AvTrak 6 Driver Application Notes

Contents

1 Introduction

This document exists to serve as a manual for the implementation of the Sonardyne AvTrak 6 modem/beacon set on the Dorado class AUV systems. The AvTrak 6 is a WideBand®2 underwater acoustic transceiver as well as a transponder. It can asynchronously be tracked, range other transponders, and can send and receive data transmissions. This flexibility suits two needs on the AUV: to serve as a tracking beacon compatible with MBARI shipboard tracking systems, and to serve as an acoustic communications link between the vehicle and vessel. These operations are critical to AUV Mapping missions, in which mission instructions and navigation fixes are required for the high level of navigation required to produce 1-meter resolution hydrography.

1.1 Modification History

February 4th, 2015: First draft edition completed.

2 Technical Description

2.1 System Overview

This implementation is targeted at building a system which can accommodate robust acoustic communications and double as a tracking solution. On board the AUV, the AvTrak is provided a wet connection which provides external power and communication via RS232 to the Main Vehicle Computer. The AvTrak 6 also carries an internal battery, which in this case is a backup power supply for beacon tracking in the event of a main bus power failure.

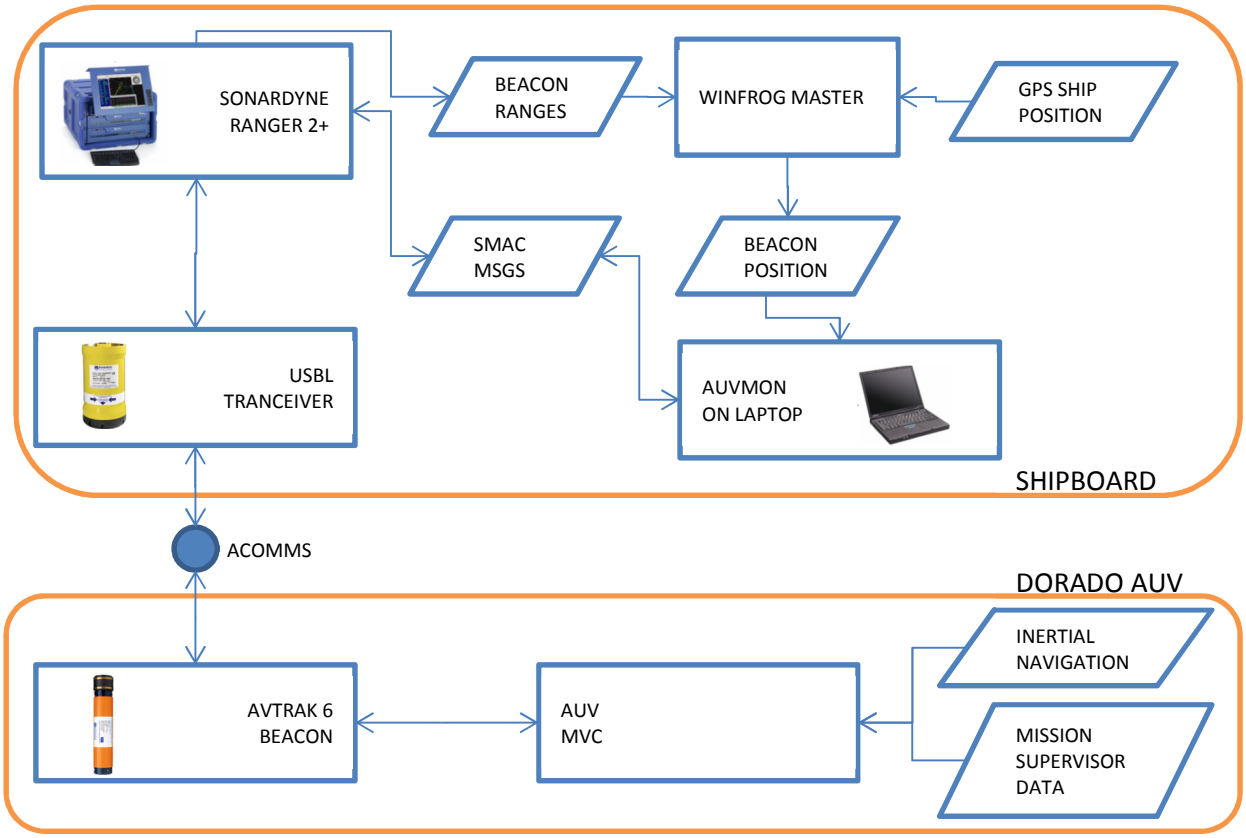

Figure 1: System overview

In order for this system to operate, the ship must have compatible equipment to provide functionality desired at the sub-sea platform. See Figure 1 for system diagram. In order to provide navigation fixes, the ship must have a Ranger 2 or newer tracking system with a Sonardyne tracking beacon. That system may also have a GPS feed, but most MBARI ships also possess computers which run Winfrog navigation software currently used to geo-colocate the tracking target with ranges provided by the Sonardyne USBL system. A third piece of software, "AUVmon", exists to collect the vehicle positions and formulate messages that will be sent via the SMS system. These messages can also contain mission specific commands such as mission aborts, start survey command, and other behavior controlling messages.

2.2 Data Transfer Protocol

2.2.1 SMS Messaging Constraints

Before discussing protocol, we must establish the constraints presented by using the Sonardyne Messaging Service. Please refer to both the documents, *Sonardyne Command Language* and *SMS Telemetry Operation with Ranger USBL and AvTrack 2*, for reference. Developers at Sonardyne have attempted to allow SMS messages and normal tracking operations to run asynchronously. However, testing has observed that this is not totally true. The constraints determined are as follows:

- SMS messages must be under 128 characters in length, and can only contain ASCII printable variables. (Decimal byte values 32-126).
- When listening for an unsolicited SMS message from the ranger software, this operation blocks all tracking for the period you choose to listen for or the first message received.
- SMS messages issued on from the topside client do not get transmitted immediately, and are delayed for the end of the last tracking cycle. This cycle terminates on a reply from a beacon or the timeout associated with the beacon.
- In standard SMS exchanges, the subsea beacon can be interrogated for messages in its buffer. Those messages must be buffered at the time the initial SMS is sent from the Ranger Software.

2.2.2 AUV Messaging Operational Requirements

- During descent, the vehicle must be tracked as often as the environment allows and SMAC messages containing these navigation fixes must be sent down to the AUV to constrain the error estimates of the INS solution until it gains velocity updates from its DVL.
- Reliable SAN messages must be transmitted at key times, such as the vehicle's arrival at the start of the survey lines, which culminate in a "Start of Survey" command.
- In the previous configuration using independent modem/tracking systems, the vehicle would transmit a message at the start and end of survey lines.
- The vehicle must be ready at any time to receive an "abort mission" command acoustically.

2.2.3 Communication Protocol

Given the operational requirements and the constraints imparted by the Ranger software, there are only a few ways in which we can proceed regarding the communications protocol between AUVmon and the MVC, using Ranger and the AvTrak as communication devices.

The chosen model incorporates changes to the existing methods. The main protocol is a call and response routine mastered by the AUVmon software. Acoustic interactions from AUVmon will always illicit an unsolicited separate response from the MVC that will transmit within a short period of time. During this time AUVmon will interrupt tracking for an SMS listen operation. The user controller AUVmon will have to set the period to listen for, but it should be set to minimum as the two-way travel time for the current depth estimate of the AUV. To re-iterate, that listening operation will block the tracking loop for a defined amount of time or until the reply has successfully arrived. Given this, all SMS exchanges between the AvTrak and USBL Ranger should be configured with flag A1, which is SMS_ACK only.

2.3 SMS Messaging

By using the Sonardyne Messaging Service (SMS), we enable new and support existing functionalities that exist in the present Dorado mapping systems. Specifically, the AvTrak can accomplish acoustic modem activities, but SMS is also the vessel for transmitting commands interpreted by the beacon itself.

2.3.1 Mission Abort and Burn Wire Release

Two commands supported by the AvTrak 6 that are directly executed onboard the beacon are the MISSION ABORT and BURN commands. Each beacon has to be individually configured to support these functions. Both functions create voltage potentials on the wet connector. Please see the *UM-8220* Manual for more detail. Specific to this implementation for Dorado Mapping vehicles, the burn wire release functionality will be enabled. This is a command issued by AUVmon, and the custom command should be formed as such:

```
SMS:1013;W1,P0,C13,A1,R0|CMD:BURN
```

Here, `1013` is the address of the AvTrak 6. If you are already communicating with the beacon, the wakeup flag (`W0`) can be removed from the string as well. Upon reception of this command the AvTrak should provide +12VDC between pins 2 and 4. This command is also sent through the RS232 port talking to the driver. The driver currently puts an entry into the Syslog and then executes the abort plan through the layered control IF.

An important note for future developers is that the current Ranger system will hold the last command number (here C13) in memory. Future SMS message sent without a command number will the last issued command number. Command number 13 is a special case, and only commands recognized by the receiving unit will be forwarded out through its command port. So regular SMS commands should always specify the command number for reliable subsea performance.

2.3.2 NMEA Style Informational and Command Messages

Other than burn-wire and abort commands, all other SMS messages will be formed in a NMEA ASCII message format, comma-delimited. For all three record types there are a common set of fields as described in the table below. Also, all messages end with a standard XOR checksum.

FIELD	CHARS	EXAMPLE
START $	1	$
TALKERID	2	MB
TYPE	3	SAN, STA, or CMD
PROTOCOL VERSION	1	Integer value
TIME_S	10	1421953941
TIME_MS	3	999
	25	

The TALKERID field can be any of the following two letter fields, and can be expanded if needed in the .h file: {SF (Surface), M1 (Mapper 1), M2 (Mapper 2)}. The code will not reject unknown TALKERID values.

2.3.2.1 Surface Aided Navigation Messages (SAN)

Surface Aided Navigations messages are intended to send information down to be ingested by the position estimation algorithms of the Kearfott inertial navigation system. Aside from the header fields, the following table describes the

FIELD	CHARS	EXAMPLE	Summary
wfTime	10		
wfLat	8	-90.00000	
wfLong	9	-121.95652	
wfSlant	7	5000	
	38	**TOTAL MSG LENGTH**	**66**

Sample String:

```
$MBSAN,1,1421953941,999,1002,36.75475,-121.91319,5000.21*3F
```

2.3.2.2 Status Messages (STA)

Status messages are the standard reply for almost all messages sent from the surface. This is a message composed of vehicle status information, including position, heading and MVC humidity.

FIELD	CHARS	EXAMPLE	Summary
auvLat	8		
auvLong	9		
auvHeading	6	359	
auvDepth	7	5000.00	
auvAlt	7		
auvVoltage	5	28.75	
auvHUMID	5	75.79	
	54	**TOTAL MSG LENGTH**	**82**

Sample String:

```
$MBSTA,1,1421953941,999,36.75475,-121.91319,359.011,5999.99,30.00,100.0*3C
```

2.3.2.3 Command Messages (CMD)

Command messages are short messages meant to provide mission critical instruction to the vehicle, as well as provide requests from the vehicle to the surface.

FIELD	CHARS	EXAMPLE	Summary
CmdType	1	See CmdType Enum (0-F)	
	2	**TOTAL MSG LENGTH**	**30**

Sample String: `$M1CMD,1,1422962612,541,3*09` *(Mission Abort)*

3 Software Description

3.1 Driver/Server

The QNX software designed to interface with the AvTrak is modeled in many ways by its functional predecessor, the Benthos Modem software. Featuring the standard Driver / Server model, and message passing through the SharedData "Output" mechanisms, the server responds to taskIF calls, leaving the driver to monitor the hardware serial interface and respond to messages issued from the surface.

3.2 The AvTrakIF

Again modeled after the AcousticModemIF, the AvTrakIF has similar or identical calls to the AvTrak Server. No changes were made to the two data interfaces, the command type and the surface-aided-navigation structure (SAN record).

3.2.1 SCL_SAN_RECORD

This structure contains data intended for ingestion by the position estimation loops of the Kearfott SeaDeViL. The following code snippet shows the fields of this structure. Observe these are the same data contained in the NMEA-style SAN message, intended to be generated by AUVmon. This is identical to the SAN data structure in the AcousticModemIF.

```
struct SCL_SAN_RECORD {
        // Data from WinFrog
        float wfTime;    // Timestamp
        float wfLat;     // Latitude in Decimal Degrees
        float wfLong;    // Longitude in Decimal Degrees
        float wfSlant;   // Slant range
};
```

3.2.2 SCL_CMD_TYPE

The command type is used as a mechanism for transporting simple mission critical instructions from AUV operators shipboard to processes employing the AvTrakIF onboard the MVC. The following code snippet explains the individual commands simply.

```
enum SCL_CMD_TYPE {
        REQ_STAT,      //surface request for status info from the vehicle.
        REQ_SAN_ON,           //no current function
        REQ_SAN_OFF,    //no current function
        ABORT_MISSION,  //instruction to excecute abort plan
        START_SURVEY,   //instruction used in start survey behavior.
        INVALID_CMD
};
```

3.2.3 Methods

The following section are brief descriptions of the methods presented by this DeviceIF. These functions are where departure from the original AcousticModemIF occurs. Common to many of these functions are address inputs. All transceivers in the Sonardyne acoustic communication regime require an address. There are no defaults for the vessels, and must be configured via an ADM command to the PAN emulation port of the Ranger software[1].

What many of these address-input ready methods provide are the ability for the AvTrak to operate communications with other compatible beacons. Although not within the scope of current operations as described in Section 2.1, these functions are available for the AvTrak 6. By investing the effort to extend these abilities to the auv software base, we can envision use for inter-vehicle communication for use in autonomy and cooperative data acquisition. Also, the ability to range a potential set of bottom mounted sites could help in improving navigational data.

3.2.3.1 sendStatus()

Prototype:

```
DeviceIF::Status sendStatus(short address);
```

Description:

This function instructs the driver to assemble a NMEA-style STA message (see Section 2.3.2.2) and transmit it to a transceiver with given input *address*. The data are assembled with the driver through use of interfaces: *LayeredControlIF, NavigationIF,* and *FanIF.*

3.2.3.2 getSANFix()

Prototype:

```
DeviceIF::Status getSANFix(AvTrakIF::SCL_SAN_RECORD *sanFix, Boolean *newData);
```

Description:

This function returns a pointer to the latest SAN structure successfully parsed by the driver, and the *newData* flag indicates whether the data has been accessed before. Use of this function changes the flag to false until a new fix is parsed and put into memory.

[1] Refer to the application note, *SMS Telemetry Operation with Ranger USBL and AvTrack 2*, for more detail.

3.2.3.3 getSCLCmd()

Prototype:

```
DeviceIF::Status getSCLCmd(AvTrakIF::SCL_CMD_TYPE *cmd, Boolean *newCmd);
```

Description:

This function returns a pointer to the latest command message as successfully parsed by the driver as a *SCL_CMD_TYPE*, and the *newCmd* flag indicates whether the data has been accessed before. These commands are received as NMEA-style messages as described in Section 2.3.2.3. Use of this function changes the flag to false until a new fix is parsed and put into memory.

3.2.3.4 sendSMS()

Prototype:

```
DeviceIF::Status sendSMS(short address, AvTrakIF::AvTrakString msg, short nbytes);
```

Description:

Sends an instruction to the driver to transmit an SMS message (*msg*) with length (*nbytes*), with standard settings, to address (*address*). This message must contain ascii printable characters as described in the the constraints, Section 2.2.1.

3.2.3.5 sendSCLCmd()

Prototype:

```
DeviceIF::Status sendSCLCmd(short address, AvTrakIF::SCL_CMD_TYPE cmd);
```

Description:

Command the driver to issue a command (*cmd*) to address (*address*). This is to support future inter-vehicle communications.

3.2.3.6 measureRange()

Prototype:

```
DeviceIF::Status measureRange(short address);
```

Description:

Obtain range to a beacon with given address (*address)*. Range is returned in microseconds including TAT. This is not fully implemented yet. Result is only handled by deposit into the syslog.

3.2.3.7 requestStatus()

Prototype:

```
DeviceIF::Status requestStatus(short address);
```

Description:

Request all status information from a beacon at given address (*address*). Currently this information is simply pushed into the Syslog.

3.2.3.8 requestLocalStatus()

Prototype:

```
DeviceIF::Status requestStatus(short address);
```

Description:

Request all status information from the AvTrak 6 beacon. Currently this information is simply pushed into the Syslog.

3.3 The sonardyneCommand Class

In order to make compliant message encoding and decoding work in a cross platform compliant way, the *sonardyneCommand* class was constructed. Within the auv codebase, this class is located in the utilities (*util)* directory. This class was written in a fashion similar to the smac library, in that it is a series of static functions meant to modify given points in memory. By implementing these functions and data structures, it is possible to create messages appropriate for communication with any device using the Sonardyne Command Language[i]. This library also provides functionality to encode and decode the NMEA messages described in Section 2.3.2.

3.3.1 Message Support

Please refer to the Sonardyne Command language document for in-depth detail of the messages and options for each. Within this library structures have been established for each unique message type's interrogation and response fields. The structure *SCL_RECORD HEADER* is the only common set of fields for all interrogation and responses seen in the command language. The messages that are supported by this library are as follows in the attached table.

Messages	Command String	Encoding Supported	Decoding Supported
Firmware Version	FV	Yes	No
Bootloader Version	BV	Yes	No
Configuration Status	CS	Yes	No
Fixed Status	FS	Yes	No
Volatile Status	VS	Yes	No
Measure Range	MR	Yes	No
Measure Slant Range	MSR	No	No
Sonardyne Messaging	SMS	Yes	Yes

As of this writing, very few of these messages are actually parsed. All raw data received, however, is entered into the Syslog on the MVC implementation of this library. The structures for storing the data exist, just the functions to decode them have not been completed.

3.3.2 NMEA Message Interactions

Most of interest to this document is the functionality of the library to construct and decode NMEA messages passed through the Sonardyne Message Service. To properly construct a NMEA message using the library follow these steps:

1. Create and populate the SCL_NMEA_HEADER_RECORD and the relevant message content, whether SAN, STA, or CMD. Each have their own RECORD typed structure.
2. Use the `encodeNMEAMSG()` function to write the NMEA message string into memory. At this point the checksum is calculated and appended as well.
3. Create and populate the SCL_RECORD_HEADER with cmdtype set to CMD_SMS and set the appropriate address in the field.
4. Use the encodePacket() function to copy your NMEA string and form the message to be transmitted to the Sonardyne electronics.

Follow this same process in reverse using the decode functions available. Please see the Appendix for a copy of the header file.

4 Remaining Tasks

At this stage of development, the sub-sea software is complete to a level ready for final integration that nearly replicates the functionality of the Benthos acoustic modem. With a little more effort some other tasks could extend the system farther. The following list addresses these items.

- Modify AUVmon to use the sonardyneCommand library.
- Modify all current clients of the AcousticModemIF to instead use the AvTrakIF. (which are the Kearfott driver, Navigation and select behaviors contained in LayeredControl.)
- Finish message parsing for standard Sonardyne commands such as measureRange and other status messages.
- Increase the functionality of the driver to future tasks such as inter-vehicle communication.

5 Appendix I: sonardyneCommand.h

The following is a copy of the sonardyneCommand library definition.

```
/*
 * sonardyneCommand.h
 *
 *  Created on: Jan 20, 2015
 *      Author: emartin
 */

#ifndef SONARDYNECOMMAND_H_
#define SONARDYNECOMMAND_H_

#include <stdio.h>
#include <string.h>
/*
 * CLASS
 * sonardyneCommand
 *
 * DESCRIPTION
 * Sonardyne command string class for General SMS (sonardyne messaging service)
 * message construction and message translation to/from SMAC messages (seafloor
 * mapping acoustic communications), neededfor development of the sonardyne
 * avtrak drivers. These response and interrogation structures do not
 * support multi-element messages (CS:ME1)
 *
 * Based on Sonardyne command language Version 1.04.05, see document
 * CommandLaguage_V1_04_05a.doc for further details.
 *
 */
#define SCL_STRINGMAX 256
#define SCL_SMSMAX    128

typedef struct t_SCL_NMEA_HEADER_RECORD {
          unsigned int talkerID; //SF,M1,M2
          unsigned int msgType; // SAN, STA, or CMD
          unsigned int version; //protocol version
          unsigned long int ts_sec; //seconds from epoch
          unsigned long int ts_msec;//millsecond fraction
          unsigned int frontLen; // length in characters of the header portion of the record.
}SCL_NMEA_HEADER_RECORD;

typedef struct t_SCL_SAN_RECORD {
          float wfTime; //timestamp
          float wfLat;  //latitude
          float wfLong; //longitude
          float wfSlant;//slantrange
}SCL_SAN_RECORD;

typedef struct t_SCL_STATUS_RECORD {
          // Data from AUV
            float auvLat;
            float auvLong;
            float auvHeading;
            float auvDepth;
            float auvAlt;
            float auvVolt;
            float auvHumid;
            unsigned int auvBehavior;
            unsigned int auvAbort;
}SCL_STATUS_RECORD;

typedef struct t_SCL_CMD_RECORD {
          unsigned int cmd; // see MsgCmdType enum for values.
}SCL_CMD_RECORD;



typedef struct t_SCL_RECORD_HEADER {
          unsigned int cmdtype; //command type will set two letter command field
          unsigned int address; //address to interrogate;
}SCL_RECORD_HEADER;

typedef struct t_SCL_FIRMWARE_RESPONSE {
```

```
        char firmwareVersionString[SCL_STRINGMAX];
        int firmwareVersionStringLength;
        char protocolVersionString[SCL_STRINGMAX];
        int protocolVersionStringLength;
}SCL_FIRMWARE_RESPONSE;

typedef struct t_SCL_BOOTLOADER_RESPONSE {
        char bootloaderVersionString[SCL_STRINGMAX];
        int bootloaderVersionStringLength;
}SCL_BOOTLOADER_RESPONSE;

typedef struct t_SCL_MEASURERANGE_RESPONSE {
        unsigned long int rangeTime; // time of flight in microseconds including TAT
}SCL_MEASURERANGE_RESPONSE;

typedef struct t_SCL_MEASURESLANTRANGE_INT {
        char interrogationSignalString[SCL_STRINGMAX];
        int  interrogationSignalStringLength;
        int  wakeupTone;
        char channel1Descriptor[SCL_STRINGMAX];
        int  channel1DescriptorLength;
        char channel2Descriptor[SCL_STRINGMAX];
        int  channel2DescriptorLength;
        char channel3Descriptor[SCL_STRINGMAX];
        int  channel3DescriptorLength;
        char channel4Descriptor[SCL_STRINGMAX];
        int  channel4DescriptorLength;

}SCL_MEASURESLANTRANGE_INT;

typedef struct t_SCL_MEASURESLANTRANGE_RESPONSE {
        char channel1Descriptor[SCL_STRINGMAX];
        int  channel1DescriptorLength;
        unsigned long int Channel1RangeTime;
        char channel2Descriptor[SCL_STRINGMAX];
        int  channel2DescriptorLength;
        unsigned long int Channel2RangeTime;
        char channel3Descriptor[SCL_STRINGMAX];
        int  channel3DescriptorLength;
        unsigned long int Channel3RangeTime;
        char channel4Descriptor[SCL_STRINGMAX];
        int  channel4DescriptorLength;
        unsigned long int Channel4RangeTime;
} SCL_MEASURESLANTRANGE_RESPONSE;

typedef struct t_SCL_SMS_INT {
        int wakeupTone; //specifies whether to proceed with a wakeuptone 0 is no tone
        int portNumber; // sets the port number (0-7) default is 0
        int commandNumber; // sets the command number
        int subCode; // 0=SMS_STD EXCHANGE, 1=SMS_ACK_ONLY, 3=NO_RESPONSE
        int rangeData; // 0=no range data on reply, 1=range data in reply
        int bufferMsg; // 0= send immediately, 1=buffer until wd timeout
        char sms[128]; // actual sms message to send
        int smsLength; // length of sms message;
} SCL_SMS_INT;

typedef struct t_SCL_SMS_RESPONSE {
        unsigned long int rangeTime; //range time in microseconds with TAT included.
        int smsReplyCode; //reply or failure code.
        long int smsRangeuS; //range in uSeconds/
        char sms[128]; //optional reply message;
        int smslen; //length of sms string
}SCL_SMS_RESPONSE;


//typedef struct t_SCL_FIXEDSTATUS_RESPONSE {
//      unsigned int uid,
//      unsigned int functionalityLevel,
//      char[SCL_STRINGMAX] firmwareVersionString,
//      char[SCL_STRINGMAX] protocolVersionString,
//      char[SCL_STRINGMAX] transducerInfoString,
//      char[SCL_STRINGMAX] sensorsInfoString,
//      MORE FIELDS TODO LATER
//}SCL_FIXEDSTATUS_PARAMS;


class sonardyneCommand {
```

```
public:
        enum CmdType {
                CMD_VERSION = 0, // Firmware and protocol version
                CMD_BOOTLOADER, // Bootloader Version
                CMD_FIXEDSTATUS, // fixed status request
                CMD_VOLATILESTATUS, // volatile status request
                CMD_CONFIGSTATUS, // configuration status request
                CMD_MEASURERANGE, // measure range to target
                CMD_MEASURESLANTRANGE, // measure slant range to target
                CMD_SMS, // sonardyne messaging service message
                CMD_UNKNOWN
        };

        enum smsReplyCode {
                SMS_UNKNOWN = 0,
                SMS_ACK,
                SMS_NO_REPLY,
                SMS_FAILED,
                SMS_OK,
                SMS_SMS,
        };

        enum MsgType {
                MSG_UNKNOWN = 0, //unknown message type
                MSG_SAN,//surface aided navigation
                MSG_STA, //status message
                MSG_CMD, //command message

        };

        enum TalkerID{
                TID_UNKNOWN = 0, //unknown source no more than 14 allowed (single char limit)
                TID_SURFACE,     //source surface
                TID_MAPPER1,       //mapper1 source
                TID_MAPPER2            //mapper2 source
        };

        enum MsgCmdType {
                REQ_STAT = 0,      // Surface request for status from vehicle
                REQ_SAN_ON,    // Vehicle requests surface to begin sending SAN messages
                REQ_SAN_OFF,   // Vehicle requests surface to stop sending SAN messages
                ABORT_MISSION, // Surface commanding vehicle to abort mission
                START_SURVEY,   // Surface commanding vehicle to start survey
       INVALID_CMD
        };

        enum BhavrType {
                ABORT_BHAVR = 0,
                DIVE_BHAVR,
                SURVEY_BHAVR,
                ASCEND_BHAVR,
                INVALID_BHAVR // Should always be the last one
        };

        sonardyneCommand();

        //void encodePacket(char * msg, short numBytes); //decoder constructor
        static void encodePacket(char * packet, int * packetsize, int cmdType); //encoder constructor for commands
without input
        static int encodePacket(char * packet, int * packetsize,
                        SCL_RECORD_HEADER * hdr); // encoder class constructor for commands with address input only
        static void encodePacket(char * packet, int * packetsize,
                        SCL_RECORD_HEADER *hdr, SCL_SMS_INT * sms); // encoder constructor for an sms message.
        static void encodePacket(char * packet, int * packetsize,
                        SCL_RECORD_HEADER *hdr, SCL_MEASURESLANTRANGE_INT * msr); // encoder constructor for measure
of slant range.

        // encode the message string for sms inclusion
        static void encodeNMEAMsg(char * msg, int * msgsize, SCL_NMEA_HEADER_RECORD * hdr, SCL_SAN_RECORD * san);
        static void encodeNMEAMsg(char * msg, int * msgsize, SCL_NMEA_HEADER_RECORD * hdr, SCL_STATUS_RECORD * status);
        static void encodeNMEAMsg(char * msg, int * msgsize, SCL_NMEA_HEADER_RECORD * hdr, SCL_CMD_RECORD * cmd);

    static void decodeHeader(char *packet, int *packetsize, SCL_RECORD_HEADER * header);

    static void decodeSMS(char *packet, int * packetsize, SCL_SMS_RESPONSE * smsResponse);
    static void decodeMeasureRange(char *packet, int * packetsize, SCL_MEASURERANGE_RESPONSE * mrResponse);
    static void decodeMeasureSlantRange(char *packet, int * packetsize, SCL_MEASURESLANTRANGE_RESPONSE * msrResponse);
```

```
    //decoder for sms nmea messages
    // encode the message string for sms inclusion
    static void decodeNMEAMsgHeader(char * msg, int * msgsize, SCL_NMEA_HEADER_RECORD * hdr);
    static void decodeNMEAMsgContent(char * msg, int * msgsize, SCL_SAN_RECORD * san);
         static void decodeNMEAMsgContent(char * msg, int * msgsize, SCL_STATUS_RECORD * status);
         static void decodeNMEAMsgContent(char * msg, int * msgsize, SCL_CMD_RECORD * cmd);
         static void validateNMEAchecksum(char * msg, int * msgsize, unsigned int * valid); //1 valid 0 invalid

         virtual ~sonardyneCommand();
private:
         static void encodeHeader(char * packet, int *packetsize, SCL_RECORD_HEADER * hdr);
         static void encodeNMEAMsgHeader(char *msg , int *msgsize, SCL_NMEA_HEADER_RECORD *hdr);
         static void generateChecksum(char *msg, int *msgsize, unsigned int *checksum);

};

#endif /* SONARDYNECOMMAND_H_ */
```