IMPLEMENTING TINY EMBEDDED SYSTEMS WITH NETWORKING CAPABILITIES

Renato Jorge Caleira Nunes
Instituto Superior Técnico / INESC-ID
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
renato.nunes@dei.ist.utl.pt

ABSTRACT

In this paper we present a hardware and software approach that allows the development of very simple embedded systems with networking capabilities. These embedded systems were developed in the context of domotics with the objective of offering an inexpensive platform to allow a cost-effective implementation of super-automated homes, i.e., homes with a very big number of sensors and actuators (several hundreds). The Domotic Modules (DM) are based on a microcontroller with very limited resources (just 8 Kbytes of Flash memory for code and 512 bytes of data memory). In spite of these restrictions we were able to implement a robust communication protocol and deploy a co-operative real-time multitask environment that support multiple applications and inter-task communication. We describe our approach and give emphasis to its generality and portability that allow its usage in other embedded platforms with very limited resources. It is entirely implemented in C language, offering a good performance, real-time characteristics and a flexible solution to problems that require communication and would benefit from a software structuring model based on tasks.

KEYWORDS

Embedded System, Real-Time System, Networking, Home Automation, Domotics.

1. INTRODUCTION

It is a fact that new hardware is constantly being developed with more processing power and memory capacity. However, simpler systems are less expensive and this can be a very important factor in some application domains. In control systems for home automation, in particular, this can make a difference as it may allow a twofold benefit: On one side, it can help reduce the cost of simple systems and promote a more rapid expansion of domotics. On another side, it can offer an effective way to address new levels of automation involving a very big number of devices, allowing the implementation of complex systems with a rich set of integrated services and superior functionality.

Home automation offers many benefits and is slowly becoming more popular. It can ease daily tasks such as watering the garden, turning on outside lights at dusk, setting up the temperature and required illumination levels when we arrive from work, closing window blinds at certain times of the day, or turning off lights in vacant rooms. Home automation can contribute to better comfort levels and, at the same time, optimize the energy consumption, leading to savings in electricity, gas and water expenditures.

Home automation can also increase our safety and security, detecting and signaling emergency and intrusion situations. And it can be very helpful to elderly people and people with disabilities that may have difficulties in moving or executing certain tasks. For example, using a voice recognition device, it can be much easier to open the outside door, turning on the lights or turning off the oven.

Currently there are many technologies available for home automation. Some are standard or open (for example, X10 [1], EIB/Konnex [2], LonWorks [3] and CEBus [4]) and many are proprietary. These technologies are incompatible with each other and none dominates the market. Another big problem regards cost, because powerful technologies (such as EIB) tend to be very expensive, especially if we target big and complex systems. This is maintaining a door open to proprietary systems, which are more economical, but also contributes to disperse and confuse the market. Furthermore, new technologies are appearing (for example, ZigBee [5] - IEEE 802.15.4) which promise new and better characteristics, but do not contribute to

a technology convergence. In the current scenario, we think that more important than adhere to a particular standard may be to offer open solutions that support interoperation and can be easily integrated with other systems and technologies. In our opinion, an inexpensive platform with a flexible software architecture and communication capabilities fits well in this scenario.

With the current work we expect to offer a contribute to the development of applications on embedded systems with very little resources, offering a structured and portable approach that supports communication, which is vital in the connected world we live in. In the prototypes developed we used wired communication (which is cheap, simple and reliable). However, other communication media, such as, short-range RF and infrared, may also be used, as long as the corresponding hardware is employed and the communication software is adapted to the media particularities.

In this paper we start by describing the hardware architecture of the domotic modules we developed, identifying its core components and emphasizing its simplicity, which can be easily applied in other domains. In particular, we present some details regarding the interface with the communication medium.

Next we describe our software approach and compare it with other solutions. We give emphasis to the way applications are structured and to aspects regarding time managing, inter-task communication and implementation of our communication protocol. Finally, we present a technique that uses very little memory and assure a reliable communication between applications.

2. HARDWARE ARCHITECTURE

In this section we describe briefly the hardware architecture we used in our approach. As stated before, our main objective was to develop a platform as simple and economical as possible, but offering a good level of flexibility and being able to support networking. This last requirement is essential to offer the desired modularity and expansion capabilities, and also to support interoperation.

2.1 The Domotic Modules

In figure 1 we illustrate the main structuring blocks of our prototype, which has a core composed by just a microcontroller and an interface transceiver with the communication medium. Besides that, the Domotic Modules (DM) have specific hardware related to our application domain - home automation. That hardware includes interfaces to switches and various types of sensors (temperature, humidity, light intensity, power consumption, etc), and power electronics to command lights, motors, consumer equipment, etc. We may have different types of domotic modules, according to the kind of interface electronics deployed around a module's core.

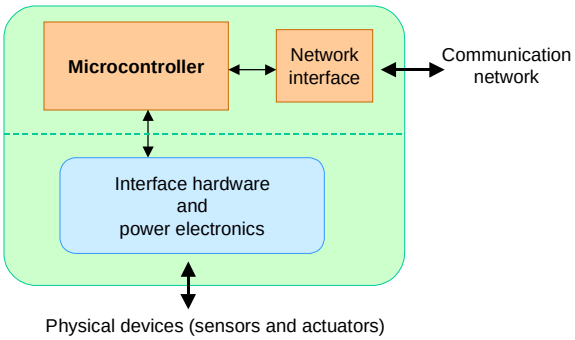


Figure 1. Architecture of the Domotic Modules

To make our solution even more cost-effective, each module may control several devices of the same type. For example, a single module may read four switches and command four lights. This distinguishes our approach from currently available products (where, typically, one module is associated with a single device) as it optimizes the use of common resources such as the microcontroller itself, power supply and the communication hardware. Although, at first, our proposal may seem less modular as it favors some concentration in each module, it makes sense in the context of super-automated homes where, in a given

proximity, there will be many home automation devices. For example, and considering just lighting, in a room we may expect to be able to control one ceiling light and two lamps, and read the corresponding switches. In a typical solution this would require as much as six modules (three to read the switches and three to command the light bulbs) while our approach needs just one module (with four switches and four lights) and even offers some space for future expansion.

2.2 Description of a Module's Core

We decided to base our modules on a simple 8-bit microcontroller with built-in memory for code and data, and typical peripherals such as timers, watchdog and input/output ports. Optionally, we may have other specific peripherals (e.g., analog-to-digital converters, PWM generators, SPI) that may be needed for the target application. However, and regarding the software architecture we propose (see section 3), we require only a timer to allow a precise measurement of time. Our prototypes of the domotic modules are based on the AT90S8515 microcontroller from ATMEL [6].

Concerning networking, we implemented a protocol using asynchronous serial communication. This option took into account the fact that most microcontrollers include at least one built-in UART (Universal Asynchronous Receiver Transmitter), simplifying the sending and receiving of bytes. Frames are composed by a set of bytes with a given format.

At the physical level we chose to use EIA-485 transceivers, because they offer good immunity to electric noise, support communication over long distances (covering easily a house or a residential building) and are inexpensive. Furthermore, this type of transceivers allows multi-point communication, being able to sustain conflicts when two or more drivers transmit at the same time. We implemented a protocol that accesses the communication medium using a CSMA/CD policy, which offers a low latency under normal traffic conditions. Details about the protocol and frame's format can be found in [7]. In section 3.5 we'll describe the main aspects of its implementation.

Different networking solutions may be implemented, which is independent of the multi-task software structure used. In other contexts, and for very short distances, I2C (Inter-Integrated Circuit protocol) could be an alternative and many microcontrollers include I2C hardware. More sophisticated options exist, for example CAN (Controller Area Network), which is available on some microcontrollers. But this is more expensive and makes the applications much more complex.

2.3 Access to the Communication Medium

Before concluding the present section we would like to mention an implementation detail regarding the access to the communication medium in the domotic modules. The adoption of a CSMA policy implies that the medium must be monitored for activity, and transmission can only start after a pre-defined period of silence (inactivity). To minimize collisions, that period of time is composed of a fixed duration plus a random small value. In this way, two nodes that intend to communicate will detect inactivity at the same time (at the end of the fixed period of silence) but the additional random value that they will wait will probably avoid a collision. For this mechanism to work it is vital to rapidly detect the start a new transmission. However, as the communication is based on sending and receiving bytes using an UART, network activity is only detected at the end of a complete byte (when the UART informs that a new byte was received), which will be too late. As we use asynchronous serial communication we should detect a start bit, which signals that a new byte will follow. In this way, other nodes would detect network activity and would refrain from transmitting. This reduces the period of potential conflict to the duration of a bit, instead of a byte (which is ten times bigger), greatly decreasing the probability of collision and enhancing the throughput of the network.

After some thinking we were able to implement a start-bit detection mechanism without any additional hardware. Our solution is illustrated in figure 2 and, basically, uses an external interrupt pin of the microcontroller. However, we do not allow an actual interrupt to occur, as this would generate too much workload to the microcontroller (one interrupt for each transition high to low in the serial line). Instead, we mask that interrupt source and simply test the corresponding flag bit whenever it is relevant. After detecting the fixed silence period, the interrupt flag is cleared and we enter the random period. At the end of this interval, the interrupt flag is tested. If it is active, some node initiated communication. Else, the line is inactive and we can start transmitting.

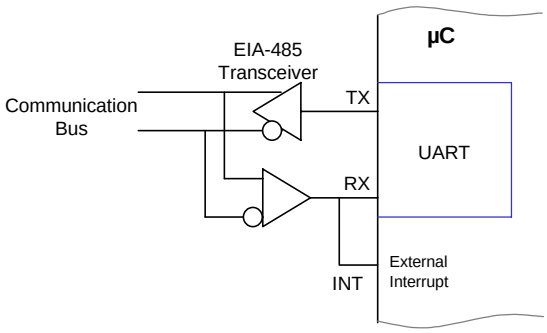


Figure 2. Interface with the communication medium

3. SOFTWARE ARCHITECTURE

In this section we describe the main aspects of the software approach we developed for the domotic modules, which are based on the AT90S8515 microcontroller from ATMEL. This microcontroller has 8 Kbytes of Flash memory for code and 512 bytes of data memory. Additionally it integrates several peripherals, particularly, two timers and one UART, which are relevant to our approach. More details can be found in [6].

3.1 Description of the Proposed Approach

We knew from the start that we were facing a challenge, as the available resources were quite limited and we wanted a flexible software solution to develop applications, have precise control over time and be able to communicate in a network. We looked at the literature and searched for real-time operating systems, kernels and executives that could satisfy our requirements and fit in our platform. However, most solutions are not meant to be applied to such a small platform and even if they can be used (for example, [8] and [9]), they consume excessive resources (memory and processing power) leaving few room to the development of applications. We found some very interesting approaches, such as the one described in [10] and [11], but they are particularly dedicated to sensor networks and put a special emphasis on reducing power consumption. Development is done in a language called *nesc* and using specific tools. We were unable to clearly identify the impact of this approach in memory and processing power. However, it is important to note that current devices that use this approach are based on the ATmega 128, which has a much bigger memory (128 Kbytes of Flash memory for code and 4 Kbytes of RAM).

Another proposal can be found in [12] and [13], which is clearly targeted at tiny platforms. As stated, it uses 500 to 1000 bytes of code for basic functions (without timing). This would be adequate to our platform but, after an analysis of the solution, we felt that the functionality offered was quite small and the programmer had to take care of some implementation details, namely the definition and management of the "virtual" positions of each task (points where processing is interrupted and later on resumed).

After some reflection, and pondering the numerous solutions analyzed, we decided to follow a different approach. Before describing our solution we start by defining the main aspects we consider relevant to our application domain:

- Preemption is not required (preemption is too onerous for our tiny systems).
- Opting for a co-operative environment means that many aspects regarding synchronization, data sharing, access to physical resources, etc, are greatly simplified as, at any given moment, only one task is running and has full control of the system. The price to pay for this simplicity is the need to manage the execution time of each task and adequately yield the CPU to allow other tasks to run. We think this is a small price to pay when we consider the advantages regarding implementation efficiency (for example, *semaphores* are not needed or they can be emulated with global variables).
- In our tiny systems there is no need for task priorities. The required functionality and the constraints that may exist are known in advance. So, an *a priori* solution can be studied and evaluated, to assure that real-time requirements that may exist are met.

- Regarding interrupts, we chose to use them only when indispensable, reserving its use for fulfilling hard real time constraints. This contributes to the determinism of the system, simplifies timing analysis and reduces problems with shared data and critical code sections.
- Communication is a key feature of our system. As described earlier, we use asynchronous serial communication, and we chose a bit-rate of 19200 bps. This rate offers a good throughput for our application domain, as messages are very short (around 10 bytes). It is important to note that, at the chosen bit-rate, the duration of a byte is approximately 520µs (the duration of a bit is 1/19200=52µs and each byte uses 10 bits: 8 data bits plus a start and a stop bit). Byte receiving and processing is done by a task which means that it will have to run at least once each 520µs. This represents an important real time requirement that acts as a bound to other tasks. From our experience in home automation, interaction with the real world (reading switches and sensors and commanding actuators) fits well in this bound as they usually are much less time demanding.
- Hard real time applications (for example, light dimming using a mains synchronized phase control) will be dealt individually and will most certainly use a dedicated timer and the corresponding interrupt routine.
- Tasks should be able to interact with each other exactly in the same way whether they are in the same module (internal communication) or in different modules (communication using the network). This offers an added modularity allowing to easily change the location of tasks.
- The system should offer a flexible means of managing time allowing a simultaneous account of multiple values, from the millisecond range up to seconds or even more.

After analyzing thoroughly our objectives and taking into account the very limited resources of our platform, we decided to adopt a software architecture where the programmer is exposed to all details regarding task scheduling and task switching. Only this approach can give the flexibility and efficiency required. To ease development and help the programmer in what regards the structuring and implementation of tasks, we developed task templates/skeletons. These offer a good start point and already include solutions to common needs such as waiting for an event, waiting for a timer to expire and sending a message.

In our approach tasks are blocks of code that are called cyclically, using a Round-Robin policy. In practice this is implemented by an infinite loop where tasks are called one after another (if needed, a given task may be called more than once per cycle). Tasks are implemented as state machines (explained later on). A task must never block and should execute its actions as fast as possible to free the processor and allow other tasks to run. If an action is lengthy, it has to be decomposed into smaller parts that are executed one at a time. Due to the determinism of the solution it is simple to analyze each part of a task (state) and evaluate worst case execution time and, consequently, determine if deadlines will be met.

Although our approach requires some additional care from the programmer, the model is simple to understand and, after some practice, it becomes very easy to develop tasks that follow this logic. The technique is very efficient and allows an optimization of the available resources.

3.2 Managing time

Time is a very important issue in most embedded systems. We adopted a solution where a task monitors precisely the passing of time (using a hardware timer and associated interrupt), and then offers a timing service to other tasks. That service is based on the use of global variables, each representing one timer. In each task cycle those variables are updated reflecting the precise amount of time elapsed (accumulated errors are avoided). From the task's point of view there is an inherent error associated with this mechanism which, however, is less than the duration of a task cycle (less than 500µs in our prototypes). This error, which does not accumulate, is perfectly acceptable for most tasks. If a task really needs a more precise control of time it will have to use a hardware timer and associated interrupt.

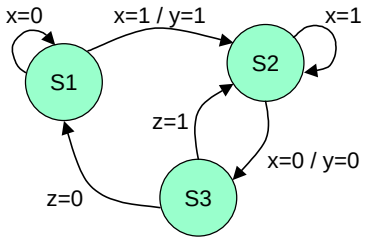
The time service task allows accounting of different amounts of time, which can be easily configured. In our prototypes of domotic modules we used the following granularities: 10ms, 100ms and 1s.

3.3 Implementing complex tasks

In our model, a task is a piece of code that is called cyclically and must execute its actions in a controlled amount of time in order to allow other tasks to run. In spite of this simple model, it is possible to implement

very complex tasks. This can be achieved using state machines. In figure 3 we illustrate a simple state machine and a fragment of code that implements it. Every time the code is called it evaluates the conditions associated with its current state and it may perform actions and advance to another state.

Long actions should be divided into shorter ones and executed in several states.




```
switch (TaskState)  {
   case StateS1:
      if(x==1)  { y = 1; TaskState = StateS2;} break;
   case StateS2:
      if(x==0)  { y = 0; TaskState = StateS3;} break;
   case StateS3:
      if(z==1)  { TaskState = StateS2;}
      else   { TaskState = StateS1;} break;
}
```

Figure 3. Implementing a state machine

3.4 Inter-Task Communication

Tasks can exchange data using global variables. However, this creates dependencies between tasks that may be undesirable. Another approach, more generic, consists in using messages, which offers a uniform way of interaction, whether tasks are in the same module (internal communication) or in different modules (communication using the network). To support the exchange of messages each task may have one mailbox. This mailbox is used both for sending and receiving and can hold just one message. This option was taken to save memory. Mailboxes are managed by the network task, which we describe next.

3.5 The Network Task

As mentioned in sub-sections 2.2 and 2.3, a communication protocol was developed that offers a data link service to applications. The access to the transmission medium is done using a variant of CSMA/CD. The protocol is implemented by a task, named NET, which has nine states:

- *NET_STATE_IDLE* - This is the idle state, where we wait for the arrival of new frames (if we detect a new frame we go to *NET_STATE_RECEIVE_FIRST_BYTE*). In this state we also monitor each application's mailbox for frames to be sent to the network and account for silent periods to know when a new transmission may be initiated (transmission is done in *NET_STATE_ SENDING*).
- *NET_STATE_RECEIVE_FIRST_BYTE* - We jump to this state when we detect the start of a transmission. As explained in sub-section 2.3, that occurs when we detect the start-bit of the first byte of a frame. In this state we wait for the complete byte to arrive, to read it from the UART. It is during this period of time that a collision is more probable which can affect the data received. If the UART fails to receive a byte or signals a receive error, we go back to *NET_STATE_IDLE*. If a valid byte is received we go to *NET_STATE_RECEIVING*.
- *NET_STATE_RECEIVING* - In this state we receive the second and following bytes of a frame, which ends with a CRC (Cyclic Redundancy Check) to allow detection of communication errors. This state distinguishes the reception of a frame's header from the rest of the frame. The header includes the address of the destination application and it is read to an internal buffer (for details about a frame's format see [7]). The address is analyzed to see if the application exists, if its mailbox is empty and if it has enough space to hold the frame. If affirmative, the rest of the frame will be read directly into the application's mailbox, releasing the NET task from having a temporary storage for an entire frame and, thus, saving an important amount of memory. If the frame cannot be received, a jamming byte is sent to produce a collision and abort current transmission. This optimizes the use of the network bandwidth.
- *NET_STATE_SEND_ACK* - When a frame is correctly received an acknowledgement byte is sent immediately. This is a particularity of our protocol that enhances its throughput as it avoids the

usage of a complete frame just for acknowledgement. After this state we go to *NET_STATE_WAIT_TX_COMPLETE*.

- *NET_STATE_WAIT_TX_COMPLETE* - Here we wait for the current byte to complete transmission. This is needed as transmission is done by the UART and takes a significant amount of time. Only then the EIA-485 driver can be disabled. Next, we jump to *NET_STATE_IDLE*.
- *NET_STATE_SENDING* - In this state the contents of an application's mailbox is sent to the network, byte by byte. A CRC is computed and sent at the end of the frame. During sending we also receive our own data (see figure 2), which is used to detect a collision. After correct reception of last byte sent, a frame transmission completes and we go to *NET_STATE_WAIT_ACK*. In case of a collision we jump to *NET_STATE_TEST_SEND_AGAIN*.
- *NET_STATE_WAIT_ACK* - After transmitting a frame we wait for an acknowledgement byte. If it is received, that signals a transmission success. Consequently, the application mailbox is cleared and we jump to the idle state. If we receive a not-acknowledge byte (or a timeout occurs) we go to *NET_STATE_TEST_SEND_AGAIN*.
- *NET_STATE_TEST_SEND_AGAIN* - In this state we know that a transmission did not succeed (note that a frame may have been delivered but the acknowledgement was lost). To improve the communication reliability we re-transmit a frame if we do not receive an acknowledgement. This state controls the number of times a given frame is re-transmitted and, when a predefined number is reached, the frame is discarded. From this state we evolve to *NET_STATE_IDLE*.
- *NET_STATE_SEND_NACK* - Sends a not-acknowledge byte to indicate that the received frame had some problem (e.g., an invalid CRC) and jumps to *NET_STATE_WAIT_TX_ COMPLETE*.

From the above description we see that the network task is fairly complex. Additionally, it also supports broadcast frames. These frames are not acknowledged and, to optimize its processing, they are delivered to a pre-defined task. This task may process the frame or forward it to other tasks.

Frames may have two priority levels - normal and high. In the case of high priority frames the network task waits a shorter period of silence before accessing the communication medium, guarantying that they are sent before the normal frames.

The NET task also handles communication within a node. When it analyzes the contents of a mailbox it looks at the destination address and, if it refers to the current module, the message is copied directly to the destination mailbox.

In spite of its complexity and all the functionality offered, our implementation of the network task uses only 1732 bytes of code and 54 bytes of static variables. Most of these variables regard data structures related to the management of the available mailboxes, their maximum lengths and status flags (*empty*, *full*, *busy*).

3.6 Reliable Communication

The retransmission mechanism that exists at the link layer is important to overcome frame losses that can occur due to collisions, because the destination mailbox is full or another reason. However, that mechanism does not assure that the messages are actually delivered as retransmission is canceled after some attempts. A way of guaranteeing that a message really reaches its destination (at least once) is to implement an acknowledgement and retransmission mechanism at the application level.

This type of behavior is usually accomplished using a buffer that stores the messages that were sent. When an acknowledgment is received, the corresponding message is removed from the buffer. Messages not acknowledged are retransmitted after some time. Unfortunately, the available memory is very small and we were forced to devise another solution.

Our solution uses a data structure to signal that an event has occurred and that a corresponding message must be sent. In practice that data structure is simply a byte array where the index identifies the event (and corresponding message) and the byte's content specifies the following information (at the bit level):

- Send immediately (bit 7);
- Fast repeat counter (bits 6 and 5; valid values between 0 and 3);
- Slow repeat counter (bits 4 to 0; valid values between 0 and 31; 31 implies repeat forever).

When an application detects some occurrence (for example, temperature above a predefined value) it activates the corresponding event, setting the "send immediately" bit and specifying values for the repeat

counters. As a consequence, the adequate message is generated and sent immediately. If an acknowledgment is received the event is cleared. Else, the message is generated again after some time and re-sent (and the counter is decremented). There are two types of counters, which have different amounts of time associated with them. With our solution we may specify up to three fast repeats, i.e., repeats with a small interval between them. After that, further sending will occur with a long interval in between. This greatly reduces the network traffic in the event of a loss of connectivity with a module or a network segment, while offering a "keep trying" mechanism. When the counters reach zero, the event becomes inactive and no more messages are sent. As a special case, if we specify 31 (all bits set) for the slow repeat counter, the message repeats forever if an acknowledgment is not received.

4. CONCLUSION

In this paper we presented a software approach that allows an efficient implementation of multiple applications in embedded systems with very limited resources. Furthermore our approach supports communication, allowing interconnection in a network and an easy interaction with other systems. The communication model offered is very simple and uses very little memory, allowing tasks to seamless communicate within a module or between different modules.

We described the key aspects of the proposed software architecture, which allows applications to be structured into tasks, using state machines, and offers a multi-task co-operative environment with real-time characteristics. We described also a task that offers timing services to the system and detailed the task that implements the network and inter-task communication. In our prototypes, the timing task uses 196 bytes of code and 4 bytes of static data, and the network task, which is quite complex, uses 1732 bytes of code and 54 bytes of static data. Our executive, for a configuration with six tasks, uses just 56 bytes of code and no data (this includes initialization of the microcontroller, calling initialization routines for each task and the cyclic loop where tasks are invoked one after another). From our experience a fairly complex task can be implemented using less than 1 Kbytes of code.

Finally we stress that development was done entirely in C, being extremely portable. In fact, the platform specific code is minimal and regards essentially the initialisation of a timer and associated interrupt routine, and access to the built-in UART for external communication. Our approach is, thus, generic and may be easily applied to other embedded platforms with very limited resources.

REFERENCES

[1] X10, http://www.x10.com

[2] KNX, Konnex Association, http://www.konnex.org

[3] LonWorks (ANSI/EIA 709), http://www.echelon.com

[4] CEBus - Consumer Electronics Bus (EIA-600), http://www.cebus.org

[5] ZigBee wireless technology (IEEE 802.15.4), http://www.zigbee.org

[6] Atmel AVR 8-Bit RISC processor, http://www.atmel.com/products/AVR/

[7] Renato Nunes, 2004, A Communication Infrastructure for Home Automation. *CCCT 2004 - International Conference on Computer, Communications and Control Technologies*, Austin, USA.

[8] Jean J. Labrosse, 2002. *MicroC/OS-II The Real Time Kernel*, CMP books, Lawrence, Kansas, 2nd edition.

[9] FreeRTOS, http://www.freertos.org/

[10] Jason Hill, et al, 2000, System Architecture Directions for Network Sensors. *ASPLOS - International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge.

[11] David E. Culler, et al, 2001, A Network-Centric Approach to Embedded Software for Tiny Devices. *EMSOFT 2001 - International Workshop on Embedded Software*, Tahoe City, USA.

[12] Hugues Smeets, 2004, A Tiny Portable Real Time Stackless Kernel. *IADIS International Conference on Applied Computing 2004*, Lisbon, Portugal, March 2004.

[13] Huges Smeets, 2001. *Conception d'une plateforme "temps réel" enfouie*, Travail de fin d'études, Université de Liège, http://www.montefiore.ulg.ac.be/~smeets/tfe.html.

[14] David Simon, 1999. *An Embedded Software Primer*, Addison-Wesley.