Serial Bus Monitoring Software for Microcontrollers Embedded in Mechatronic Systems

Mircea Popa

Computer and Software Engineering Department, “Politehnica” University of Timisoara, No. 2, Vasile Parvan Blv., 300223 Timisoara, Romania
Phone: (40) 256 403275, E-mail: mircea.popa@ac.upt.ro

Marian Vasile

Computer and Software Engineering Department, “Politehnica” University of Timisoara, No. 2, Vasile Parvan Blv., 300223 Timisoara, Romania
Phone: (40) 256 403275

Sebastian Fuicu

Computer and Software Engineering Department, “Politehnica” University of Timisoara, No. 2, Vasile Parvan Blv., 300223 Timisoara, Romania
Phone: (40) 256 403263

Abstract: The automotive systems hold an important place in the large area of mechatronic systems. More and more intelligent embedded modules are included in the automobiles. All those modules are based on microcontrollers or similar circuits. A lot of communication protocols were developed for transferring data among the microcontrollers embedded in an automobile. Among them, the SPI bus was intended primarilly for testing, debugging and diagnostic applications. The monitoring of the SPI bus is approached in this paper. SPI (Serial Peripheral Interface) is a master – slave bus supported by many microcontrollers. Starting from an existing hardware which connects a SPI bus to a PC via the CAN bus, dedicated software was created for accessing the information flowing on a SPI bus between several modules. There were time and frequency constraints imposed by the application and structural constraints imposed by the given hardware based on a ST10F276 microcontroller.

Keywords: mechatronic systems, automotive systems, microcontroller, serial communication, SPI bus

1 Introduction

As part of the huge diversity of the mechatronic systems, the automotive systems hold an important place. More and more intelligent embedded modules are included in the automobiles. All those modules are based on microcontrollers or similar circuits. Modern automobiles contain tens of microcontroller based embedded systems and the communication between them became a critical issue.

A lot of communication protocols were developed for transferring data among the microcontrollers embedded in an automobile. All are serial and several of them are, [1]: J1850 bus (used for diagnostics and data sharing applications), MI bus (used to drive mirrors, seats, window lifts or head light levelers), DSI bus (it links sensors and components in the automobile), BST bus (used especially for the air-bags), MML bus (it is a multimedia bus), Byteflight or SI bus (used for safety critical applications in automobiles), FlexRay (it is a high speed serial communication bus for in-vehicle networks), D2B bus (connects audio, video, computer peripheral and telephone components in a single ring topology within the automobile), IDB-1394 (it is the automotive version of the IEEE 1394 or Firewire protocol), OBDII bus (it defines a communication protocol and a standard connector to acquire data from passengers), SAE J1708 bus (used for transferring data between embedded systems in heavy-duty vehicle applications), LIN bus (used for in-vehicle communication and networking serial bus between intelligent sensors and actuators), CAN bus (used for low-speed applications as window or seat control but also for high-speed applications as engine management or brake control) and finally SPI bus (used for different in-vehicle communication and testing, debugging and diagnostic applications).

SPI (Serial Peripheral Interface) is a one master – many slaves bus supported by microcontrollers from different families. In order to build SPI based multi-microcontroller systems and for fastening the testing and debugging processes, tools for monitoring the bus are necessary. There are such classic tools: digital oscilloscopes, logic analyzers etc. They offer information at the bit level, timings, voltage levels but they lack a consistent software component and are expensive because of the many functionalities they have but are not useful for the monitoring of the SPI bus.

Another solution is to build a dedicated tool through which a PC can monitor the SPI bus, accessing all the data transferred on the bus and also being able to insert packets on the bus. This tool is financially justified only if it is used where the SPI bus is widely used. One can find microcontrollers which offer the hardware support for a bridge between the SPI bus and other interfaces but the software component is specific for an application or a family of applications and must be built for each case.

A dedicated tool for monitoring a SPI bus is presented in this paper. The tool ensures the access of a PC to a SPI bus through a module which achieves a bridge between the SPI and CAN buses. The PC is connected to the CAN bus through a dedicated card. The CAN bus was used between the PC and the SPI bus because the PC – CAN connection is a solved problem and there are fields in which both the SPI and CAN buses are used for connecting microcontrollers, the automotive field being a good example. Starting from a given hardware module, based on a ST10F276 microcontroller, a dedicated software module was created which allows the PC to monitor and communicate with the SPI bus offering to the user the possibility to receive all the transferred data on the bus for test and debug purposes.

The next section presents other similar works, the third section details the tool created, the fourth section reports performances and the last section outlines the conclusions.

2 Related Work

The problem of accessing a SPI bus through other interfaces was treated in other papers too.

Reference [2] describes a local SPI sensor bus for transferring data from several smart sensors to the Internet. The data is converted in 8 bits ASCII code which can be read directly by any computer through its serial RS232 port. From the point of view approached in this paper, the work can be seen as a SPI – RS232 bridge.

Another achievement is presented in [3]. Sensor node architecture for distributed computation and sensing in distributed embedded systems is described. It is based on an Atmel ATMega 128L microcontroller which includes several serial interfaces (at the hardware level) such as: I2C, RS232, RS485 and SPI. Bridges can be built between these interfaces but a consistent software component is necessary to be created.

A general purpose CAN bus based board is presented in [4]. It is based on a microcontroller with several build - in interfaces, including a CAN interface, but not a SPI interface, through which transfers can be achieved between those interfaces. The specific software component must be created.

A different and more general approach is described in [5]. The conversion between several interfaces is done through the reconfigurable hardware process, particularly in wearable systems. In such systems the high performance tasks (e.g. video decoding) must be done in severe low energy consumptions conditions, for maximizing the battery life. The concept and a prototype implementation of an autonomous wearable unit with reconfigurable units are described. The reconfiguration is done by ASIC and by adaptive interfaces.

Unlike the above mentioned achievements, a monitoring tool for a SPI bus is presented in this paper which is cheap, having a single major functionality and ensures the connection to the CAN bus, which is widely used in embedded systems. It is based on a given hardware module, built around a ST10F276 microcontroller. The software layer was created which allows a PC to access a SPI bus via a CAN bus. The PC can monitor and communicate with the SPI bus offering to the user the possibility to test and debug an existing SPI system or a helpful tool for creating a new SPI system.

3 The SPI Monitor

3.1 Hardware

Figure 1 presents the position of the SPI Monitor in a SPI system.

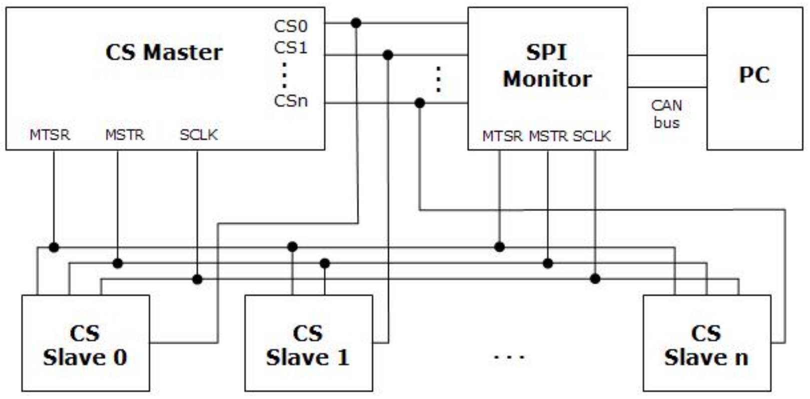

Figure 1
The position of the SPI Monitor

The SPI bus is a one master – many slaves synchronous serial bus, [6]. The communication media is made by 3 lines: the serial clock, SCLK, the receive data line and the transmit data line. All the receive data lines of the slaves are connected to the transmit data line of the master and all the transmit data lines of the slaves are connected to the receive data line of the master. The master selects each slave through a CS (Chip Select) line.

The SPI Monitor is connected to the SPI bus through two lines: the serial clock, for synchronization with the SPI monitored bus and the receive data line, for acquiring the data. Those two lines belong to the microcontroller SPI interface. The SPI Monitor has also a number of input lines equal to the number of the slaves, connected to the CS slave lines, for determining the moment when a CS, meaning a slave, was activated by the master. The SPI Monitor is connected to the PC via the CAN bus, [7]. The PC has a CAN card which interfaces it to the CAN bus. The data captured on the SPI bus is sent by the SPI Monitor through the CAN bus to the PC, where the user can visualize it using a special tool responsible with spying the information flowing on the CAN bus.

The SPI Monitor has a given hardware support, a board based on the ST10F276 microcontroller, [8]. It is a high performance microcontroller, several of its features being the following ones: 16 bit CPU with 4 stages pipeline, single – cycle context switching support, flexible memory organization with high capacity FLASH and RAM memories both on – chip and external, fast and flexible bus, with programmable external bus characteristics for different address ranges, multiplexed – demultiplexed external address/ data busses and five programmable chip select signals, powerful internal peripheral system with counters, PWMs, A/D converters, USART, I2C and two CAN serial channels, advanced interrupt system with 16 priority level interrupt system with 56 sources and 8 channel Peripheral Event Controller for fast, single cycle interrupt driven data transfer, watchdog timer, on – chip bootstrap loader, real time clock, low power modes and up to 111 general I/ O port lines individually programmable.

Starting from this hardware support, the whole software layer was created.

Because of the time constraints fast transfer mechanisms had to be used. The data acquisition was made through the interrupt system, including the PEC interrupts. Because of the high amount of time spent with entering and leaving one interrupt service routine, the code of those routines was minimal (it was written in assembly language and the CPU clock cycles were carefully counted for each instruction). Likewise, the capture of the external interrupts, corresponding to the chip selects used by the SPI master to activate the SPI slaves, had to be in minimal time, the fastest mechanism of the ST10 microcontroller was used, i. e. the fast external interrupts. Also for fastening the transfers, the CPU and the CAN1 controller of the microcontroller worked in parallel in many situations. The reason was that that the process of transmitting the monitored data to the PC is a very slow one. The CPU is responsible with preparing in advance the messages sent to the PC on the CAN bus by the CAN1 controller. When the CAN1 controller finishes the transmission of one message it finds an already prepared message ready to be sent.

Another limitation was the amount of RAM memory which could be used. Taking into account that the reading of the SPI monitored data is a much faster process than transmitting it to the PC on the CAN bus, large buffers and an adequate technique for managing them were needed. The chosen solution was a big circular buffer. It is used commonly to store the information monitored from all SPI slaves. The loading of the SPI monitored data into the buffer is made by the interrupt service routine corresponding to the SPI interface of the ST10 microcontroller. A manager, implemented by a function that runs continuously when no other function runs, ensures the unloading of the buffer and sending to the PC the data. At a very high transmission rate of the SPI frames on the SPI bus, this buffer can become full. As a consequence the monitoring process is stopped and it is waited for the moment when the manager empties the buffer. After that the monitoring process restarts.

3.2 Software

Figure 2 presents the structure of the software application.

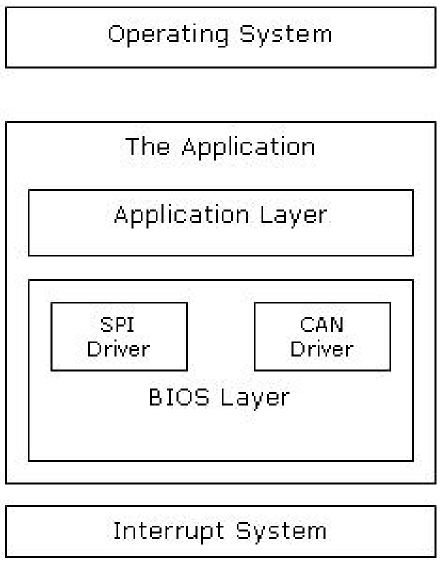

Figure 2
The structure of the software application

The software application is built on two layers, the Applicative Layer (the upper) and the BIOS Layer (the lower) and uses other two layers, the Operating System (OS) layer and the Interrupt System (IS) layer. The OS is responsible for executing the activities assigned by the programmer according to a schedule. It initializes the whole application, servers the external and internal watchdogs in order to avoid the CPU reset and calls continuously the manager of the upper layer when no other instructions are executed by the CPU. The IS is responsible for executing the interrupt service routines of the drivers that are on the inferior layer whenever the corresponding interrupts occur. These two special layers are independent one from the other. The OS is a time – driven layer, meaning that it calls its functions based on a time schedule and the IS is a event – driven layer, since it calls the ISRs based on the occurrence of some events which are not respecting any time rules (the interrupts appear asynchronously).

The BIOS Layer contains the SPI and CAN drivers. Only the SPI driver was created, the CAN driver was given. The SPI driver contains an initialization function and the SPI interrupt service routines: the one for handling the receiving of SPI data, the one for detecting the SPI errors and also the ISRs corresponding to the activation to the activations of the SPI slaves chip selects. The SPI and CAN drivers offer support for the Applicative Layer independently from the OS because in the case of the CAN driver its functions are directly called by the upper layer, according to the strategy of the application manager, and in the case if the SPI driver its ISRs are executed based on SPI external events that do not depend on the OS (the ISRs are called by the IS). The SPI driver is responsible for storing the SPI monitored data into the control and data buffers, which will be used by the upper layer, and to detect the SPI errors. The CAN driver offers to the upper layer the interfaces for initializing and starting the CAN communication, for reading and writing messages on the CAN bus, for reading the state of a CAN logical channel, for reading the state of the CAN1 controller and for executing the corresponding ISR when a CAN message is received from the PC.

The Applicative Layer contains the manager, implemented as a function, which loads the SPI monitored data (stored by the SPI driver in the buffers) into messages to be sent to the PC on CAN. It restarts the SPI monitoring process if it has been stopped because the buffer was full and loads the new configurations sent by the user for the SPI slaves. Also it notifies the user about the SPI errors detected. This layer is called also the upper layer since it contains the manager of the whole application, which ensures that the data monitored from a SPI bus is routed to the CAN bus.

Both the SPI and CAN drivers are implemented using one or more software modules. Each module consists of a C source file (.c) and a C header file (.h).

The created and implemented modules are:

- SPI driver: Bspi.c (contains: the SPI driver initialization function, the RX and Error ISRs, the CSx ISRs and others), Bspi.h (contains general SPI driver macros), Bcnfspi.c (contains the configuration structures for the SPI slaves) and Bcnfspi.h (contains: the configuration of the Interrupt System regarding the SPI driver, the configuration of the PEC registers, the configured data buffer length, the configured number of slaves, the SPI baud rate macro and others),

- Applicative Layer: Vsmc.c (contains: the manager of the entire application, the function for loading new configurations for the SPI slaves and others) and Vsmc.h (contains general application macros).

Next, some details about the code will be given, e. g. the RX and Error interrupt service routines. The RX ISR has the following purposes: to update the RX PEC counter when it reaches zero (main purpose), to update the RX PEC destination pointer when it exceeds the data buffer superior boundary, to mark the current SPI frame as corrupt when the data buffer becomes full and to signalize the occurrence of the data buffer full condition.

RX PEC is the PEC channel used for transferring the newly arrived SPI characters to the data buffer of this application. The SPI characters come from SSC (High – Speed Synchronous Serial Interface) more exactly from the SSCRB register (one of the SFR registers of the SSC interface). The Applicative Layer takes the SPI characters corresponding to one SPI frame from this buffer and sends them on the CAN bus to be visualized by the user. When the RX PEC counter reaches zero, it has to be reloaded by the corresponding ISR, this one, in order to further transfer the SPI characters from the SSCRB receiving register to the application data buffer.

The routine starts by verifying whether the RX PEC destination pointer (DSTP) exceeds the superior boundary of the data buffer and if so, the destination pointer is set up to point to the beginning of the data buffer.

The next step is to check the data buffer full condition. If this condition is found activated it will be verified if the control write index is or is not zero. If it is zero the slave ID of the last cell of the control buffer, corresponding to the currently received SPI frame, is marked as corrupt. This means that characters of this SPI frame will be lost because of the data buffer full condition.

The next operation is to verify if the RX PEC destination pointer equals the data read cursor. If they are equal it means that the writing to the data buffer is faster than the reading from the data buffer and that the RX PEC wants to write over unread data. As a consequence the data buffer full condition is signalized. In this case, the RX PEC is not updated, it remains zero and when the next SPI character will arrive it will be lost because the RX PEC will not transfer it into the data buffer. This situation, may last for more than one received SPI characters (all will be lost) until at least one character from the data buffer was transferred by the upper layer, so the transfer from the SSCRB register to the data buffer can continue.

If the RX PEC destination pointer is not equal to the data read cursor, meaning the data buffer is not full, one must signalize this situation in case that the data buffer has previously been full in order to loose no more SPI characters because now there is free space in the buffer. Then, the RX PEC counter must be updated.

Next, it is verified if the RX PEC destination pointer is greater than the data cursor. If this is true, all the cells to the end of the data buffer have been freed by the upper layer and the new value for updating the RX PEC counter is calculated from the current pointing position of the RX PEC destination pointer to the last cell of the data buffer. If the condition is false, the new value for updating the RX PEC counter is calculated from the current pointing position of the RX PEC destination pointer to the cell indicated by the data read cursor, because this cell cannot be overwritten with a new SPI character since it was not read by the upper layer and transmitted on the CAN bus. Figure 3 presents the Rx ISR diagram.

The Error ISR signalizes an SPI error occurrence. Phase error and baud rate error can be detected. Whenever a SPI communication error is detected on the SPI bus, it has to be notified to the upper layer in order to be signalized to the user through sending a message on the CAN bus. Figure 4 presents the Error ISR diagram.

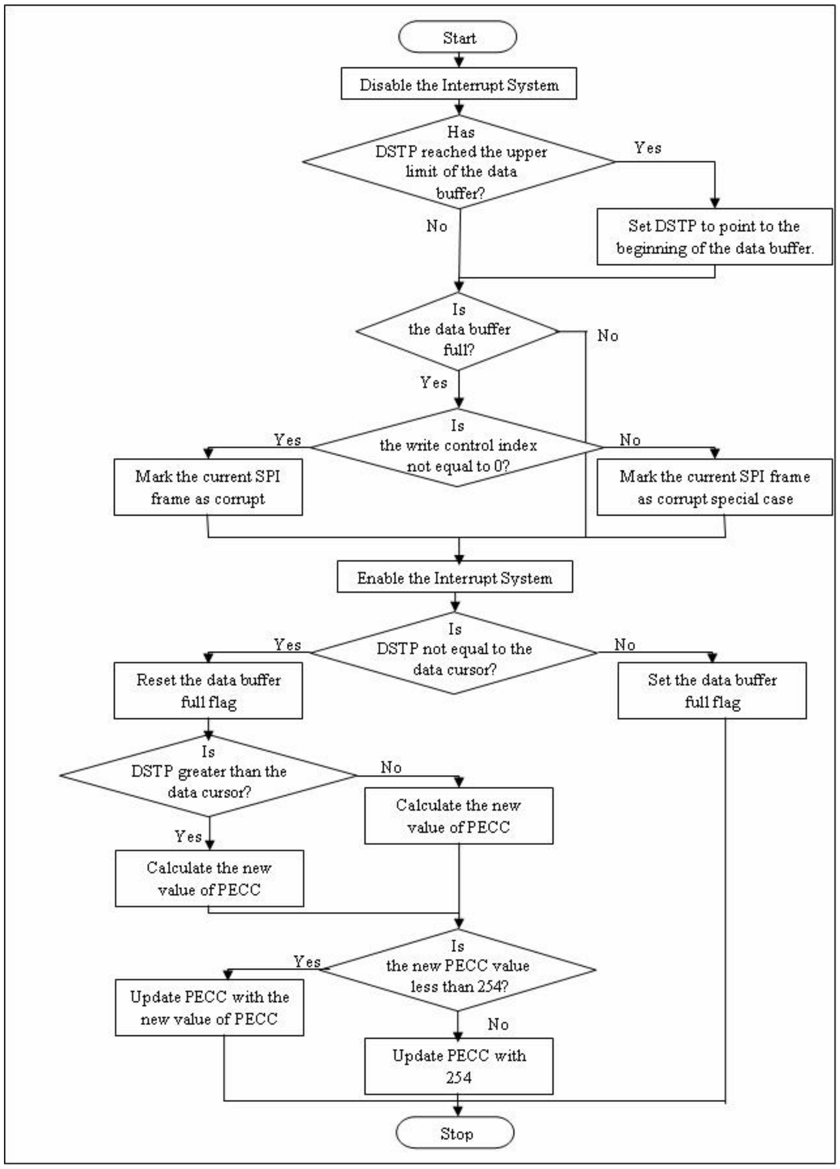
Start
Disable the Interrupt System
Has DSTP reached the upper limit of the data buffer?
Yes
No
Set DSTP to point to the beginning of the data buffer.
Is the data buffer full?
No
Yes
Is the write control index not equal to 0?
Yes
No
Mark the current SPI frame as corrupt
Mark the current SPI frame as corrupt special case
Enable the Interrupt System
Is DSTP not equal to the data cursor?
Yes
No
Reset the data buffer full flag
Set the data buffer full flag
Is DSTP greater than the data cursor?
No
Calculate the new value of PECC
Yes
Calculate the new value of PECC
Is the new PECC value less than 254?
Yes
Update PECC with the new value of PECC
No
Update PECC with 254
Stop

Figure 3
The RX ISR diagram

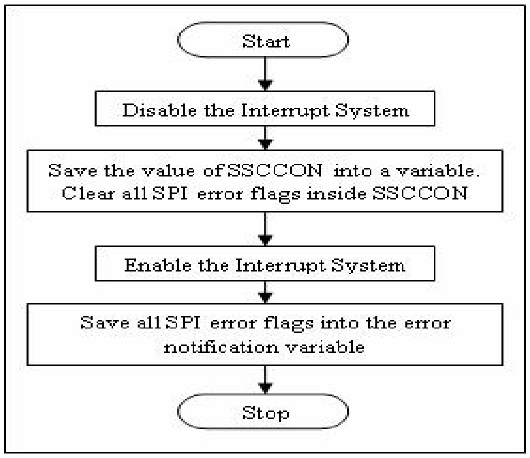
Start
Disable the Interrupt System
Save the value of SSCCON into a variable.
Clear all SPI error flags inside SSCCON
Enable the Interrupt System
Save all SPI error flags into the error notification variable
Stop

Figure 4

The Error ISR diagram

The routine starts by disabling the interrupt system in order to prevent the routine interruption by a higher priority interrupt. The SSCON register is saved, the error flags are cleared and the interrupt system is activated.

The operations which follow are not critical and process the variable where the SSCON register was saved. They can be interrupted. The result is the value that will be transmitted by the upper layer on the CAN bus in order to notify the user of the appeared SPI error. The reason why the SSCON register is saved in a local variable and not processed itself during the amount of time when the interrupt system is deactivated is because this operation has is longer than the saving operation. When the critical part was executed and the interrupt system was activated again, the following operations, which are not so critical, can be interrupted.

4 Performances

The obtained performances satisfied all the requirements of the application.

The desired maximum baud rate was 1 Mbps and the obtained one was 6 Mbps. Another time requirement was to handle the moment between the activation of the CS signal and the occurrence on the SPI data line of the first bit from a frame in less than 2µs. The measurements of the duration of the specific sequence execution time showed a maximum of 1.95µs, at 44MHz clock frequency.

For seeing other performances too, the SPI monitor was used in a real project. The

SPI system was made by one master and two slaves. The communication with one of the slaves was achieved through frames with one character of 8 bits and with the other one through frames made by nine characters of 8 bits. The SPI baud rate for communication was 250 Kbits. The measurements showed that the memory consumption was very low: the entire application used only 162 bytes of RAM, 78 of them being allocated for the control and data buffers. These low RAM needs are an important advantage of the application because the memory space in embedded system is always drastically limited.

Conclusions

The created monitoring tool offers to the user the possibility to receive all the data transferred on a SPI bus and to test or debug the bus. The user accesses the SPI bus through a PC with added CAN interface which is connected to a CAN – SPI module. The user receives the information in numerical format, well structured and easy to read. The received information includes the SPI slave ID, taking part at the communication, and also a timestamp which makes possible the placement of the SPI data on time scale, without needing an oscilloscope.

Further improvements are possible, by adding several modules and features, such as: to support SPI characters with more than 8 bits, to monitor simultaneously both data lines of the SPI bus: the receive and the transmit lines, to improve the performances by implementing the critical parts of the application in assembly language and by optimizing the code, to create a complex graphical user interface (GUI) and to use both CAN controllers included in the ST10 microcontrollers working simultaneously for increasing the performance.

References

[1] http://www.interfacebus.com/Design_Connector_Automotive.html

[2] D. Wobschall, H. S. Prasad: Esbus – a sensor bus based on the SPI serial interface, in Proceedings of IEEE Sensors, Orlando, USA, June 2002, pp. 1516-1519

[3] A. Savvides, M. B. Srivastava: A Distributed Computation Platform for Wireless Embedded Sensing, in Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD’02), IEEE Computer Society, Freiburg, Germany, September 2002, pp. 220-225

[4] H. Boterenbrood, B. Hallgren: The Development of the Embedded Local Monitor Board (ELMB), in Proceedings of the 9th Workshop on Electronics for LHC Experiments, Amsterdam, Niederlands, September – October 2003

[5] C. Plessl, R. Enzler, H. Walder, J. Beutel, M. Platzner, L. Thiele, G. Troster: The case for reconfigurable hardware in wearable computing, Personal Ubiquitous Computing, Vol. 7, pp. 299-308, 2003

[6] SPI description – http://www.rpi.edu/dept/ecse/mpsd/SPI.pdf

[7] CAN Specifications – http://www.infineon.com/cmc_upload/migrated_files/document_files/Application_Notes/can2spec.pdf

[8] ST10F276 Data Sheet – http://mcu.st.com/mcu/index.php