey speed is in M/sec

Key: X = Don't care
1 = Bit set
0 = Bit cleared

Depth_goal_encoded_transit:
Depth_goal_encoded_survey: These parameters set the depth or altitude that the vehicle uses during the transit to the survey point, and for the survey itself. Which value they represent is indicated in the speed_depth_flags parameter described above. The values are encoded/decoded using Encode_depth() and Decode_depth() even when the value is an altitude.

Speed_encoded_transit:
Speed_encoded_survey: This two parameters sets the speed of the vehicle for the transit and survey respectively. The value may be specified in either RPM or in meters per second. The speed_depth_flags parameter (described above), is used to indicate which method should be used to interpret the data.

Device_cmd_transit:
Device_cmd_survey: If the vehicle is equipped with a sidescan, this parameter encodes the range.

Num_rows: The number of rows in the survey. A value of 0 indicates that there is no survey portion.

Row_length: Length of the row in meters.

Spacing_0: distance in meters between the 1st and 2nd, 3rd and 4th, 5th and 6th etc. , rows.

Spacing_1: Distance in meters between the 2nd and 3rd, 4th and 5th, 6th and 7th, etc., rows.

Heading_encoded: This parameter is only used when a redirect contains a survey segment. It indicates the direction of the first leg of the survey, scaled to fit a range of 0 to 255, thus there is a resolution of about 1.4 degrees.

Lat_start, lon_start: These parameters are only used by the vehicle to acknowledge the redirect message. It is sent back to the source of the redirect message to indicate that the message was received, and where the redirection will start from. If the vehicle is currently in a redirection, the start will be at the end of the last redirect. If the vehicle is not in a redirect, the start will be from the current vehicle location.

Spare: These 3 bytes are reserved for future use and must be set to 0.

Parameter	Encode	Decode
mode	(none)	(none)
spare	(none)	(none)
latitude	Encode_latlon()	Decode_latlon()
longitude	Encode_latlon()	Decode_latlon()
speed_depth_flags	(none)	(none)
depth_goal_encoded_transit	Encode_depth()	Decode_depth()
speed_encoded_transit	Encode_speed()	Decode_speed()
device_cmd_survey	(none)	(none)
num_rows	(none)	(none)
row_length	(none)	(none)
spacing_0	Encode_est_velocity()	Decode_est_velocity()
spacing_1	(none)	(none)
heading_encoded	Encode_heading()	Decode_heading()
lat_start	Encode_latlon()	Decode_latlon()
lon_start	Encode_latlon()	Decode_latlon()
spare	(none)	(none)

Sample:

```
Acoustic Modem 219
 From: 0 to 1 (Ack 0)
 type:Redirect (7)
 Lat:    25N16.511'
 Lon:    77W09.441'
           Transit     Survey
 Alt:        10.0 Alt: 10.0
 Speed: 2.0 m/s 2.0 m/s
 Cmd:       100         100
 Num Rows:               11
 Row length:            230
 Spacing:            20  20
 Heading               45

 07522CF9113D20C9
 9964003D6464003D
 640BE60014142035
 F911EF21C9000000
```

Bathymetry Data

Message Number: MDAT_BATHY (9)

Vehicles: REMUS, REMUS 6000

Purpose: This message is for the transmission of bathymetry information as collected by a vehicle. Each message contains 3 data sets, each with a latitude, longitude, vehicle depth and altitude. Ideally, these 3 sets should each be collected at different locales, and not be just the most recent data collected, so as to provide as much spatial coverage as possible.

Format:

```
typedef struct
  {
  unsigned char     mode;              // MDAT_BATHY
  unsigned char     spare;
  unsigned short    depth[3];
  unsigned short    altitude[3];
  LATLON_COMPRESSED latitude[3];
  LATLON_COMPRESSED longitude[3];
  }
  MODEM_MSG_DATA_BATHY;
```

Parameters

mode: This one byte value indicates what type of data is contained in the remaining 31 bytes of the message.

spare: unused byte

Depth: The encoded depth in meters.

Altitude: The encoded vehicle altitude off the bottom in meters. The sum of the vehicle depth and altitude is the water depth.

Latitude, longitude: The position where the measurement was made.

Parameter	Encode	Decode
mode	(none)	(none)
spare	(none)	(none)
depth	Encode_depth()	Decode_depth()
altitude	Encode_hires_altitude()	Decode_hires_altitude()
latitude	Encode_latlon()	Decode_latlon()
longitude	Encode_latlon()	Decode_latlon()

Sample:

```
Acoustic Modem 233
 From: 0 to 1 (Ack 0)
 type:State (14)
 Lat:     25N16.945'
 Lon:     77W09.856'
 Time:  Mar 4, 17:01:44
 Fix age: 00:04
 Heading: 270
 Pitch: 6 degs
 Depth: 2323.0
 Faults:24000000 0001
 Percent batt: 90 watts 500
 Leg:      4
 Est vel:1.88 M/sec 3.65 knots
 Goal:  25N16.945'   77W09.856'
 Oil: 55
 GFI: 0

 0E86FA11AD20C901
 1B4432BF47D10000
 002401042F0E7D87
 FA111620C95A200A
```

CTD Message

Message Number: MDAT_CTD (10)

Vehicles: REMUS, REMUS 6000

Purpose: Transmits 2 points of CTD data, including salinity, temperature, depth, and sound speed from the vehicle.

Format:

```
typedef struct
  {
  unsigned char     mode;           // MDAT_CTD
  unsigned char     spare;
  unsigned char     salinity[2];
  unsigned char     temperature[2];
  unsigned short    depth[2];
  unsigned char     sound_speed[2];
  LATLON_COMPRESSED latitude[2];
  LATLON_COMPRESSED longitude[2];
  char spare2[8];
  }
  MODEM_MSG_DATA_CTD;
```

Parameters:

mode: This one byte value indicates what type of data is contained in the remaining 31 bytes of the message.

spare: unused byte

salinity: The encoded salinity.

Temperature: The encoded temperature

depth: The encoded depth in meters.

Sound_speed: The encoded sound speed.

Latitude, longitude: The position where the measurement was made.

Spare2: additional unused bytes

Parameter	Encode	Decode
mode	(none)	(none)
spare	(none)	(none)
salinity	Encode_salinity()	Decode_salinity()
temperature	Encode_temperature()	Decode_temperature()
depth	Encode_depth()	Decode_depth()
sound_speed	Encode_sound_speed()	Decode_sound_speed()
latitude	Encode_latlon()	Decode_latlon()
longitude	Encode_latlon()	Decode_latlon()

Sample:

```
Acoustic Modem 233
 From: 0 to 1 (Ack 0)
 type:Bathymetry (9)

 Lat:    25N16.945'
 Lon:    77W09.825'
 Depth: 2319.0
 Alt:   21.9
 Bathy: 2340.9

 Lat:    25N16.945'
 Lon:    77W09.834'
 Depth: 2321.0
 Alt:   21.7
 Bathy: 2342.7

 Lat:    25N16.946'
 Lon:    77W09.844'
 Depth: 2322.0
 Alt:   21.2
 Bathy: 2343.2

 094F431145114611
 90087D08490886FA
 1186FA1187FA11C5
 20C9BE20C9B620C9
```

Command Message

Message Number: MDAT_COMMAND

Vehicles: REMUS, REMUS 6000

Purpose: This message is used to transmit commands to the vehicle. Generally, there is a single parameter (the command), however some commands take an additional parameter.

Format:

```
typedef struct
  {
  unsigned char  mode;
  unsigned char  spare;
  unsigned short command;
  char           parameter[28];
  }
  MODEM_MSG_DATA_COMMAND;
```

Parameters

mode: This one byte value indicates what type of data is contained in the remaining 31 bytes of the message.

spare: unused byte

command: There are a number of two byte commands:

0: Not used
1: Abort to end position of mission
2: Abort immediately
3: Start mission
4: Release descent weight (REMUS 6000 only)
5: Release ascent weight (REMUS 6000 only)
6: Release pickup float (REMUS 6000 only)
7: Allow modem communications to serve as a ranger ping
8: Disable allowing modem communications to serve as a ranger ping
9: Set USBL pitch offset (REMUS 6000 only)
10: Set USBL roll offset (REMUS 6000 only)
11: Enable modem USBL (REMUS 6000 only)
12: Disable modem USBL (REMUS 6000 only)
13: Set USBL sound speed (REMUS 6000 only)
14: Accept next fix (REMUS 6000 only)
15: Dump redirect commands
16: Get GPS fix
17: Abort to mission start location
18: Abort to destination (destination follows as a parameter)
19: Dump redirect commands except current
20: Abort, Drop Ascent weight, power to surface (REMUS 6000 only)

Note: Messages 64 to 95 (inclusive) assigned to Frank Heirtzler, Foster-Miller, Inc. for crawlers.

	Parameter
Move to lat/long(not redirect)	Lat, lon, speed, search parameters(type, #rows, length, spacing)
Move polar	Distance, heading, speed, search parameters

Move polar-relative	Distance, relative heading, speed, search parameters
Pause/resume	None
Poll status(request sensor information)	data type
Payload data	Data type, data string (payload specific)
Instructions to payload	ROBO command string (payload specific)
Move at current heading	Distance, speed
Wakeup handshake	None
Set LBL parameters	LBL initialization parameters

Parameter	Encode	Decode
mode	(none)	(none)
spare	(none)	(none)
command	(none)	(none)
parameter	(none)	(none)

CADCAC Message

Message Number: MDAT_CADCAC (13)

Vehicles: REMUS

Purpose: This message is used to transmit CADCAC (Computer Aided Detection/Computer Aided Classification) information. Three detections are transmitted within each message. Each detection has a a lat/lon position, 3 scores, and a target id. The position is self-explanitory. The 3 scores represent the results of 3 different processing algorithms. The scores represent the degree of confidence the individual algorithms have in declaring a mine like object a mine. A score of 0 indicates no confidence, 100 indicates a high degree of confidence. A perfect score is 300, all three algorithms at 100%.

Format:

```
typedef struct
  {
  LATLON_COMPRESSED lat;
  LATLON_COMPRESSED lon;
  unsigned char     score[3];   // 0-255
  unsigned char     target_id;  // modulo 256
  }
  CADCAC_DET_COMP;

typedef struct
  {
  unsigned char     mode;
  unsigned char     spare;
  CADCAC_DET_COMP   detection[3];
  }
  MODEM_MSG_DATA_CADCAC_DET;
```

Parameters

mode: This one byte value indicates what type of data is contained in the remaining 31 bytes of the message.

spare: unused byte

Detection: Encoded information regarding a single CADCAC detection. The following parameters are within this structure:

> **Latitude, longitude:** Location of the detection.
> **Score:** 3 three scores received from the system. Range is 0-255.
> **Target_id:** The lower 8 bits of the 32 bit target ID generated by the CADCAC system.
> (note: An anticipated future revision of this specification will use this as a fourth score).

Parameter	Encode	Decode
mode	(none)	(none)
spare	(none)	(none)
latitude, longitude	Encode_latlon()	Decode_latlon()
score	(none)	(none)
target_id	(none)	(none)

Sample:

```
Acoustic Modem 65
 From: 4 to 0 (Ack 0)
 type:CADCAC Detect (13)
 Latitude:    32N40.171'
 Longitude:  117W11.848'
 Scores:     94.9% 17.6%, 6.3%
 Target ID:  0
 Latitude:    32N40.204'
 Longitude:  117W11.880'
 Scores:     94.9% 17.6%, 6.3%
 Target ID:  0
 Latitude:    32N40.228'
 Longitude:  117W11.881'
 Scores:     94.9% 17.6%, 6.3%
 Target ID:  0

 0D004E3B17DBA8AC
 F22D1000673B17C2
 A8ACF22D10007A3B
 17C1A8ACF22D1000
```

Error Message

Message Number: MDAT_ERROR_MSG (15)

Vehicles: REMUS, REMUS 6000

Purpose: This message is used to transmit clear text messages from an asset to a man in the loop. They are not intended for computer parsing.

Format:

```
typedef struct
  {
  unsigned char     mode;
  char              message[31];
  }
  MODEM_MSG_ERROR_MSG;
```

Parameters

mode: This one byte value indicates what type of data is contained in the remaining 31 bytes of the message.

message: 31 byte clear text ASCII message. If the message is less than 31 bytes, it must be null terminated, However if all 31 bytes are used, null termination is not required.

Parameter	Encode	Decode
mode	(none)	(none)
message	(none)	(none)

MLO Request (preliminary information, subject to change)

Message Number: MDAT_MLO_REQUEST (17)

Vehicles: REMUS (and others)

Purpose: This message is used by re-acquisition assets and a "master" to coordinate prosecution of CADCAC targets by the re-acquiring assets.

Format:

```
typedef struct
  {
  LATLON_COMPRESSED lat;
  LATLON_COMPRESSED lon;
  unsigned char     flags
#define MLO_FLAG_UNUSED    0x00
#define MLO_FLAG_REQUEST   0x01 // re-acq requesting target
#define MLO_FLAG_PROSECUTE 0x02 // master says ok,
#define MLO_FLAG_BUSY      0x03 // master says later
#define MLO_FLAG_COMPLETED 0x04 // reacq/master indicates completed
#define MLO_FLAG_FAILURE   0x05 // re-acq indicates failure
#define MLO_FLAG_MODE_MASK 0x07

#define MLO_FLAG_ALT       0x00   // min/max_depth_alt is an altitude
#define MLO_FLAG_DEPTH     0x08   // min/max_depth_alt is a depth
  unsigned char     duration_minutes;
  unsigned short    min_depth_alt;
  unsigned short    max_depth_alt;
  unsigned char     radius_meters;
  }
  MLO_REQUEST;

typedef
  {
  unsigned char     mode;
  MLO_REQUEST       request[2];
  char              spare[5];
  }
  MODEM_MLO_REQUEST_MSG;
```

Parameters:

mode: This one byte value indicates what type of data is contained in the remaining 31 bytes of the message.

Flags: The bottom 3 bits are used to indicate the type of transaction. Essentially this is an enumerated value:
MLO_FLAG_UNUSED: None of the other data is valid.
MLO_FLAG_REQUEST: Re-acq asset is requesting to prosecute the MLO at the lat/lon indicated
MLO_FLAG_PROSECUTE: Master is granting permission to prosecute MLO for the duration listed in the message.
MLO_FLAG_BUSY: Master is indicating that the location is question is not available for the duration listed in the message.

MLO_FLAG_COMPLETED: Sent by re-acquisition asset when it has successfully completed prosecuting an MLO. It may also be sent by the master to a re-acquisition asset to tell it not to prosecute the MLO at that location, and to remove it from its "list" of potential MLOs.
MLO_FLAG_FAILURE: Sent by asset to indicate that it has failed in its attempt to re-acquire and identify a particular MLO. This might be used (for example) by an re-acq asset that has had a critical failure and has been forced to prematurely abort its mission.

MLO_FLAG_ALT and MLO_FLAG_DEPTH are used to indicate whether the provided depths/altitudes are altitudes or depths.

Latitude, longitude: The position where MLO is located.

Duration_minutes: The meaning of this value is function of mode.
MLO_FLAG_REQUEST: Duration that the requesting asset would like the area
MLO_FLAG_PROSECUTE: Duration for which the requesting asset is granted an area
MLO_FLAG_BUSY: Duration of the busy period; how long before the area is free.
MLO_FLAG_COMPLETED: Duration ignored, should be 0.
MLO_FLAG_FAILURE: Duration ignored, should be 0.

Min/max_depth_alt: The encoded depth/altitude in meters.

Radius: Distance from the position for which the vehicle has rights.

Parameter	Encode	Decode
mode	(none)	(none)
latitude, longitude	Encode_latlon()	Decode_latlon()
min/max_depth_alt	Encode_depth()	Decode_depth()

This is a preliminary definition of this message and may change

Appendix A: Encode/Decode Routines

In order to provide as efficient data transmission as possible, the data is encoded using a variety of formats. This section describes the methods used for compression and decompression of these parameters.

```
 /* Latitude/Longitude
  * Each are encoded into 3 bytes, giving a resolution of about 2
  * meters. The compressed form is stored in a LATLON_COMPRESSED
  * struct, which is 3 bytes.
  */
typedef struct
  {
  char compressed[3];
  }
  LATLON_COMPRESSED;

typedef union
  {
  long               as_long;
  LATLON_COMPRESSED  as_compressed;
  }
  LONG_AND_COMP;

LATLON_COMPRESSED Encode_latlon(double latlon_in)
  {
    /* Latitude and longitude are compressed into 3 byte values each.
     */
  LONG_AND_COMP out;
  double encoded;

  encoded = latlon_in * ((double)(0x007FFFFFL)/180.0);
  encoded += (encoded > 0.0) ? 0.5 : -0.5;   // deal with truncation
  out.as_long = encoded;
  return(out.as_compressed);
  }

double Decode_latlon(LATLON_COMPRESSED latlon_in)
  {
  LONG_AND_COMP in;

  in.as_long = 0;                //Clear the MSB
  in.as_compressed = latlon_in;
  if (in.as_long & 0x00800000L)  // sign extend
    in.as_long |= 0xFF000000L;
  return(double(in.as_long) * (180.0/(double)(0x007FFFFFL)));
  }

  /* Encodes 0-360 degree heading into a 0-255 (1 byte)
   */
unsigned char Encode_heading(float heading)
  {
  return((unsigned char)(heading * 255.0/360.0 + 0.5));
  }
double Decode_heading(unsigned char heading)
  {
```

```
  return((double)heading * 360.0 / 255.0 );
  }

  /* Encodes velocity in meters per second into a single byte with a
   * resolution of 2.5 cm/sec. Input range is 0- ~6 meters/second (12
   * knots).
   */
char Encode_est_velocity(float est_velocity)
  {
  return((char)(est_velocity * 25.0));
  }

float Decode_est_velocity(char est_velocity)
  {
  return(est_velocity/25.0);
  }

  /* Code-decode salinity in range of 0-45 parts per thousand at a
   * resolution of 0.1 part per thousand.
   */
unsigned char Encode_salinity(float salinity)
  {
  float output;

  if (salinity < 20.0)
    return(0);
  else
    {
    output = (salinity - 20.0) * 10;
    if (output > 255)
      output = 255;
    return((unsigned char)output);
    }
  }

float Decode_salinity(unsigned char sal)
  {
  if (sal == 0)
    return(sal);
  return(((float)sal/10.0) + 20.0);
  }

unsigned short Encode_depth(float depth)
  /*  0  -100   meters:    0-1000 (10 cm resolution)
   * 100 -200   meters: 1001-1500 (20 cm resolution)
   * 200 -1000  meters: 1501-3100 (50 cm resolution)
   * 1000-6000  meters: 3101-8100 (1  meter resolution)
   */
  {
  if (depth < 0)
    return(0);
  if (depth < 100)
    return((depth+.05)           * 1.0/0.1);
  if (depth < 200)
    return(((depth-100)  + 0.1 ) * 1.0/0.2 + 1000);
  if (depth < 1000)
```

```
    return(((depth-200)  + 0.25) * 1.0/0.5 + 1500);
  if (depth < 6000)
    return(((depth-1000) + 0.5 ) * 1.0/1.0 + 3100);
  return(8100);
  }

float Decode_depth(unsigned short depth)
  {
  /*  0  -100   meters:    0-1000 (10 cm resolution)
   * 100 -200   meters: 1001-1500 (20 cm resolution)
   * 200 -1000  meters: 1501-3100 (50 cm resolution)
   * 1000-6000  meters: 3101-8100 (1  meter resolution)
   */
  depth &= DEPTH_MODE_MASK;   // Only using bottom 13 bits.
  if      (depth <= 1000)
    return(depth               * 0.1/1.0);
  else if (depth <= 1500)
    return(100  + (depth-1000) * 0.2/1.0);
  else if (depth <= 3100)
    return(200  + (depth-1500) * 0.5/1.0);
  else if (depth <= 8100)
    return(1000 + (depth-3100) * 1.0/1.0);
  else
    return(6000);
  }

unsigned char  Encode_temperature(float temperature)
  {
  if (temperature < -4)
    temperature = -4;
  temperature += 4;
  temperature = temperature * 256.0/40.0 + 0.5;
  if (temperature > 255)
    temperature = 255;
  return((unsigned char)temperature);
  }

float Decode_temperature(unsigned char temperature)
  {
  return(temperature * 40.0/256.0 - 4.0);
  }

unsigned char Encode_sound_speed(float sound_speed)
  {
  return((sound_speed - 1425.0) * 2);
  }

float Decode_sound_speed(unsigned char sound_speed)
  {
  return((float)sound_speed/2.0 + 1425.0);
  }

unsigned short Encode_hires_altitude(float alt)
  /* 10 cm resolution to 655 meters.
   */
  {
```

```
  alt *= 100;
  if (alt > 65535.0)
    return(65535U);
  else if (alt < 0)
    return(0);
  else
    return((unsigned short)alt);
  }

float Decode_hires_altitude(unsigned short alt)
  {
  return((float)alt/100.0);
  }

unsigned short Encode_gfi_pitch_oil(float gfi, float pitch, float oil)
  {
    /* We are encoding 3 parameters that are somewhat specific to
     * the REMUS 6000 into 2 bytes.
     *   GFI= 5 bits: 0-100 (3.3 resolution)
     *   OIL= 5 bits: this gives us resolution of 3.3 percent
     *   Pitch=6 bits; 180 into 64, resolution of 3 degrees.
     */
  unsigned short result;
  unsigned short temp;

  if (gfi < 0)             // 5 bits of gfi
    gfi = 0;
  else if (gfi > 100)
    gfi = 100;
  result = (unsigned short)gfi * 31.0/100.0;

  if (oil < 0)             // 5 bits of oil
    oil = 0;
  else if (oil > 100)
    oil = 100;
  oil *= 31.0/100.0;
  result |= ((unsigned short)oil << 5);

  if (pitch > 90)          // 6 bits of pitch
    pitch = 90;
  else if (pitch < -90)
    pitch = -90;
  pitch *= 63.0/180.0;  // Scale to 6 bits;
  if (pitch > 0.0)
    pitch += 0.5;
  else if (pitch < 0)
    pitch -= 0.5;
  temp = ((short)pitch << 10);
  result |= temp & 0xFC00;

  return(result);
  }

void Decode_gfi_pitch_oil(unsigned short gfi_pitch_oil, float *gfi,
  float *pitch, float *oil)
  {
```

```
  unsigned short temp;
  short temp_pitch;

  temp = (gfi_pitch_oil & 0x001F) * 100.0/31.0;
  *gfi = temp;

  temp = (gfi_pitch_oil & 0x03E0) >> 5;
  *oil = temp * 100.0/31.0;

  temp_pitch = ((short)(gfi_pitch_oil & 0xFC00)) >> 10;
  *pitch = temp_pitch * 180.0/63.0;
  }

typedef struct
  {
  char compressed[3];
  }
  TIME_DATE;

typedef union
  {
  long      as_long;
  TIME_DATE as_time_date;
  }
  TIME_DATE_LONG;

TIME_DATE Encode_time_date(long secs_since_1970)
  {
    /* The time is encoded as follows:
     *    Month 4 bits
     *    Day   5 bits
     *    hour  5 bits
     *    Min   6 bits
     *    Secs  4 bits   // 4 secs per bit
     * The year is not encoded.
     */
  struct tm tm;
  TIME_DATE_LONG comp;

    /* Note: substitute localtime() for local timezone
     * instead of GMT.
     */
  tm = *gmtime(&secs_since_1970);
  comp.as_long  = (unsigned long)tm.tm_sec    >>  2;
  comp.as_long += (unsigned long)tm.tm_min    <<  4;
  comp.as_long += (unsigned long)tm.tm_hour   << (4 + 6);
  comp.as_long += (unsigned long)tm.tm_mday   << (4 + 6 + 5);
  comp.as_long += (unsigned long)(tm.tm_mon+1)   << (4 + 6 + 5 + 5);

  return(comp.as_time_date);
  }

long Decode_time_date(TIME_DATE input, short *mon, short *day, short
*hour, short *min, short *sec)
  TIME_DATE_LONG comp;
```

```
  comp.as_long = 0;
  comp.as_time_date = input;

  *mon  = (short)((comp.as_long >> (4 + 6 + 5 + 5)) & 0x000F);
  *day  = (short)((comp.as_long >> (4 + 6 + 5))     & 0x001F);
  *hour = (short)((comp.as_long >> (4 + 6))         & 0x001F);
  *min  = (short)((comp.as_long >> (4))             & 0x003F);
  *sec  = (short)(((comp.as_long )                  & 0x000F) * 4);

      /* It is possible to force this routine to return the
       * seconds since 1970. Left as an exercise for the reader…
       */
  return(0);
  }

unsigned char Encode_watts(float volts, float amps)
  {
      /* Resolution is 4 watts, max is unsaturated is 1018 watts.
       */
  float watts = (volts * amps)/4;
  if (watts > 255)
    watts = 255;
  else if (watts < 0)
    watts = 0;          // ok, not possible...
  return((unsigned char)watts);
  }

float Decode_watts(unsigned char watts_encoded)
  {
  return((float)watts_encoded * 4.0);
  }


typedef enum
  {
  SPEED_MODE_RPM   = 0,
  SPEED_MODE_MSEC  = 1,
  SPEED_MODE_KNOTS = 2,
  }
  SPEED_MODE;

char Encode_speed(SPEED_MODE mode, float speed)
  {
      /* Max RPM:   2540
       * Max Speed: 4.2 meters/sec (8.1 knots)
       */
  switch(mode)
    {
    case SPEED_MODE_RPM:
        // Clamp to avoid errors.
      speed /= 20.0;
      if (speed > 127)
        speed = 127;
      else if (speed < -127)
        speed = -127;
      break;
```

```
    case SPEED_MODE_KNOTS:  // Convert to m/sec
      speed *= 0.5144444;   // Fall thru...

    case SPEED_MODE_MSEC:
      speed *= 30;
      if (speed > 127)
        speed = 127;
      else if (speed < -127)
        speed = -127;
      break;
    }
  return((char)speed);
  }

float Decode_speed(SPEED_MODE mode, char speed)
  {
  switch(mode)
    {
    case SPEED_MODE_RPM:
      return(speed * 20.0);

    case SPEED_MODE_MSEC:
      return((float)speed/30.0);

    case SPEED_MODE_KNOTS:
      // "Knots mode not supported by modem codec"
      return(0);
    }
  return(0);
  }
```

Appendix B: Changes

Feb 16, 2004: Added preliminary definition of MLO REQUEST message. Reversed the order of this appendix, so that the newest changes were at the top.

October 12, 2003: Added “missing” message in list of commands. Set USBL sound speed was missing from the list of commands, resulting in the documentation for commands 14-18 being off by 1. (They were listed as 13-17). Added commands 19 (dump redirect) and 20 (abort, drop descent, power to surface) as well.

September 19, 2003: Fixed description of BCD_LL type in Ranger message. Description had said it was 4 bytes, but was really 5.

August 28, 2003: Added description of the RANGER message format.

August 3, 2003: Added mention that negative integer values use twos complement format.

April 28, 2003: Added Foster Miller command list.

April 25, 2003: Added samples for some messages.

April 22, 2003: Added missing definition of TIME_DATE structure.

April 11, 2003: Fixed errors in COMMAND message, the structure name was wrong, and the command number was listed as an unsigned char, it is actually an unsigned short. This should not affect any applications. Also, the CADCAC detection message listed the score range as 0 to 100. It is actually 0-255 for each score.

February 28, 2003: Fixed estimated velocity codec. Original specification inadvertently used the obsolete scaling of 40 counts per meter/sec. The correct scaling is 25 counts per meter/second. This gives a range of +/- 5.08 meters/sec (9.8 knots).