DOI. 10.1109/ROSE.2009.4669174

ROSE 2008 – IEEE International Workshop
on Robotic and Sensors Environments
Ottawa – Canada, 17-18 October 2008

Inter-Task Communication and Synchronization in the Hard Real-Time Compact Kernel HARETICK

Mihai V. Micea[1], Cristina Certejan[1], Valentin Stangaciu[1], Razvan Cioarga[1], Vladimir Cretu[1], Emil Petriu[2]
[1] Department of Computer and Software Engineering (DCSE), "Politehnica" University of Timisoara, 2, Vasile Parvan Blvd., 300223 – Timisoara, Romania, mihai.micea@cs.upt.ro, certejan@gmail.com, stangaciu@gmail.com, razvan.cioarga@cs.upt.ro, vladimir.cretu@cs.upt.ro
[2] School of Information Technology and Eng. (SITE), University of Ottawa, 800, King Edward, Ottawa, KIN 6N5, Canada, petriu@site.uottawa.ca

Abstract *– HARETICK is a hard real-time compact operating kernel designed specifically to support critical applications on DSP and embedded platforms including intelligent sensor networks and robotic environments. It provides operating support for both hard real-time and soft/non real-time tasks. The hard real-time task execution context is based on non-preemptive mechanisms. This paper focuses on the inter-task communication and synchronization techniques involving the two types of tasks previously mentioned. As a case study, a highly predictable synchronous serial communication (i.e., SPI) interface implemented on an ARM7-based HARETICK platform, is presented and discussed, along with some of the most interesting experimental results.*

Keywords *– Inter-process communication, synchronization, hard real-time, HARETICK.*

I. INTRODUCTION

Many modern sensing, measurement and embedded control environments require data acquisition and processing in a strict timely fashion, imposing hard deadlines to the system response to events [1]–[5]. As a result, specification, design and implementation of hard real-time (HRT) operating kernels for embedded systems is a topic of extremely high interest in the academic and industrial communities.

Such an HRT operating kernel, named HARETICK (Hard Real-Time Compact Kernel) [6] is being developed at the Digital Signal Processing Laboratories (DSPLabs), "Politehnica" University of Timisoara, since 2000, on various DSP and embedded platforms, including Motorola DSP56303EVM [7], Motorola MSC8101ADS [8] and, recently, the ARM7-based Philips LPC2xxx processors [9].

HARETICK is a single-user, multitasking, hybrid real-time operating kernel [10], which provides support for two concurrent task execution environments: the HRT context, for the execution of hard real-time tasks in a non-preemptive manner, and the SRT context, for the execution of soft real-time (or regular) tasks in a classical, preemptive and priority-based manner. The HRT context has precedence at any time over the SRT context. Therefore, HARETICK is able to guarantee that all the tasks scheduled and executed within the HRT context will meet all their temporal specifications, even in the worst case operating conditions [11]. As a consequence, HARETICK currently allows only one interrupt source: the *Real Time Clock* (*RTC*).

This paper focuses on a very important aspect of the HARETICK kernel operation: inter-task communication and synchronization (ITCS), with special emphasis on the mechanisms developed to support the ITCS between the non-preemptive HRT tasks and the preemptive SRT tasks.

II. HARD AND SOFT REAL-TIME TASKS IN HARETICK

A generic application developed for the HARETICK platform has three types of tasks, interconnected by control and/or data precedence relations: SRT tasks, HRT tasks and an initialization task.

The SRT tasks are modeled, scheduled and executed in a similar manner as in classical (non real-time) operating systems. As a particular feature, the SRT tasks have non-restricted access to the application's global data zone, along with any other type of application task. In our discussion here, we denote the *j*-th SRT task in the application by L_j.

The initialization task is the first task to be launched, at the application startup, and is responsible for the initialization of the application data structures, including the output parameters of all the HRT tasks of the application. The motivation for introducing such a task will appear more clearly in the next section of the paper. From the execution point of view, the initialization task runs within the SRT context.

A *ModX* (*Executable Module*) is defined as a periodic, modular, HRT task, with complete and strict temporal specifications, scheduled and executed in non-preemptive context:

$$M_i \equiv \langle \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S}, \boldsymbol{F} \rangle \quad (1)$$

where: $\boldsymbol{P} = \{\boldsymbol{P_{IN}}, \boldsymbol{P_{OUT}}, \boldsymbol{P_{GLB}}\}$ is the set of input, output and global parameters of M_i, respectively; $\boldsymbol{S} = \{\boldsymbol{S_{IN}}, \boldsymbol{S_{OUT}}\}$ is the set of input and output signals which M_i interacts with; $\boldsymbol{F}$ is the task instruction code (its functional specification); and:

$$\boldsymbol{T} = \left\{T_{pr}^{M_i}, T_{ex}^{M_i}, T_{dl}^{M_i}, T_{dy}^{M_i}, N^{M_i}\right\} \quad (2)$$

represents the set of temporal parameters of M_i, in their respective order: period, execution time, deadline, delay of execution during each period, and execution count.

The non-preemptive approach of modeling each HRT task of an application as a ModX, requires actual values (in system time units) for a minimum of temporal parameters, such as period, execution time and execution count. These values are set or calculated during the application specification and analysis phases, and will be verified during the validation phase. These phases must be performed prior to the actual scheduling and execution of the application, in order to ensure maximum predictability. The execution time is considered to be a constant value during the entire task operation, and equal to the task *WCET* (*Worst Case Execution Time*) which results from the offline program timing analysis. The modular approach of modeling the HRT task as the ModX, enables automatic techniques of WCET estimation during application analysis.

As asynchronous mechanisms have been eliminated from the model, input signals are processed by their corresponding ModXs by *periodic polling* techniques [12].

While a ModX M_i, once loaded on the target platform, is scheduled for as long as the application is running, its execution count parameter, N^{M_i}, specifies three possibilities for the effective execution of M_i: a) continuous execution ($N^{M_i} = \infty$), without decrementing the execution count; b) normal execution ($0 < N^{M_i} < \infty$), with decrementing of the execution count; c) non-execution ($N^{M_i} = 0$): M_i is not executed, although currently scheduled. The ModXs reaching this latter status are called *Ghost ModXs*. An interesting feature of the ModX model is that its execution count (and, therefore, its effective execution) can be controlled (changed) at runtime by other ModXs or even by itself.

III. INTER-TASK COMMUNICATION AND SYNCHRONIZATION IN HARETICK

In this section we will discuss the ITCS mechanisms developed for the HARETICK kernel.

For exemplification, consider a particular data processing application running on a target platform with HARETICK support and involving also data exchanges over a communication interface with another host. The host provides a frame of a certain word length of input data to the processing target, which applies some filter and then sends the results back to the master host. Then, the procedure can be reinitiated by the host with a fresh new frame of input data. Fig. 1 depicts the specification of this application, which has been designed for the HARETICK kernel. The communication interface (considered a synchronous interface in our example, without loosing the generality of the discussion) is handled by three HRT context ModXs: `M_Comm_LVL1_RxTx` handles the synchronous reception and transmission of a word of data (Level 1 protocol), while `M_Comm_LVL2_Rx` and `M_Comm_LVL2_Tx` handle the reception and, respectively, the transmission of an entire frame (Level 2 protocols), using finite-state automata for frame encapsulation and processing.

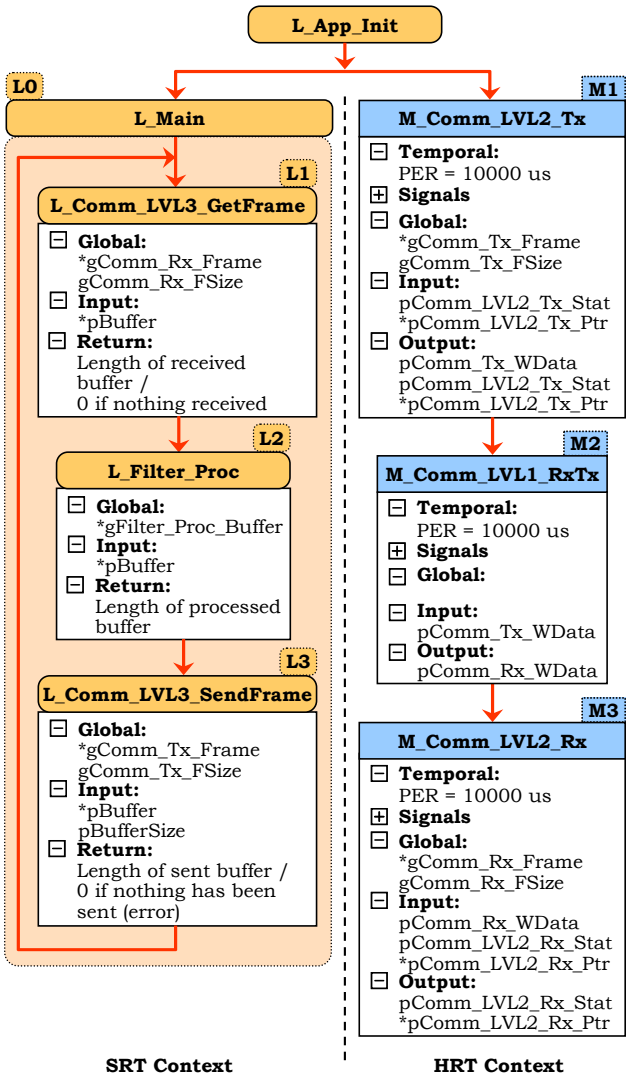

Fig. 1. Example Data Communication and Processing Application.

Within the SRT context, the actual data processing has been designed using the following tasks:

- `L_Comm_LVL3_GetFrame`: non-blocking task which copies the received frame data from the Level 2 buffer to another buffer, referred by the input parameter. This task implements the Level 3 (application level) protocol of the communication interface;
- `L_Filter_Proc`: task which processes in-place the data provided in a buffer referred by the input parameter;
- `L_Comm_LVL3_SendFrame`: task which copies data from a buffer specified by the input parameters (base pointer + size) to the Level 2 transmission frame buffer. This task is also a part of the Level 3 protocols of our example communication interface.

Our discussion will overlook the ITCS at the SRT context level, as the SRT tasks are modeled and implemented in a traditional manner.

A. ITCS at the HRT Level

ModXs represent sequences of application code, similar to the "basic blocks" from the compilers theory, which are scheduled and executed periodically, without blocking and without being interrupted. As a result, ModXs implement atomic operations, thus eliminating the need of synchronization mechanisms and of controlling the concurrent access to shared resources.

Information exchange between the application ModXs is performed through the input, output and global parameters, which define the set ***P*** (see Eq. (1)). The global parameters and all the output parameters of each ModX are implemented as persistent data structures in the HARETICK kernel in the sense that they keep their value over time, until updated. The global parameters can be modified by any application task (ModXs or SRT tasks), while the output parameters can be modified only by their own ModX. Specific memory resources are allocated within the kernel to support these types of parameters. The data and program memory layout of the kernel, detailing the previously mentioned structures to accommodate the application ModX operation and ITCS for our example, are presented in Fig. 2.

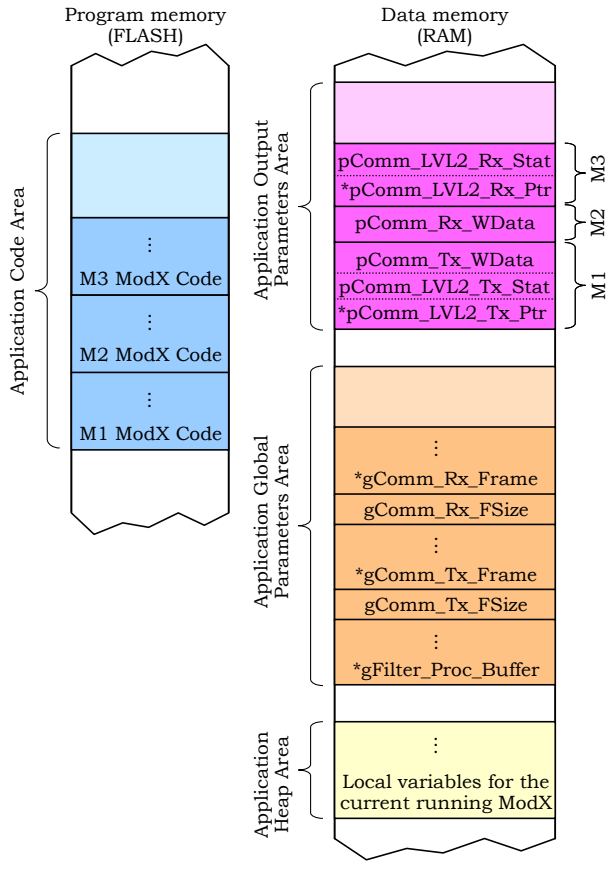

Fig. 2. Application ModX Data and Program Memory Maps in the HARETICK Kernel.

Consider, for example, the execution of the ModX `M_Comm_LVL1_RxTx` (denoted as `M2` in Fig. 1), in the process of data transmission over the communication interface. The control dependence of the application specifies that `M2` will be cyclically scheduled and run after the `M_Comm_LVL2_Tx` ModX (denoted as `M1`), both with a 10000 μs period. `M2` has an input parameter, `pComm_Tx_WData`, to get a new word of data from the transmit frame built by `M1` and to send it over the communication interface, on each execution cycle. At launch, the value of this input parameter is copied by `M2` directly from the corresponding memory location of the output parameters area of ModX `M1`. If `M2` receives a new word from the communication link, the value of the `pComm_Rx_WData` output parameter will be updated accordingly, to be used by the ModX `M3`.

To accommodate the local variables of the ModXs, HARETICK provides a unique, volatile memory area, called *Application Heap*, which will be used, in turns, by each ModX during its runtime, taking into account that all ModXs are non-interruptible tasks. As seen, the HRT context implemented by HARETICK does not use any stack-based mechanisms for passing parameters or for local/volatile variables.

An interesting aspect, resulting from the ModX being a periodic task, is the facility of transmitting current status information from one execution instance to another execution instance of the same ModX. This is accomplished by specifying the same parameter as both input and output of a particular ModX. Such *circular parameters* in our example are, in the case of ModX `M1`, `pComm_LVL2_Tx_Stat`, for keeping track of the current status of the transmission frame construction using a finite-state automaton, and `pComm_LVL2_Tx_Ptr`, for managing the position of the currently processed word within the frame. This aspect also stresses the importance of the application initialization task, `L_App_Init`, executed prior to all other application ModXs, with the main goal of setting initial values for such parameters as described above.

B. HRT-SRT Tasks Communication and Synchronization

Two major aspects of the HARETICK kernel prevail when it comes to accommodate the ITCS between a ModX and a SRT task: (a) ModXs are non-preemptive tasks, and (b) the HRT context has precedence over the SRT context, meaning that any ModX can interrupt any SRT task at any moment. Therefore, classical inter-task communication and synchronization mechanisms, such as flags/semaphores, conditional critical regions, monitors, mutexes, messages and so on [14]–[17], do not apply to our case.

The basic mechanism we designed for the ITCS between a ModX and an SRT task under HARETICK consists of *guarded* (or flagged) *buffers*, and treats the information exchange between the two tasks as a master-slave

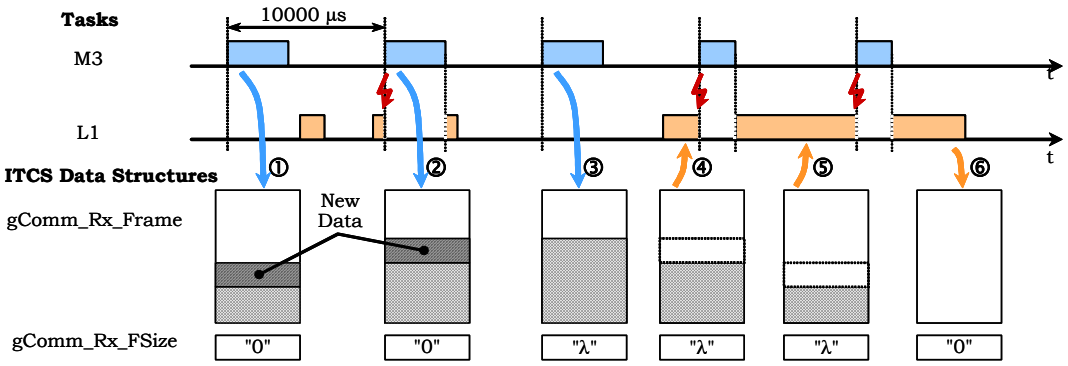

Fig. 3. Exemplification of HRT–SRT Inter-Task Communication and Synchronization.

communication, with the ModX being the master. This means that the ModX will manage the information exchange.

To exemplify these mechanisms, consider the reception of a frame of data to be further processed, from the communication interface, implemented with the application introduced in Section III (see Fig. 1 and Fig. 2). The data structures used by the kernel/application and the operating principles of the ITCS between the `M_Comm_LVL2_Rx` ModX (denoted as `M3`) and the `L_Comm_LVL3_GetFrame` SRT task (denoted as `L1`) are depicted in Fig. 3.

The specific data structures used here consist of the `gComm_Rx_Frame` buffer and of the `gComm_Rx_FSize` variable, as the access guard for the buffer. Both structures are declared globally at the application level, thus being equally available to ModXs and SRT tasks. Instead of using a specific flag as guard, we chose the `gComm_Rx_FSize` parameter, under the following rule: when it is zero (empty frame buffer), ModX `M3` has exclusive access over the ITCS data; on the other hand, any other non-zero value indicates a frame of data ready to be accessed exclusively by the SRT task `L1`.

The inter-task communication and synchronization principles can be summarized by the following six steps, shown in Fig. 3:

(1) ModX `M3` receives a new word of data from the lower level (`M2`), and inserts it into the buffer (after the corresponding decapsulation and interpretation operations, implemented by its internal state automata). The `gComm_Rx_FSize` guard value remains zero. The SRT task `L1` checks the guard for a non-zero value, and, in this case, the task finishes its execution (returning 0 – "reception buffer empty or not ready").

(2) The same case as above. ModX `M3` interrupts the SRT task `L1`, each time the former is scheduled for execution. New words of received data are inserted into the buffer. The guard is still zero, indicating the ModX has exclusive access to the ITCS structures.

(3) `M3` encounters an EOF (End of Frame) -type word at reception, calculates the final length of the received frame (λ in our case) and writes this value into the guard parameter, thus granting exclusive access for the `L1` task to the data structures.

(4) During its next launch, `L1` verifies the non-zero value of the guard and starts its loop to copy the received frame into the buffer referred by the input parameter (in our case, the filter processing buffer, `gFilter_Proc_Buffer`). At this step, Fig. 3 depicts also the execution of ModX `M3`, which interrupts the task `L1`, checks the guard and leaves the frame buffer unchanged, recognizing the exclusive access of `L1` over the ITCS data structures. Therefore, this execution instance of `M3` will be shorter than the previous ones.

(5) `L1` continues accessing the received frame buffer, while being interrupted by `M3` at its scheduled time instances.

(6) When `L1` finishes the copying process, it sets the value of the buffer guard back to zero (`gComm_Rx_FSize = 0`), thus releasing its lock on the ITCS structures. The operation of resetting the guard value will be one of the final instructions of the `L1` code, as it is equivalent to the *commit* operation of a shared database access transaction.

IV. CASE STUDY: A PREDICTABLE SPI COMMUNICATION INTERFACE

An application similar to the one exemplified in the previous sections has been developed and tested on several ARM7-based Philips LPC2xxx platforms [9], within the *CORE-TX* project [13] being developed at the DSPLabs. CORE-TX is designed as a complex platform for the development and analysis of collaborative robotic environments and intelligent wireless sensor networks. The nodes of the CORE-TX system, called WITs (*Wireless Intelligent Terminals*), have a modular architecture based on a motherboard and several daughter boards, all interconnected through a synchronous serial peripheral interface (SPI). While the motherboard has the main role of central processing and control element, the daughter boards perform functions specific to wireless sensors and robots: data acquisition, wireless communication, robotic movement, power supply and management, etc. Such functions require in most cases hard real-time behavior of the underlying

platforms, and, therefore, on each daughter board of the WIT runs the HARETICK kernel.

We have implemented and tested two versions of SPI communication between the motherboard and a daughter board: a half-duplex and a full duplex SPI version. In both versions, the motherboard is the master, generating the serial synchronous bit clock and cyclically polling the daughter boards as slaves for command and data exchange. Table 1 presents the implementation parameters of the half-duplex SPI application.

Some of the most interesting measurement results, acquired with a high performance digital oscilloscope (Tektronix TDS220), are shown in Fig. 4 and Fig. 5.

Fig. 4 captures the operation of the two execution contexts of the HARETICK kernel with respect to the SPI reception part: Channel 1 (top) depicts the Layer 3 SPI reception SRT task, while two consecutive executions of the reception ModX appear on Channel 2. The second execution of the ModX is shortly followed by an execution of the HRT context scheduler, HSCD. The measurement has been configured at a time base of 500 μs, thus revealing also the period of the reception ModX: ~ 3333 μs, as specified at the application design phase (see Table 1). The HSCD system ModX operates also within the kernel HRT context, along with the application ModXs.

Table 1. Implementation Parameters of the SPI Application under the HARETICK Kernel.

Parameter	Value
Synchronous serial bit clock frequency	1.8 MHz
Feasible SPI bitrate achieved	800 bps
Period: Reception ModX	3333.33 μs
Period: Transmission ModX	10000 μs
Execution time: HRT Scheduler (HSCD) + Executive	224 μs
Execution time: Reception ModX + Executive	14 μs
Code size: Reception ModX	12940 Bytes
Code size: Transmission ModX	12656 Bytes
Data size: Reception ModX	14176 Bytes
Data size: Transmission ModX	13964 Bytes

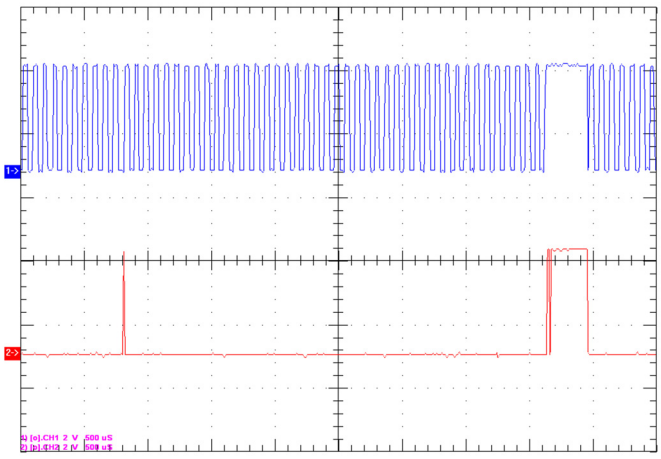

Fig. 4. Measurement of the SPI Reception: Layer 3 SRT Task – Top, Versus the Reception ModX and the HSCD Scheduler – Bottom.

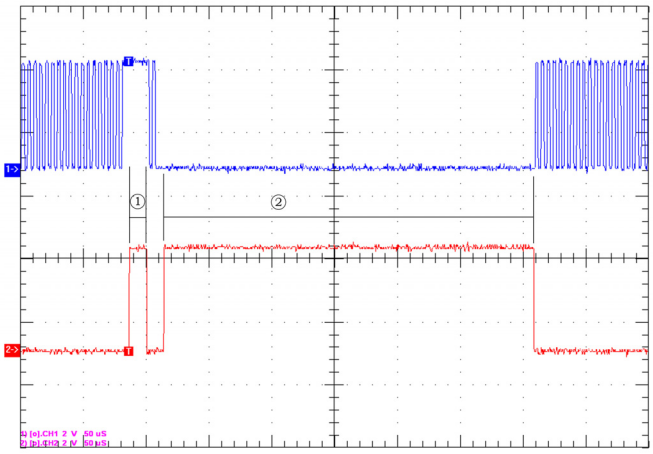

Fig. 5. Detailed View of the SRT Task Execution – Top, Versus An Execution of the Reception ModX Followed by the HSCD – Bottom.

Fig. 5 captures in more details the operation of the reception SRT task (Channel 1, top) while being interrupted by an execution of the reception ModX (interval ①, Channel 2, bottom) and then by the execution of the HSCD ModX (interval ②, Channel 2, bottom). The measurement has been configured at a time base ten times smaller than in the case of the previous capture: 50 μs.

To capture and measure the operation of the desired SRT tasks, their code has been modified to toggle an output line of the target platform each time they will be launched. To track the execution of the HRT context, the system executive (HDIS) will assert another output line before launching a ModX and, after the ModX finishes, the executive suffix de-asserts the line.

V. CONCLUSIONS

This paper presents and discusses the problem of inter-task communication and synchronization for the hard real-time operating kernel HARETICK, which provides execution support for both traditional SRT (or non real-time) tasks, as well as for periodic, non-preemptive, HRT tasks, modeled as ModXs.

A set of extensive feasibility and performance tests have been conducted on the implementation of a specific communication application – the SPI link within a CORE-TX wireless intelligent terminal (WIT). The test and measurement results obtained confirm the behavior of the system to be both correct and highly predictable, as specified at design phase. The proposed ITCS mechanisms – persistent data structures for ModX output parameters and guarded buffers for HRT-SRT data exchange – proved to be feasible as well as efficient. On the other hand, the memory-related costs imposed to the target platform are relatively high.

Future work will focus on extending and testing the SPI communication to connect several daughter boards and the motherboard of a WIT, as well as to optimize the system, especially to increase its bit rate over the SPI link.

VI. RELATED WORK

Inter-task synchronization is an important aspect of an operating system, and it becomes a key feasibility issue in the case of real-time behavior requirements. Most real-time operating systems and kernels implement some types of mechanisms to solve the ITCS [14]–[17]: flags, semaphores (POSIX, Ada95), conditional critical regions, monitors (Modula-1, Mesa, POSIX, RT Java), mutexes and conditional variables (POSIX), protected objects (Ada95), messages and mailboxes (POSIX, CHILL, Occam2, Ada95), priority inheritance and priority ceiling protocols, etc. They all have specific problems (such as deadlocks, livelocks, priority inversion, and so on) and they also introduce some degree of non-determinism into the system.

The operating principles of the HARETICK kernel and, to some extent, the model of its HRT tasks, feature many similarities with time-triggered systems such as the MARS system [18], the time-triggered architecture (TTA) [19] and the Giotto methodology [20], [21]. Giotto specifies the information exchange between tasks is carried out through their "ports", by "guarded drivers" (atomic, non-preemptive unit of computation). Based on the synchronous reactive programming principles and to ensure the predictability of their ITCS mechanisms, these systems force the inter-task communication to be scheduled along with the tasks, thus burdening the scheduling schemes. On the other hand, in HARETICK, the ITCS occurs as a natural and direct process, managed by the non-preemptive HRT tasks.

ACKNOWLEDGEMENTS

This work has been supported in parts by the following grants from the Romanian Ministry of Education, the National University Research Council, and the National Authority for Scientific Research (ANCS): CNCSIS-A_717/2005-2007 and CEEX-ET-07/2006-2008. Currently, this work is supported in parts by the PNCDI_II-ID_22/2007-2010 grant.

REFERENCES

[1] J. A. Stankovic, T. F. Abdelzaher, C. Lu, L. Sha, J. C. Hou, "Real-Time Communication and Coordination in Embedded Sensor Networks", *Proc. IEEE*, **7** (91), pp. (1002–1022), Jul. 2003.

[2] J. Hill, M. Horton, R. Kling, L. Krishnamurthy, "The Platforms Enabling Wireless Sensor Networks", *Commun. ACM*, **6** (47), pp. (41–46), Jun. 2004.

[3] V. Cretu, T. Jurca, M. V. Micea, I. Sora, "Instrumentation and measurement in Romania: Technical developments at 'Politehnica' University of Timisoara," *IEEE Instrum. Meas. Mag.*, **3** (6), pp. (41–47), Sep. 2003.

[4] M. V. Micea, M. Stratulat, D. Ardelean, D. Aioanei, "Implementing Professional Audio Effects with DSPs", *Trans. Autom. Control Comput. Sci.*, **60** (46), Ed. "Politehnica" Timisoara, pp. (55–60), 2001.

[5] M. V. Micea, L. Muntean, D. Brosteanu, "Embedded Techniques for Autonomous Robot Orientation", in *Proc. 6th Intl. Conf. Developm. and Applic. Syst., DAS 2002*, Suceava, Romania, 2002, pp. (22–27).

[6] M. V. Micea, V. Cretu, "Highly Predictable Execution Support for Critical Applications with HARETICK Kernel", *Intl. J. Electronics Communic. (AEUE)*, **5** (59), Elsevier, pp. (278–287), 2005.

[7] Motorola, Inc., "DSP56307EVM User's Manual", Rev. 3/1999, DSP56307EVMUM/D, Semiconductor Products Sector, DSP Division, Austin, USA 1999.

[8] Motorola, Inc., "MSC8101 Reference Manual: 16-Bit Digital Signal Processor", MSC8101RM/D, Rev. 1, Jun. 2001.

[9] Philips Semiconductors, "LPC2119/2129/2194/2292/2294 User Manual", Koninklijke Philips Electronics N.V., May 2004.

[10] M. V. Micea, "HARETICK: A Real-Time Compact Kernel for Critical Applications on Embedded Platforms", in *Proc. 7th Intl. Conf. Development and Applic. Syst., DAS2004*, Suceava, Romania, May 2004, pp. (16–23).

[11] M. V. Micea, V. Cretu, V. Groza, "Maximum Predictability in Signal Interactions with HARETICK Kernel", *IEEE Trans. Instrum. Meas.*, **4** (55), pp. (1317–1330), Aug. 2006.

[12] M. V. Micea, V. Cretu, L. M. Patcas, "Program Modeling and Analysis of Real-Time and Embedded Applications", *Trans. Autom. Control Comput. Sci.*, **63** (49), Ed. "Politehnica" Timisoara, pp. (207–212), May 2004.

[13] R. D. Cioarga, M. V. Micea, B. Ciubotaru, D. Chiuciudean, D. Stanescu, "CORE-TX: Collective Robotic Environment - the Timisoara Experiment", in *Proc. 3rd Romanian-Hungarian Joint Symp. Applied Computational Intellig., SACI'2006*, Timisoara, Romania, May 2006, pp. (495–506).

[14] A. Burns, A. Wellings, "Real-Time Systems and Programming Languages", 3rd Ed., *Addison Wesley*, 2001.

[15] IEEE, "IEEE Standard 1003.13-2003, Standard for Information Technology – Standardized Application Environment Profile – POSIX Realtime and Embedded Application Support (AEP)", *IEEE*, 2003.

[16] G. C. Buttazzo, "Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications", 2nd Ed., *Springer*, 2005.

[17] L. Sha, R. Rajkumar, J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Trans. Computers*, **9** (39), Sept. 1990.

[18] H. Kopetz, G. Fohler, P. Puschner, et.al., "Real-Time System Development: The Programming Model of MARS", in *Proc. IEEE Intl. Symp. Autonomous Decentralized Syst.*, Apr. 1993, pp. (190–199).

[19] P. Puschner, R. Kirner, "From Time-Triggered to Time-Deterministic Real-Time Systems", in *Proc. 5th IFIP Working Conf. Distrib. Parallel Embedded Syst.*, Braga, Portugal, Oct. 2006.

[20] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, W. Pree, "From Control Models to Real-Time Code Using Giotto", *IEEE Control Syst. Mag.*, **1** (23), pp. (50–64), 2003.

[21] T. A. Henzinger, B. Horowitz, C. M. Kirsch, "Giotto: A Time-Triggered Language for Embedded Programming", *Proc. IEEE*, **91**, pp. (84–99), 2003.