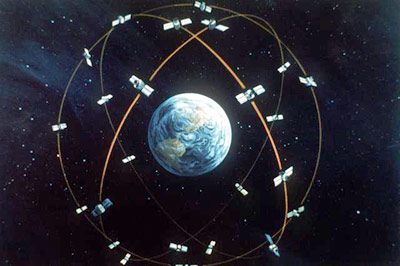
**Senior Design**

**CEC 470**

**Mini Project 1**



Team Leader: Elliot Robinson

Software Implementation Lead: James Gregoire

Testing Lead: Bryce Dodds

Software and Testing Collaborator: Wenzhi Zhao

Documentation Lead: Carole Fountain

Faculty Advisors

Dr. Richard Stansbury

Dr. Salamah Salamah

Dr. William Barrot

**Abstract**

The first project for the senior design class is a software implementation of GPS Parsing. Given a file containing GPS NMEA 0183 strings, the program is to read the file and extract the required data, which includes:

* Date
* UTC Time
* Time since Unix Epoch, TAI
* Latitude
* Longitude
* Altitude
* Number of Satellites Tracked, and for each satellite:
  + PRN
  + Elevation
  + SNR
  + Azimuth

Once the required data is extracted from the input file, it is to be stored in a data structure. The content of the data structure must be written to an output file in ASCII, comma delimited.

**Introduction**

NMEA 0183 is a data specification for communication between marine electronic devices which includes GPS. It is defined and controlled by the National Marine Electronics Association (NMEA). The NMEA 0183 standard uses a simple ASCII, serial communications protocol that defines how data is transmitted in a “sentence” from one “talker” (in our case one satellite) at a time to multiple listeners (GPS receivers). The standard also defines the contents of each sentence type so all messages can be parsed accurately.

The application layer protocol rules for a NMEA sentence include: each sentence starts with a dollar sign ($) character; the next five characters identify the talker (GPS, GLONASS, etc) and the type of message (Fix information, Lat/Lon data, Overall Satellite data, Recommended Navigation Data, Date and Time, etc). This first information field is ended by a comma – each following field is separated by a comma. The end of the sentence is usually denoted by an asterisk (\*) which is then followed by a checksum for error checking purposes.

One of the larger goals of this project is to design software that is portable and reusable – the next project will involve hardware, and the final mini-project will combine the two.

**Design and Implementation**

The following describes in a non-formal manner the design details of the various elements of the “Functional Requirements” section of the SRS.

General Requirements:

1. The system shall be invoked from the command line with two parameters: the input file and output file.
2. The system shall truncate the output file.
3. The system shall read one line from the input file.
4. The system shall ensure the line is a valid NMEA sentence.
5. The system shall validate the checksum of the NMEA sentence.
6. The system shall pack the validated NMEA sentence into a NMEA Message.
7. The system shall parse the validated message, updating a data structure if the information is newer than the current data structure time.
8. If the data structure is updated, the system shall append it to the output file.
9. The system shall repeat step 3 through 7 until reaching the end of the input file.

High-Level Design Considerations

* This project must be programmed in the C programming language using no third-party libraries.
* This project shall be designed in a modular manner to facilitate future modifications.

Custom Data Types

**struct Date** { // Contains a month, day, year date  
short month;  
short day;  
short year;  
};

**struct Satellite** { // Contains data pertaining to a single satellite  
short prn;  
short elevation; // Degrees  
short snr;  
short azimuth;  
};

**struct NMEAData** { // Keeps track of pertinent data gathered from NMEA sentences  
Date date;  
short utcTime; // Yes, underscores would be better here…  
time\_t epochTime;  
time\_t taiTime;  
float lat; // North lat == positive, South lat == negative.  
float lon; // Same as above for East and West  
char dmsLat[16]; // (dddommm’sss.ss”\0)  
char dmsLon[16]; // Same as above.  
float altitude;  
short numSatellites;  
Satellite satellites[12] // A maximum of 12 satellites can be considered in view in NMEA. I see no reason to store more than what the system can describe.  
short isDelta; // 0 if the structure has not been changed, 1 If the structure has changed and needs to be written to the output file.  
};

**struct NMEAMessage** {  
char type[6];  
char data[73];  
};

Functions

**Signature: int main (int argc, char \*argv[])**Description: Main entry point. Contains the file-processing loop.  
Pre-Conditions: None.  
Post-Conditions: Supplied output file has been created and filled with all valid NMEAData created.  
Return: Returns 0 on success, non-zero on failure.

**Signature: int verifySentence (char \*sentence)**Description: Verifies that the supplied string is a valid NMEA string which is prefixed with $, ends with \*<checksum>, and does not contain any additional $ characters.  
Pre-Conditions: None  
Post-Conditions: None  
Return: Returns 0 on success, 1 on failure;

**Signature: int validateChecksum (char \*sentence)**Description: Calculates the checksum for the sentence and checks it with the provided checksum.  
Pre-Conditions: Sentence has been verified.  
Post-Conditions: None.  
Return: Returns 0 on success, 1 on failure.

**Signature: int parse (struct NMEAData \*dataStore, struct NMEAMessage \*message)**Description: Calls the appropriate parse function for the sentence based on the message type.  
Pre-Conditions: An instance of NMEAData has been instantiated. Sentence has been verified and its checksum validated.  
Post-Conditions: dataStore has been updated to contain pertinent data from sentence as necessary. If dataStore was updated, dataStore->isDelta will equal 1.  
Return: Returns 0 on success, 1 on failure, or -1 if given an unknown message type.

**Signature: int parseGGAMessage (struct NMEAData \*dataStore, struct NMEAMessage** \*message)  
Description: Verifies the structure of and parses a GGA message.  
Pre-Conditions: An instance of NMEAData has been instantiated. Sentence has been verified and its checksum validated.  
Post-Conditions: dataStore has been updated to contain pertinent data from sentence as necessary. If update occurred, dataStore->isDelta shall be set to 1.  
Return: Returns 0 on success, 1 on failure, or -1 if the message time is older than the NMEAData time.

Additional functions to parse messages shall be created for every message to be parsed. Messages without a time field will never return -1. The complete list of messages to be parsed and what they provide is as follows:

**GGA**: UTC Time, Epoch time (derived), TAI (derived), Lat, Lon, DMS Lat (derived), DMS Lon (derived), Altitude (feet MSL, conversion), Satellites being tracked

**GLL**: UTC Time, Epoch time (derived), TAI (derived), Lat, Lon, DMS Lat (derived), DMS Lon (derived)

**GSV**: Satellites being tracked, PRN, Elevation, SNR, and Azimuth for each satellite.

**RMC**: Date, UTC Time, Epoch time (derived), TAI (derived), Lat, Lon, DMS Lat (derived), DMS Lon (derived)

**ZDA**: Date, UTC Time, Epoch time (derived), TAI (derived)

Additional struct NMEAData Information

Provided Data:

Date  
UTC Time  
Lat (degrees, floating point)  
Lon (degrees, floating point)  
Altitude (feet, possible conversion from given meters necessary)  
Number of tracked satellites  
For each satellite:  
PRN  
Elevation  
SNR  
Azimuth

Derived Data:

Epoch time (derived from UTC time)  
TAI time (derived from Epoch time)  
Lat (DMS in the format “dddommm’sss.ss””, derived from floating point lat)  
Lon (DMS in the format “dddoommm’sss.ss””, derived from floating point lon)

**Description of Test Plan, Test Execution, Test Results**

Test Plan

**Function name**: **int main()**Test Description: Provide valid and invalid test files and check for desired output  
Success Case: Given correct input file the system shall return correct output. Given file with incorrect data only valid information from that file shall be returned.

**Function name: int verifySentence (char \*sentence)**Test Description: Provide valid and invalid NMEA strings and ensure only valid NMEA strings are passed  
Success Case: Given correct input NMEA string the system shall return 0. Given invalid input string the system shall return 1.

**Function name: int validateChecksum (char \*sentence)**Test Description: Provide valid and invalid NMEA strings and ensure only valid NMEA strings are passed  
Success Case: Given correct input NMEA string the system shall return 0. Given invalid input string the system shall return 1.

**Function name: int parse (struct NMEAData \*dataStore, char \*sentence)**Test Description: Calls the appropriate parse function for the sentence based on the message type.  
Success Case: Returns a 0 for success, 1 for failure, or -1 if an unknown message type is found.

**Function name: int parseXXXMessage (struct NMEAData \*dataStore, char \*sentence)**Test Description: Verifies the structure of the message and parses it based on the message type (GGA, GLL, GSV, RMC, ZDA).  
Success Case: Returns a 0 for success, 1 for failure, or -1 if the message time is older than the NMEAData time.

Conclusion

References

www.gpsinformation.org/dale/nmea.htm

en.wikipedia.org/wiki/NMEA\_0183

List of Test Cases and Execution Log