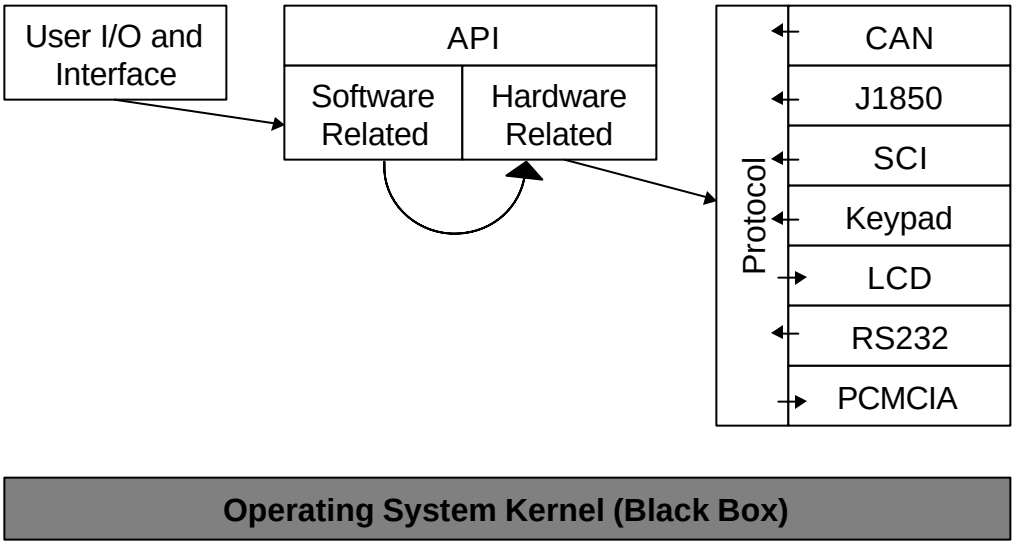
Function Flow Block Diagram

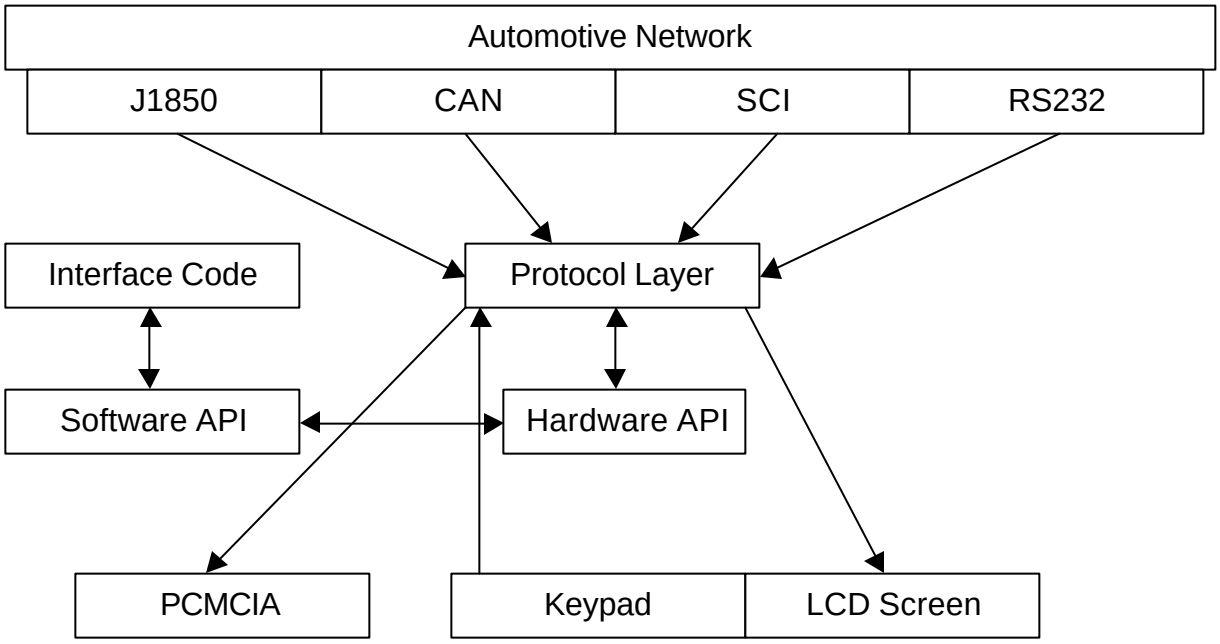
All the functions in all the modules operate atop the framework provided by the API, and therefore make calls to and are called by the operating system kernel in whatever fashion the kernel needs.



The user interface code, which contains the implementation of the menu system required to operate and configure the device and display the log of messages, directly uses the software API. Additional high-level capabilities, data analysis screens and menus can be added with minimal effort and use the facilities provided by the software API. The hardware API is a portal to the hardware-specific code contained in the protocol module. The protocol level contains the software routines that execute the actual data transfer to and from the hardware surrounding the micro-controller, facilitating the collection of network data and transfer of the interface display to the LCD hardware.

Data Flow Model

The data flow for this product will be primarily from the automobile network to the LCD screen for user interaction. There will also be data flowing from the keypad to implement the user interface, and from the device out through the PCMCIA port for data log dumps to an external computer for further analysis.



At the top are the automotive networks, which is the primary source of raw data this device will utilize. The code in the Protocol Layer will provide the interface to these hardware subsystems. Additional hardware-specific code will exist in the Protocol Layer for data flow to the LCD and from the keypad. The Hardware API provides a simple, consistent interface to the code in the Protocol Layer. The Software API provides higher-level tools such as a menuing facility and storage for data messages that are pulled from the automotive networks.

Additional information about the contents of each of these modules can be found in the **Functional Outline** section of this document.

Data Format

Menus

The Software API contains the menuing facility, while the actual menus are specified in the Interface code. Each menu will be stored in a data structure containing:

- Title
- Number of menu items
- Current item
- Each item (test strings)
- Which menu to jump to or which routine to run for each item in the menu

This format is used by the menuing engine to present a logical, usable interface to the user. There also exists a stack of menu IDs, allowing a heirarchy and a history to be maintained, allowing the user to easily return to the previous menu at any time.

Message Log Storage

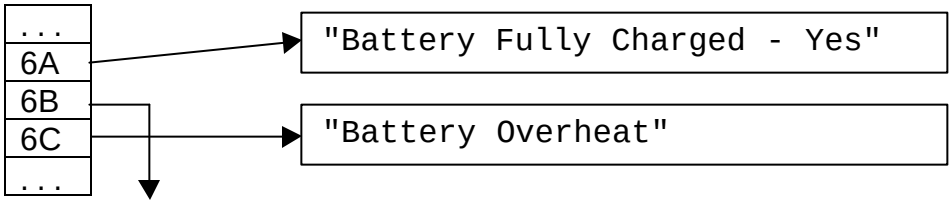
The messages retrieved from the automotive networks are of varying length, with the length of each message being given as the first byte in the content of the message. The messages will be stored in the same format, in a continuous stream occupying whatever space is available in the device's memory space after the code has been accounted for.

Header	Len	. . . . .	Footer	Header	Len	. . . . .	Footer	. . . . .

Using this method, traversing the list is a relatively simple task, and displaying the log of message data will be straightforward.

Message Lookup Table

Each message retrieved from the automotive networks contains an 8-bit identifier specifying the meaning of the message. There will be a 256-element list of string pointers, one for every possible message. The identifying byte in the message will be used as an index into this array allowing for fast conversion of a hexadecimal ID to an English test message.



Functional Outline

Interface.h / Interface.c

This file pair contains the code and data defining the user environment on the system. The specific content and hierarchical structure of the menus, the display of the user log, and configuration selection routines will be encoded in this module.

API.h / API.c

This file pair contains the libraries of routines that enable easy interface development and provide gateways to hardware-specific code segments. There are 2 parts to the API – the software-specific routines and the hardware-specific routines.

Software-based Routines

These routines provide capability to the interface programmer.

char *MessageLookup(U16)
Given an unsigned, 16-bit value, this routine returns a pointer to the meaning of that network message. Further decoding may be necessary to determine the meaning of the parameters provided with that message.

void SetLogFilter(U8)
This routine sets a flag selecting which types of messages will be displayed in the log. The flags are identified with #define'd 8-bit values.

void ClearLogFilter(U8)
This routine clears a flag, selecting which types of messages will be omitted from the log display. The flags are identified with #define'd 8-bit values.

void BeginRecordingLog(void)
This routine causes the software to begin saving network messages that come across the system. The log is *not* cleared before recording begins.

void StopRecordingLog(void)
This routine causes the software to begin discarding network messages that come across the system. The saving is suspended – the log is *not* cleared.

void ClearLog(void)
This routine clears the log buffer and initializes the position within the log.

void ResetCurrent(void)
This routine returns the cursor to the beginning of the stored messages log.

void FindNextMesage(U16 MessageType)
This routine searches forward from the current position for the next message in the log matching the criteria specified by the #define'd 16-bit value MessageType.

U8 GetLogPosition(void)
This routine returns the current position of the user in the log display.

void SetLogPosition(U8)
This routine sets the current position of the user in the log display.

void SetConfigurationFlag(U8)
This routine sets a configuration flag, #define'd like the filter flags discussed above.

void ClearConfigurationFlag(U8)
This routine clears a configuration flag, #define'd like the filter flags discussed above.

void DrawString(char *Str, U8 n, U8 R, U8 C)
This routine copies a string to the screen, starting at row R and column C. The copy continues for n characters or until the string will run off the right edge of the screen, whichever is smaller.

Void DrawMenu(void)
This routine draws the menu specified by the system parameters to the LCD. The menu to be displayed is determined by the global variables specifying current menu selection, menu depth, and current active menu.

void MoveCursor(char KeyPress)
This routine will take the appropriate action based on the current screen (menu, log display, etc.) and the key pressed by the user.

Hardware-based Routines

These routines provide a common interface to lower-level, hardware-specific code. If the hardware is changed, but capabilities are not increased or decreased, this code will not need to be changed. However, if data sources or sinks are added or removed from the hardware platform, these data ports must be added or deleted from this library.

void CopyFromBuffer(void *Destination, U8 WhichBuffer, U16 n)
This routine transfers n bytes from the buffer specified by WhichBuffer, a #define'd 8-bit constant, into the memory starting at the location specified by Destination.

void CopyToBuffer(U8 WhichBuffer, void *Source, U16 n)
This routine transfers n bytes starting at the memory location specified by Source into the buffer specified by WhichBuffer, a #define'd 8-bit constant.

Protocol.h / Protocol.c

This file contains the hardware specific code for data transfer between the microcontroller and the ancillary hardware. If the hardware environment changes, only this file must be modified (with the exception discusses in the *Hardware-based* Routines section above) to update the software to talk to the new hardware environment. These routines are called by the CopyFromBuffer(…) and CopyToBuffer(…) routines in the API file pair.

SendToPCMCIA(void *Source, U16 n)
This routine transfers data from the memory location specified by Source to the PCMCIA port on the device.

GetFromKeypad(void *Destination)
This routine gets the most recent key pressed from the keyboard buffer and stores it in the memory location specified by Destination.

SendCharToLCD(char theChar, U8 R, U8 C)
This routine transfers a single character (at the specified row and column) to the LCD hardware via whatever interface the LCD controller requires.

GetFromCAN(void *Destination)
This routine retrieves the most recent network packet available on the CAN bus and stores it in the memory location specified by Destination.

GetFromSCI(void *Destination)
This routine retrieves the most recent network packet available on the SCI bus and stores it in the memory location specified by Destination.

GetFromJ1850(void *Destination)
This routine retrieves the most recent network packet available on the J1850 bus and stores it in the memory location specified by Destination.

GetFromRS232(void *Destination)
This routine retrieves the most recent message from the RS232 buffer and places it in the memory location specified by Destination.

Kernel.h / Kernel.c

These files contain the core operating system for the device. These files will be obtained from an outside source and will function as “black boxes” from our point of view. However, routines for basic event processing are assumed to be present.

Test Plan

We will test our design in a bottom-up fashion, ensuring the correctness and stability of each level before relying on it for use in higher-level code. In this sense, we will test each module as a white box first, ensuring proper internal operation. After white box testing is complete, that module will be exercised as a black box while testing higher-level code as a white box. Since we have various levels of development, we will employ this strategy at each level to ensure correctness and completeness.

First, we will run our software in our functional simulation environment, using Microsoft Visual C++ v5.0 on the Windows platform. We will check the functional behavior, logic, and conceptual integrity of all our software modules in this environment in the manner described above.

1. Test simple input/output formatting routines
2. Test compound output screen formatting routines
3. Test menu drawing/operation routines
4. Test configuration/filtering routines
5. Test log storing mechanism
6. Test data display forms/routines

After the software has been verified on the Windows simulator, it will be cross-compiled and the preceding steps will be repeated on the hardware emulator, again verifying correctness and completeness, and to ensure that the completed code will fit and run within the memory space that will be available on the final hardware. We will run abbreviated module tests within the 2Kb file size constraint until we have a license which will allow us to compile larger files. At that time, we will be able to compile the entire software project and complete the instruction-level testing.

After the hardware emulator phase has been completed, the software will be ready for cross-compiling and uploading to the physical hardware for final testing and delivery.

Coding Standards

File Structure

All software will be developed using .h/.c file pairs. The .h file (header file) will contain the interface or declaration for a code module, while the .c file (content file) will contain the implementation of that module.

Comments

Comments will be included in the source files before every procedure, routine, and function that describe the parameters that particular code segment takes, what its behavior is, and what it returns. Within each code segment, comments will be included every few statements, where needed, to clearly explain the process being executed by the code.

Long variable names will be used, to assist in readability and to help self-document the code. Each word in the variable name will be capitalized, including the first letter. The first letter of the variable name will no be capitalized if the name starts with “the” or “a”. Also, each variable declaration will be commented explaining the purpose and use of the variable.

Indentation

Each code sub-block will be indented exactly 3 spaces from the parent block to enhance readability on hard copy. Braces ({ and }) will generally appear on lines by themselves and indented with the code they enclose. Braces may be included on a line with code if there is only one statement within the braces. In this case, the opening brace, the statement and the closing brace should all appear on the same line.

#define Statements

#define statements will be used within the code to avoid arbitrarily hard-coding constants into the code. All #define'd IDs will use all capital letters to avoid confusion with variable names. #define statements such as those used to determine the hierarchy in the menu system should be dependant upon themselves to resize and reevaluate their values such that inserting a #define’d ID will not require reentry of all subsequent IDs.

Development Environments

This software is being developed for the eventual placement on a dedicated system custom tailored to suit the customer’s needs. However, several levels of development are being undertaken:

The first level, the interface simulation, is to be developed in Microsoft Visual C++ v5.0. At this level, the base functionality of the interface will be developed as proof-of-concept. This will include the virtual creation of the physical device, as represented by a dialog-based Windows program, the input from the device's "keyboard" and the output to the device's "LCD screen".

The second level, the functional development of the entire software, is also to be developed in Microsoft Visual C++ v5.0. At this level, the behavior of the actual embedded code will be developed and tested, though on a different platform. ANSI Standard C will be used for all embedded code so that cross-compiling for the embedded platform will hold as few surprises as possible.

The third level, the simulation of the specific micro-controller, will be executed within the Cosmic environment, emulating the Motorola HC12 micro-controller at the machine-code level. At this level, the code developed in the second level will be cross-compiled for the micro-controller and run on the simulated hardware to ensure correctness and completeness.

The fourth level will be downloading the cross-compiled code to and testing it on the actual hardware. After this point, the software development will be complete for the customer, then the documentation must be completed.