EVS24
Stavanger, Norway, May 13 - 16, 2009

An Embedded Computer Based System for Monitoring, Diagnostics and Communication in Hybrid and Electric Vehicles.

M. Paschero[1], G. Del Vescovo[1], A. Rizzi[1], F. M. Frattale Mascioli[1]
[1]*Sustainable Mobility Research Center, INFOCOM Department,*
Via Eudossiana 18, 00184, Roma, maurizio.paschero@pomos.it

Abstract

In the present paper a software system for onboard processing of automotive data is proposed and described. The project of a system composed by some modules interacting on the basis of a producer-consumer paradigm is discussed. The modules run as separate processes on an embedded system equipped with 266 MHz CPU, Linux operating system and several peripherals, including a TFT display. The producer side is represented by the processes which are in charge of receiving and decoding information coming from CAN or RS232 data links, forming a vector of physical quantities describing the status of the vehicle. The vector is placed in a shared memory segment, where the consumer processes can read and use it for their purposes. Consumer processes include a graphical display (virtual dashboard), a data logger recording data on a memory card, and a transmitter module sending information over wireless channels. In order to achieve the best flexibilty and reusability over a range of vehicle types, some of the modules are fully configurable by means of a specified configuration file placed on disk. Configuration files report the information used to decode the incoming data from the devices through the data links, as well as the directives on how to display data on screen. Partial implementation is described in detail. Future application to real vehicle prototypes is briefly discussed.

Keywords: data acquisition, diagnosis, electronic, instrumentation

1 Introduction

The increasing cost-effectiveness and reliability of embedded computing systems make them a suitable choice for flexible and organic information management in vehicles (1), (2), (3). An onboard computer equipped with a touchscreen LCD provides a highly customizable and updatable way of monitoring critical information coming from engines or battery packs, like speed, operating condition, state of charge. Moreover, the use of this device enables a simple data interchange between the vehicle and the outside world by means of wireless technologies (2).

Nowadays, in a modern automobile, and especially in hybrid and electric vehicles is possible to find up to fifty electronic control units (ECU) for various subsystems. Typically the biggest processor is the engine control unit, which is also referred to as ”ECU” in the context of automobiles. The most of those units need to communicate among them in order to control the performances and the emissions of the vehicle. To allow microcontrollers and devices to communicate with each other within a vehicle without a host computer a few communication protocol have been proposed. Among them it is remarkable to mention the serial communication bus and the ControllerArea Network (CAN) bus. The serial communication bus consists in transmitting one bit after each other. Among serial busses it should be mentioned the Recommended Standard 232 (RS232) proposed in 1969 by the Electronics Industries Association (EIA) which de-

fines the characteristics of the electrical signals such as voltage levels, signaling rate and other relevant quantities. A CAN bus network can include multiple nodes, and each node is able to send and receive messages, but two or more different nodes can not obtain the bus simultaneously. Each message is transmitted serially onto the bus, bit after bit. If the bus is free, any node may begin to transmit. Otherwise, if two or more nodes begin sending messages at the same time, the message with the more dominant ID obtain the bus according with the communication protocol rules and it is received by all nodes. The CAN protocol was officially released in 1986 at the Society of Automotive Engineers (SAE) but only in the most recent years it has been extensively implemented. In fact, nowadays it is very easy to find commercial components such as thermal engines, electric motors, inverters, storage systems etc. etc. provided with a CAN controller which allow the component to be interconnected to the CAN network of the vehicle. Nevertheless it is still possible to find components communicating through serial bus or very rarely on analog channels. Based on the previous arguments it seems reasonable to assume that each relevant quantity related to the instantaneous status of the vehicle can be measured and transmitted on a communication bus. The full understanding of this capability makes clear the opportunity to equip the vehicle with a computer based embedded system capable to store and elaborate this information. This opportunity becomes even more attractive for electric and hybrid vehicles where a large amount of information coming from different devices must be taken under consideration.
In the present paper, an embedded computer based system is proposed for a set of automotive applications, with special regard to hybrid and electric vehicles. More precisely in Sec. 2 will be shortly introduced the hardware platform used for the implementation. In Sec. 3 will be described the software architecture of the whole system. Special emphasis will be given to the strategies used to make the software reusable and scalable. In Sec. 4 will be described the modules of the software which have been already implemented. In Sec. 5 will be shortly described the projects for the development of two prototypes which are now under development in the laboratories of Polo per la Mobilità Sostenibile (POMOS) (4) which will mount a signal elaboration system based on the software described in this paper.

2 Hardware Platform

The hardware platform required to host the proposed system need to be compact, robust, light and capable to support different communication standards such us CAN, RS232, USB, Ethernet, GSM, GPRS, UMTS. Moreover it should be equipped with a solid state disk to allow hosting of the developed software and a removable solid state memory to allow post processing of monitoring stored data. Furthermore it should mount a LCD touch screen to allow the implementation of a digital dashboard. Finally the system should be able to work properly in a wide range of temperature and with a voltage level available in a standard vehicle. Based on the previous specifications, the hardware platform adopted for the development of the proposed system is the Engicam GPX21 (5), a board based on the Freescale i.MX21 32-bit multimedia processor, with several peripherals and connection facilities. The GPX21 board is shown in Fig. 1. The board

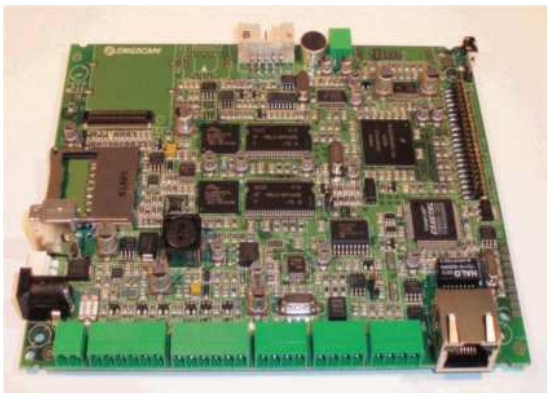

Figure 1: The GPX21 board

is connected to a Thin Film Transistor (TFT) 5.7” display with 640 x 480 resolution shown in Fig. 2. The salient system specifications are re-

Figure 2: The GPX21 LCD

ported in Table 1. The available computational power and the advanced operating system allow the handling of multiple processes communicating by a shared memory segment. The development can be done in standard C or C++ using the gcc arm-linux toolchain. Advanced graphical applications can be developed thanks to the presence of the nano-X graphic server installed on the system (6). Communication with the host PC can be established via the Secure SHell (SSH) protocol over the Ethernet link. This way, the GPX21 console can be accessed remotely, in order to transfer executable code and other files, perform configuration actions and view the console output of running programs for testing purposes.

Table 1: GPX21 specifications

Name	Value/Range
CPU model	ARM926
CPU frequency	266 MHz
Multimedia accelerator	Integrated with CPU
RAM	64 MB
Serial ports	RS232, RS485
Wireless connectivity	GSM, GPRS, UMTS
USB	2.0
Ethernet	10 Mbps
CAN bus	500 bps
Operating system	Linux 2.4
Flash disk	64 MB
Display	TFT 5.7”
VGA Resolution	640 x 480
Temperature range	-40, +85 C
Supply voltage	9, 30 V
Power consumption	5 W
Size	170 x 123 x 44 mm
Weight	400 gr

3 Software Architecture

The software architecture is designed with the aim of making the system modular and easy to be upgraded with new functions. This requirement can be fulfilled by defining an interprocess communication procedure which allows different independent processes to interact with a unique set of data. Of course the competition among processes to access the data must be synchronized through well defined strategies. A simple scheme of the software architecture is shown in Fig. 3. In the left side of the figure is shown the CAN bus of a generic vehicle, and other incoming automotive data by means of a serial communication link. Recently, the most of components (*i.e.* thermal engine, electric motor, energy storage system, GPX21, etc. etc.) are connected to the CAN bus through a CAN controller. Each component transmit on the CAN bus to all other components the information about its status following the CAN protocol rules. However, it is not uncommon to have automotive devices communicating through serial ports, following standards like the RS232.

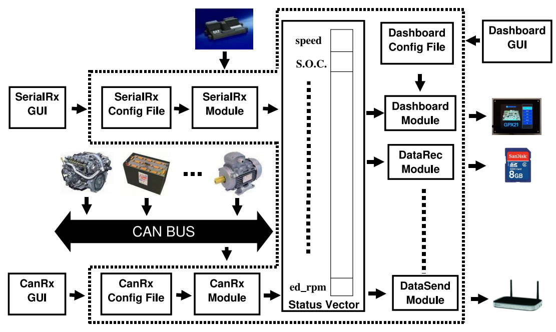

Figure 3: Scheme of the software architecture

On the right side of the figure are shown all the hardware peripherals which can be considered to take advantage of the output of the elaboration of the signals read from the communication busses. Among them can be mentioned an LCD screen to display data, an SD flash memory to store data and a Wi-Fi modem to transmit, but other devices can be taken under consideration. The central portion of the figure (enclosed in a dashed line) represents the software used to allow the communication between the hardware components mounted on the vehicle and the hardware peripherals used to make the information about the vehicle usable for the final user of the car. All the software modules enclosed in the dashed line are implemented as independent processes running in the embedded system CPU.

The software architecture can be described through a producer-consumer paradigm. More precisely the producer side is constituted by the modules labeled with ‘CanRx’ and ‘SerialRx’ in Fig. 3 whereas the consumer side is constituted by the modules labeled by ‘Dashboard’, ‘DataRec’, and ‘DataSend’ in the same figure. As previously remarked the software is designed to allow the addition of other modules on both the producer and the consumer side when a larger number of facilities is needed. The communication between the producer and the consumer side is realized by means of a shared memory approach. The shared memory can be interpreted at an high level as the status vector of the of the vehicle. It is constituted of n real numbers representing the instantaneous values of n quantities associated with the status of the vehicle. The status vector is formed using the following rule: each component of the vector reports the most recent known value of the physical quantity it refers to. This value is the one contained in the last received CAN (or serial) data message referring to that specific signal. This approach allows the consumer side to ignore the rules of the communication protocol used to transmit signals in the vehicle. In fact the producer side modules take care of formatting data read on the communication busses in a high level format making the physical structure of the vehicle transparent for the consumer side modules. The mentioned modules on the producer and the consumer side are designed to be usable in many types of vehicle and does not require, for example, the full *a priori* knowledge of the components connected to the CAN bus, so they do not need to be recompiled if the configuration of the CAN network is changed. The claimed generality can be guaranteed through the use of a few configuration files describing in a specified format the hardware components connected to the CAN bus of the actual implementation. This step, needed to guarantee the generality of the software, could be overpassed if the companies working in the automotive sector will find an agreement on the assignation of the CAN ID to specified quantities once for all, by defining a fully open standard. In this situation the server could inquire each node of the bus to obtain the complete knowledge of the CAN network, avoiding the requirement of a configuration file. The configuration file must include a table of the CAN ID and the formatting of the transmitted bytes for each components connected to the CAN bus. Moreover this configuration file should include high level information regarding the number and the physical nature of the real parameters needed to describe the status of each components. A similar configuration file is used for other types of data links, like serial communication ports. These configuration files, depending on the particular implementation, will allow the producer modules to generate the shared memory structure corresponding to the actual configuration of the CAN bus or serial communication and, similarly, will allow the consumer modules to read a given sub-portion of the status vector in order to perform the desired elaboration. Another configuration file is used for specifying the desired layout of the graphical dashboard. By this file is possible to select the number and type of graphic controls to use, and to connect them to the physical quantities (status vector components) of interest. In order to aid the editing of the configuration files, GUI applications are provided which can be run on a regular PC. These GUI application allow the user to insert the desired configuration entries with a friendly interface. They speed up the file editing process and protect against error making several checks on the inserted data.

4 Software implementation

The principle at the base of the proposed software architecture is the sharing of the status vector of the vehicle among different software modules. This scheme has been implemented by means of processes communicating via a shared memory segment. The modules implemented so far are the CanRx, the DataRec and the Dashboard processes (see Fig. 3). Moreover, to allow a simpler development of the Dashboard module, some higher level graphical functions built on top of the lower level nano-X API functions have been written. These high level functions make easier the drawing of signal-monitoring controls like gauges, led arrays, 7 segment digit displays. The adopted mechanism for the inter-process communication is based on shared memory. It has the advantage of allowing the best performance, thanks to its low-level data sharing paradigm which allows to avoid data copies and costly stream management. The main drawback of the shared memory approach is the access synchronization (simultaneous write and read operation have to be avoided to prevent race conditions). The access regulation has been realized by means of a simple semaphore, allowing just one process at once to access the memory segment. The memory segment, besides the state vector values, contains ancillary information about the state vector (name, range, measure unit of each component). The state vector instances are placed in a circular buffer. The use of a buffer is adopted to loosen the timing requirements for the access of the various modules. The buffer holds the last N issues of the vector. For the preliminary tests, N has been set to 64. The developed software modules will be described in the following subsections.

4.1 CanRx module

The CanRx process is in charge of receiving the CAN messages issued by the devices connected to the bus. The received messages are decoded with the aim of getting the numerical values expressing the physical quantities of interest. The decoded values are then written to the proper subportion of a previously created shared memory segment. A CAN message has a maximum length of 8 bytes (64 bits). The bits in a single message are usually employed to carry more than one value of interest. For example, bits from 0 to 15 carry motor speed value, bits from 16 to 23 carry engine torque value, and so on. Numerical values of interest, carried by some bits in a CAN message, are referred to as signals. Getting the physical quantities from a CAN message usually requires knowledge about meaning and location of each signal (e.g. bits from 0 to 15 of the CAN message with a certain ID number carry motor speed information). Moreover, information on how the value is encoded is necessary. Decoding a signal typically involves: converting the number from binary to decimal, then applying an affine transformation consisting in possible multiplication by a scale factor (gain) and addition of an offset value. The employed CAN message ID numbers, and the meaning, location and encoding of each signal vary from device to device, and each company typically adopts proprietary conventions, due to the lack of a widespread standard scheme. In order to achieve a fully general software module, the mapping between CAN messages and physical quantities of interest is not hard-coded in the executable, but read on start-up from a configuration file. The file tells the CanRx module the IDs of the CAN messages to be processed along with the location, name and decoding information for each signal of interest. The configuration file can be easily edited on a PC by means of the provided GUI interface. The CanRx module performs some initialization steps and then enters an infinite loop in which the messages received from the CAN controller are processed. The program flow can be summarized as follows (the parameters in italics can be provided by the user from the command line):

1. Open *can device* in the requested *mode* (blocking or non-blocking) and set the *baud rate*.

2. Read up the configuration file placed in a conventional disk location.

3. Access or create the shared memory segment and semaphore, identified by a conventional key.

4. Write the ancillary information on the shared memory (name, range, measure unit of each signal).

5. Enter the infinite loop.

   (a) Read CAN messages from the message queue.
   (b) Get current time from the system clock, for time stamping.
   (c) Decode the messages to get the signal values.
   (d) Write up time stamp and signal values to the state vector in the shared memory circular buffer, at the current cursor position.
   (e) If CAN read mode is non-blocking, sleep for some *sleep time*.

Some details about the user parameters follow. The *can device* to be opened is usually identified in UNIX-like systems by a device file under the /dev directory (e.g. /dev/can0). The *baud rate* to be set for the provided device is expressed in Kbit/s. The reading *mode* affects the kind of read operation, blocking or non-blocking. In blocking mode, the read operation does not return until a message is actually received. This could block the process for a while. In non-blocking mode the read operation returns immediately anyway, even if no message has been received. In this case, to avoid continuous access of the process to shared resources (e.g. CPU, shared memory), a sleep call has to be performed to stop the process for a *sleep time* expressed in milliseconds. If no device is provided from the command line, a simulation mode is activated, in which the read operations are simulated with randomly generated CAN messages. The simulated reading mode is of non-blocking type.

4.2 DataRec module

The DataRec module is in charge of dumping the information passing through the shared memory segment to a persistent storage device. More precisely, every issue of the status vector, along with its time stamp, has to be output to a log file. Each file has also the ancillary information about signals reported inside of it, and it has a unique name allowing to retrieve the interval of time it refers to. The main requirements the DataRec module has to meet are the following:

1. It must not miss any issue of the state vector appearing in the proper buffer inside the shared memory segment.

2. It has to handle the case in which the disk it is writing gets full. In this situation the older log records have to be erased to make room for the new ones (FIFO policy, even if more sophisticated strategies could be employed).

3. It must have a lower priority with respect with other processes (CanRx, graphics) in accessing the shared resources (mainly CPU and shared memory).

4. The generated log files must be easy to browse and suitable for post-processing.

Data in log files are organized as follows. Each row contains one issue of the state vector, with the time stamp on the first column. Columns are separated by TAB characters, ancillary information is reported at the end of the file. This way, the generated files are easy to load with well known software packages like MATLAB or spreadsheet applications. The number of rows contained in each file is fixed. This way, all files have a similar size, chosen with the idea of getting a good compromise between the time interval covered by a single file, and the ease of loading, browsing and processing. The actual time interval covered by a file depends on the rate at which new state vectors are written to the buffer. The DataRec module performs some initialization steps and then enters an infinite loop in which the state vector issues read from the shared memory segments are dumped to the chosen media. The program flow can be summarized as follows (the parameters in italics can be provided by the user from the command line):

1. If the given *path* is an accessible directory, access or create the log file directory inside of it.

2. Check free disk space. If disk space is not sufficient, try to erase older DataRec log files, if present.

3. Access the shared memory segment and semaphore, identified by a conventional key.

4. Read up the ancillary information from the shared memory (name, range, measure unit of each signal).

5. Enter the infinite loop.

   (a) If first iteration, or if a log file has been closed, start a new log file. This involves generating its name from clock time and checking for free disk space. If space if not sufficient, erase older DataRec log files.
   (b) Until completion of the log file, perform the following steps in loop.
      i. Read up the state vector issues from the shared memory buffer. The considered state vector circular buffer positions are the ones ranging from one after the last read position until the current position set by the writing modules. The read state vectors are immediately dumped to disk.
      ii. Sleep for some *sleep time*.
   (c) Write the ancillary information to the completed log file and close it.

Some details about the user parameters follow. The *path* has to be necessarily provided, it has to be a valid directory on the chosen media to write onto. There must be a predetermined minimum free disk space, or otherwise some disk space has to be occupied by older DataRec log files, which can be erased by the DataRec module. The *sleep time* is expressed in milliseconds.

4.3 Dashboard module

The Dashboard module, still in a development stage, is in charge of performing a suitable visual rendering of some of the values contained in the status vector. This is done by simulating some typical analog or digital monitoring controls, like gauges, LED arrays, 7 or 16 segments alphanumeric displays. The controls are expected to exhibit a dynamic behaviour similar to the one of their hardware counterparts, in particular the velocity of variation and inertia have to be reproduced for a comfortable and easily readable rendering. The screen drawing function is carried out trough the facilities of the nano-X graphic server, providing a programming interface which allows to get a handle to the screen and to draw very low-level geometrical patterns like lines, arcs, rectangles and so on. The nano-X server also provides an interface for getting input from the user (e.g. through a touchscreen). In order to draw the hardware-like controls able to display the current value of a signal, a collection of C functions have been written and some data structures have been defined. Each data structure is in charge of holding the current status and layout of a control (e.g. the number of LEDs in a LED array, the color of on and off LEDs, size and position of the LEDs, mapping between the number of lighted LEDs and the value to represent, inertia, currently displayed value, etc...). The function in charge of managing a control takes the current status structure and the new value to display as its parameters, and updates the graphical rendering of the control accordingly. An example of digital dashboard generated using the Dashboard module is shown in Fig 4. In order to achieve a reusable software module, the dashboard layout is specified via a configuration file placed in a conventional location. The file contains the type and layout of the controls to draw, the value that each of them has to display (signals are identified by their name, reported among the ancillary information on the shared memory segment), the mapping between the position of the control (number of lighted LEDs, gauge angle, etc...) and the numerical value. The DataRec module performs some initialization steps and then enters an infinite loop in which some of the status vector components read from the shared memory segment are graphically rendered to screen. The program flow can be summarized as follows (the parameter in italics can be provided by the user from the command line):

1. Read up the configuration file placed in a conventional disk location.
2. Access the shared memory segment and semaphore, identified by a conventional key.
3. Read up the ancillary information from the shared memory (name, range, measure unit of each signal).
4. Connect to the nano-X graphic server and create the main window.
5. If some correspondence is found between the signal names on shared memory and on configuration file, create and initialize the data structures for the controls specified on configuration file.
6. Enter the infinite loop.
   (a) Read the current values of the signals to display, accessing the most recent status vector in the shared memory.
   (b) Update each graphic control.
   (c) Sleep for some *sleep time*.

The *sleep time* is expressed in milliseconds, it should be set to a value assuring a proper update rate of the screen. The usual refresh rate of a visual information ranges from 20 to 30 frames per second.

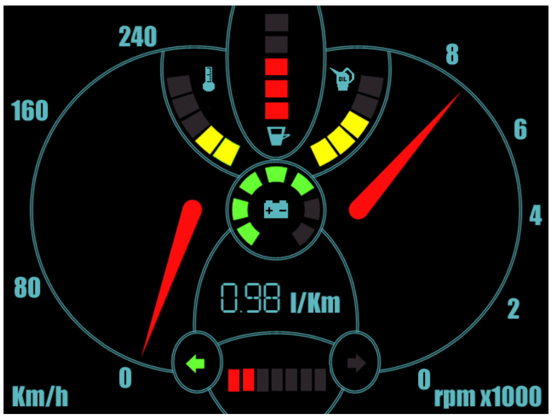

Figure 4: Example of digital dash board generated using the Dashboard module.

5 Future Developments

In the first stage the CanRx and DataRec modules have been realized in a simple, not highly optimized form. The next development step will involve the full implementation of three new modules: Dashboard, SerialRx and DataSend. The core graphical routines for the Dashboard module have already been realized and some demo programs have been carried out. The complete implementation of the module requires the capability of connecting to the shared memory segment and reading from it in a way rather similar to the one of the DataRec module. The other key feature to realize is the capability to automatically create and setup the required graphical controls according to a configuration file read from disk. The SerialRx module will read automotive signal information from a serial (RS232, RS485) port, decode it and write it to the shared memory segment like the CanRx module. In order to realize it, some information must be got as concerns the way automotive devices (drives, battery packs, etc...) encode signal values in serial communication character streams. Once an adequately general outline of encoding techniques has been achieved, a method must be studied to make the SerialRx module reusable in a wide range of contexts, by the definition of a proper configuration file describing the encoding style of the expected information. The DataSend module will read the information contained in the status vector issues located in the shared memory segment, like the DataRec and Dashboard modules. The DataSend module will be in charge of transmitting the status vector information over a range of possible channels (GPRS, WiFi LAN, etc...) for telemetry and VANet (Vehicular Adhoc Network) applications. In order to implement the DataSend module, information has to be acquired concerning the features of the different wireless technologies and the programming interfaces used to communicate with the relative hardware controllers.

After the first implementation, modules have to undergo an accurate optimization step. In fact, the limited resources available on embedded systems like the GPX21 require a high level of computing efficiency. The preliminary tests carried out on the CanRx and DataRec modules show that the time requested by the steps performed by the processes (information decoding, disk writing, memory access, etc...) can be very difficult to control in presence of limited resources (CPU, memory, etc...) and in absence of accurate code optimization. Achieving the best efficiency is thus necessary in order to make possible a strict time scheduling of the processes. Each module will be optimized and different configurations including from two to five modules running simultaneously will be thoroughly tested. In order to carry out the tests, an USB-to-CAN interface for PC will be employed. By using this device, it is possible to feed the GPX21 running the modules with a stream of known CAN messages sent from a PC. This way, the capability of processing high quantities of incoming data and to report accurate time stamp information will be carefully measured and analyzed.

The signals elaboration system described so far will be mounted and tested in the Polo per la Mobilità Sostenibile prototypes. The most imminent project is the realization of a formula SAE vehicle (7) build to compete in the formula SAE Italian race. This prototype will be equipped with a Honda CBR motor and a Walbro ECU which offers communication through RS232 communication port. The most important quatity to monitor in race vehicles is the engine rotational frequency which need to be visulized in a flashy fashion to give an importatn feedback to the driver. Another important quantity to be dispalyed is the vehicle speed. Besides these classical information a few uncommon quantities need to be dispalyed, namely, the engine water temperature, the battery voltage, the exhaust oxigen sensor, the opening time of the cylinder injector and the advantage ignition angle of the cylinder. A screenshot of the dashboard designed for the formula SAE vehicle is shown in Fig. 5. Another prototype under development which will be equipped with the proposed system is the Bizzarrini P538 Eco Targa Florio. This prototype is a LPG-gas electric hybrid-propulsion reissue of the Italian sports car Bizzarrini Livorno P538 (originally issued in 1967). All the devices mounted on the P538 prototype (thermal and electric engine, Battery Management System, ECU) communicate through CAN bus. More precisely, the vehicle will mount an Alfa Romeo Tehrmic engine, a permanet magnet electric motor by CIS and a Kokam battery pack with MIRMU Battery Managment System (BMS). The real-world application of the proposed system to the mentioned prototypes will provide valuable information for further development and improvement.

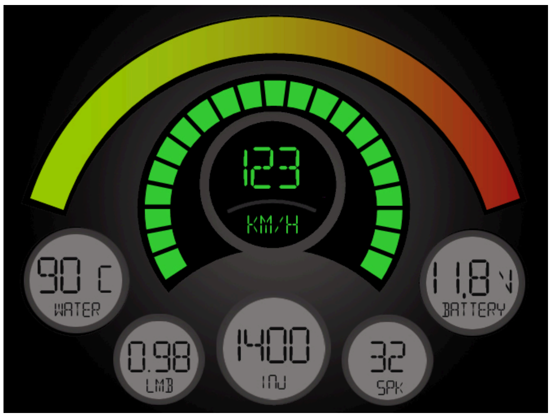

Figure 5: Digital dash board for formula SAE generated using the Dashboard module.

References

[1] A. Puschnig, R. T. Kolagari, *Requirements engineering in the development of innovative automotive embedded software systems*, in Requirement Engineering Conference, 2004. Proceeding 12th IEEE International, pp. 328-333.

[2] M. Rodelgo-Lacruz, F. J. Gil-Castineira, F. J. Gonzalez-Castano, J. M. Pousada-Carballo, J. Contreras, A. Gomez, M. V. Bueno-Delgado, E. Egea-Lopez, J. Vales-Alonso, J. Garcia-Haro, *Base technologies for vehicular networking applications: review and case studies*, in IEEE International Symposium on Industrial Electronics, ISIE 2007, pp. 2567-2572.

[3] I. Katramados, A. Barlow, K. Selvarajah, C. Shooter, A. Tully, P. T. Blythe, *Heterogeneous sensor integration for intelligent transport systems*, in Road Transport Information and Con-

trol - RTIC 2008 and ITS United Kingdom Members' Conference, IET, 2008, pp. 1-8.

[4] *Pomos*, http://www.pomos.it/, accessed on 2009-04-04.

[5] *Engicam*, http://www.engicam.com/, accessed on 2009-04-04.

[6] *nano-X*, http://www.microwindows.org/, accessed on 2009-04-04.

[7] *Formula SAE*, http://students.sae.org/competitions/formulaseries/, accessed on 2009-04-04.

Authors

Dr. Maurizio Paschero is a post doctoral research associate at the Information and Communication Department (INFOCOM) of the University of Rome ”La Sapienza” since 2008. He received the Laurea degree in Electronics Engineering in 2003 and the Ph.D in Information and Communication Engineering in 2006 from the University ”La Sapienza” of Rome and the Ph.D. in Mechanical Engineering in 2008 from Virginia Polytechnic Institute and State University. His major fields of interest include smart structure, stability of structure, circuital modeling and synthesis, neural networks, fuzzy systems.

Dr. Guido Del Vescovo is a post doctoral research associate at the Information and Communication Department (INFOCOM) of the University of Rome ”La Sapienza” since 2008. He received the Laurea degree in Electronics Engineering in 2004 and his Ph.D in Information and Communication Engineering in 2008 from the University of Rome ”La Sapienza”. His major fields of interest include supervised and unsupervised data driven modeling techniques, neural networks, fuzzy systems, evolutionary algorithms and granular computing.

Dr. Antonello Rizzi is an Assistant Professor at the Information and Communication Department (INFOCOM) of the University of Rome ”La Sapienza” since 2000. He received the Laurea degree in Electrical Engineering in 1995 and the Ph.D in Information and Communication Engineering in 2000 from the University of Rome ”La Sapienza”. His major fields of interest include supervised and unsupervised data driven modeling techniques, neural networks, fuzzy systems and evolutionary algorithms. In particular, he is currently working on Granular Computing and hierarchical reasoning. He is author or co-author of more than 50 publications. Since 2008, he serves as Director of the Intelligent Signal Processing and Inductive Modeling Systems Laboratory of the ‘Polo per la Mobilità Sostenibile della Regione Lazio’ (Sustainable Mobility Research Center), INFOCOM Department.

Prof. Fabio Massimo Frattale Mascioli received the Laurea degree in Electronic Engineering in 1989 and the Ph.D. degree in Information and Communication Engineering in 1995 from the University ”La Sapienza” of Rome. In 1996, he joined the INFOCOM Department of the University ”La Sapienza” of Rome as Assistant Professor. Since 2000, he has been Associate Professor of Circuit Theory at the same department. His research interest mainly regards neural networks and neuro-fuzzy systems and their applications to clustering, classification and function approximation problems. Currently, he is also working on circuit modeling for vibration damping, energy conversion systems and electric and hybrid vehicles. He is author or co-author of more than 70 papers. Since 2007, he serves as scientific director of the ‘Polo per la Mobilità Sostenibile della Regione Lazio’ (Sustainable Mobility Research Center), INFOCOM Department.