Design and Implementation of a FTT-CAN Communication Infra-Structure for the RT-femtoJava Processor

Rita Kalile Almeida Andrade, Thomás Alimena Del Grande, Tiago Bücker, and Carlos Eduardo Pereira

Abstract The paper describes the development of a flexible time-triggered (FTT) communication infrastructure for a customizable Real-time Java processor called RT-FemtoJava. The proposed infrastructure allows a holistic scheduling of both messages and tasks in the platform. It permits a high level of abstraction for implementing distributed and communicating tasks. Two different results are presented: (i) the incorporation of a FTT-CAN communication and a holistic scheduler for the RT-FemtoJava processor and (ii) the design and implementation of the FTT-communication profile on top of a wireless protocol. The developed infrastructure allows the deployment of real-time distributed embedded systems that can balance performance and resource constraints.

1 Introduction

When dealing with Distributed Embedded Real-Time Systems (DERTS), having a reliable and deterministic communication system is mandatory, especially when it involves critical operations, such as in flight-control or process control systems. Additional to this need for a deterministic temporal behavior, the requirement for flexible operation is becoming increasingly important in modern industrial systems. The FTT-CAN protocol [1] is an approach that aims to meet both deterministic vs flexible behavior requirements by supporting both time-triggered (TT) and event-triggered (ET) communication schemes.

Rita Kalile Almeida Andrade
Federal University of Rio Grande do Sul - UFRGS - Informatics Institute

Thoms Alimena Del Grande, Tiago Bcker
Federal University of Rio Grande do Sul - UFRGS - Electrical Engineering Department

Carlos Eduardo Pereira
Federal University of Rio Grande do Sul - UFRGS - Informatics Institute and Electrical Enginnering Department

In this work, an FTT interface for the RT-FemtoJava processor [8] is presented. The RT-FemtoJava is a custommizable processor that interprets Java bytecodes, allowing a high level of abstraction when it comes to writing the software and is suitable for real-time applications for having a real-time clock and an API designed for such utilization. Tasks executed on the RT-FemtoJava processor are scheduled by a holistic scheduler which schedules messages and tasks according to system timing requirements.

The remainder of this paper is organized as follows. A brief overview of the FTT-CAN is presented in Section 2. Section 3 describes related works dealing with the use of FTT-CANs extensions. Section 4 introduces the RT-FemtoJava processor. Section 5 presents the implementation of the FTT-CAN protocol for the RT-FemtoJava and a wireless solution for the same platform in Section 6. Section 7 proposes a holistic scheduler. Concluding, the final remarks are presented in Section 8.

2 Flexible Time Triggered on CAN (FTT-CAN) - briefly review

As already mentioned, FTT-CAN combines time- and event-triggered communication with temporal isolation. An elementary cycle separates the communication in two phases: one for time-triggered messages and another one to event-triggered messages. The scheduling of time-triggered messages is performed at runtime by a master node.

Additionally, the FTT-CAN uses the collision avoidance that is intrinsic of the CAN protocol, reducing the communication overhead. The protocol uses a master-multi-slave transmission control, meaning that the same master message can trigger simultaneously the transmission of the messages in different nodes [1]. CANs arbitration control is also used to control the event-triggered traffic, eliminating the need for pooling messages. Slaves try to transmit pending event-triggered messages immediately after the starting of the appropriate phase. Interested readers should refer [1] for details on FTT-CAN.

3 Related Works

Recent proposals address extensions to FTT protocol to improve some drawbacks. The approach presented in [2] introduces an extension to the FTT-CAN that improves the bit stuffing pessimism and eliminates priority inversion situation and introduces an offset method to enforce correct message order. Additionally, to reduce the jitter, a time slot for TT tasks was proposed in order to reduce the interference of the ET messages within the TT phase. The extension was implemented over an embedded Real-Time Linux.

In [4] a framework was built to support design of task and message dispatching that uses a centralized approach through a holistic scheduler. This work specifies necessary tasks and messages parameters and a mechanism to synchronize the scheduling of them. This mechanism was validated by the SimHol simulator.

In [3] a computational model based on RMI and RTSJ definition was presented. That work assembles a convergence layer that manages the underlying resources involved in a master-slave communication through a new API. It is based on the Flexible Time-Triggered communication paradigm adapted to the unicast environment provided by RT-RMI. The cost of sending and processing a trigger signal is evaluated using a mono-processor environment. Both master and slaves reside in the same virtual machine, in order to minimizes the network effects on the application. For that work uses the jTime [3] virtual machine.

This work differs from the presented approaches above, in the sense that it proposes a holistic scheduling systems that follows the RTSJ standard and the FTT-CAN paradigm. From developers point of view, calls to remote methods do not differ from calls to local objects. Event-triggered messages are scheduled according to the actual runtime situation (i.e. messages priority and ready tasks's priorities) without disturbing time-triggered messages. The proposed mechanism runs over the configurable Java platform called RT-FemtoJava platform.

4 RT-FemtoJava

RT-FemtoJava is a configurable platform that implements a stack machine processor with different organization (e.g. multicycle, pipeline, VLIW) which natively executes Java bytecodes and provides a set of APIs to implement the embedded systems software. The RTFemtoJava processor is configured through the SASHIMI environment [6], which takes as input Java bytecoded. Additionally, it optimizes the binary code to assure the predictability of applications software. Details on this optimization process can be found in [7].

The embedded systems software is written using an API based on the Real-Time Specification for Java (RTSJ) which was developed to express time and other constraints of the embedded real-time applications. This specification introduces the concept of schedulable objects, which are instances of classes that implement the Schedulable interface, such as the RealtimeThread. It also specifies a set of classes to store parameters that represent a particular resource demand from one or more schedulable objects. For example, the ReleaseParameters class (superclass from AperiodicParameters and PeriodicParameters) includes several useful parameters for the specification of real-time requirements. Moreover, it supports the expression of the following elements: time values (absolute and relative time), timers, periodic and aperiodic tasks, asynchronous events and their handlers, and different scheduling policies.

The next section a proposal to integrate the RTSJ-based API with the communication API trough a holistic scheduler that follows the FTT-CAN protocol will be presented.

5 FTT-CAN Integration

In order to allow a RT-FemtoJava processor to communicate over a FTT-CAN network, a FTT-CAN module was written in VHDL language. Advantages of this solution from a software implementation are the lower jitter and lower processor utilization. The main disadvantage is, clearly, higher die area.

It utilizes a CAN module that implements the native CAN protocol, that involves data framing, bit synchronization, bit stuffing, CRC checking, bus arbitration and so on. Taking advantage of that, the FTT-CAN module is built on top of the CAN module, by means of finite state machines that manage the timing constraints imposed by the FTT-CAN protocol.

Before synthesizing the project, one needs to specify if the node in question is a possible master or not. Bus masters are responsible for generating the Trigger-Message. They are also responsible for scheduling the TT messages. For this purpose, it was used a dual-port RAM memory, allowing a future development of an admission control system either by the RT-FemtoJava or a separate entity. In the current version, the master node reads parameters -like period and phase - from this memory, and schedules the messages with the granularity of one Elementary Cycle, that means, the smaller period of a synchronous message is the period of the Elementary Cycle.

By the start of the Trigger Message, the nodes set a global counter to 0 (zero), so that all FTT-CAN nodes in the network are synchronized. This process avoids priority inversion in the synchronous window because the transmission of the message with the highest priority will not be delayed by any means, while in a software implementation the transmission could be delayed by another thread or process running in the processor. Bit-stuff pessimism in the synchronous window is also avoided with the creation of time-slots that are longer than the longest message (considering all possible bit stuffing).

The interface between the RT-FemtoJava and the FTT-CAN module is obtained via memory-mapped registers. In the current configuration, the processor writes in specific registers the message identifiers that it wishes to produce to or consume from the bus. After that, the processor can write data in transmission registers that will have identifiers previously configured and read data from reception registers. The processor is responsible for polling a status register to know if determined messages have arrived or have been transmitted successfully.

6 A wireless approach for FTT

Additional to the us of a FTT-CAN approach, a wireless FTT interface was also implemented on the top of IEEE 802.15.4 standard, taking advantage of the superframe structure, shown in figure 1. The superframe is bounded by the transmission of periodic beacon frames, followed by a Contention Access Period (CAP) and an optional contention free-period (CFP), used by low latency applications. A simple equivalence between the elementary cycle of FTT and this superframe structure gives an interesting solution to develop a FTT wireless system.

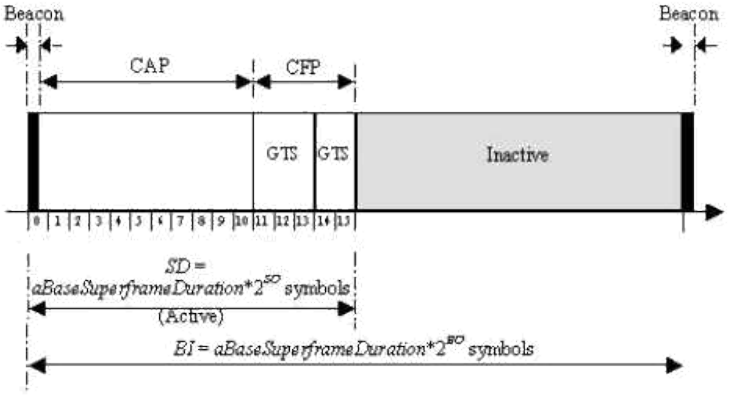

Fig. 1 Exemple of Superframe Structure.

The FTT communication over the IEEE 802.15.4 standard was also written in VHDL language and connected to the RT-FemtoJava. The master node, defined before synthesizing the project, broadcast the beacon frames at periodic intervals according to the generics sets in the top of the entity. Like in the FTT-CAN module, the scheduled of TT messages are supported by a dual-port RAM memory. In this case, however, the memory must also contain the address of nodes able to transmit. If any TT message is sent in the current elementary cycle, the master node sends the correct CFP parameters to allocate a guaranteed time slot where only the scheduled node can transmit.

Slave synchronization is done in every beacon frame receipt, when a counter starts and produces the 15 time slots of the active superframe structure. In the CAP region, when a node wants to transmit, the transceiver is set to energy detection mode and returns an interrupt signalling whether the channel is busy or not. This mechanism allows the traffic of asynchronous messages. How said in the last paragraph, TT traffic are in the CFP region, where only the scheduled node can transmit. Traffic isolation is promoted by the beacon frame subfield final CAP slot.

The interface between RT-FemtoJava and FTT module is the same defined in the previous section. A little difference is related to the producer/consumer model implemented, where slave nodes use a point-to-point communication with master to produce its message and this one broadcasts it. Therefore, the master must consume all network messages.

7 The FTT-CAN Middleware

Software is increasingly becoming the major cost factor for embedded devices. Nowadays, with the growing complexity of DERTS, it is necessary to use techniques that increase software productivity. In this context, a FTT-CAN middleware was developed to simplify the design and implementation of real-time embedded applications.

This middleware asserts more transparency in the implementation of distributed and communicated Java objects. Furthermore, the middleware incorporates a holistic scheduler that handles the scheduling of both messages and tasks, according to system timing requirements. The figure 2 illustrates the proposed middleware - which will be detailed in the next subsections.

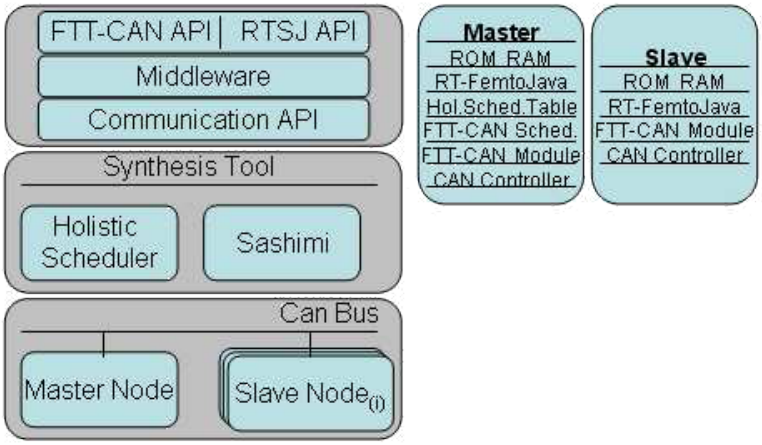

Fig. 2 Design flow of the Platform.

7.1 The Framework

This framework is composed of a middleware and APIs who allows to abstract inherent details about the distribution and the communication protocol. Standalone tasks must follow the RTSJ-based API, mentioned at Section 4. Communicating tasks must be specified using primitives of the FTT-CAN API and their temporal parameters using the RTSJ-based API.

The middleware identifies temporal tasks parameters and their messages and organize them to be used by the FTT-CAN Scheduler, that composes the FTT-CAN Module. The messages are separated in asynchronous or event-triggered (ET) and synchronous or time-triggered (TT) which are based at priority and periodicity, respectively. Messages parameters are marshalled and unmarshalled by the middleware, making the distribution transparent to programmers.

Communication facilities are provided through the APICOM for the RT-FemtoJava processor, which adds an interface between the application layer and the communication system detailed in Section 5 and 6. The communication system was proposed to provide synchronous and asynchronous message exchange among objects running at different RT-FemtoJava processors into the same chip and/or running at different nodes connected through a communication network. The API allows applications to establish a communication channel through the network, which is used to send and receive messages. The service allows the assignment of different priorities and periods to messages and runs in a multithread environment. From the application point-of-view, the system is able to open and close connections, in a client-server mode, or run in publisher-subscriber mode.

7.2 The Holistic Scheduler

According to the communication paradigm, every communicating task uses messages to exchange data with other tasks. However, at a first moment, the ET tasks are not considered by holistic scheduling process, but they are equally supported by the platform development. Although predictability is a requirement for both ET and TT phases, our focus here is on the response time of the TT phase because it requires a high degree of responsiveness (since TT are usually time critical with hard real-time requirements).

The FTT-CAN Scheduler in the Master Node uses a table to make the global synchronization to join the dispatching of tasks and messages exchanges. This table is built by the holistic scheduler. The holistic scheduler creates a graph which contains the order of dependences among communicating tasks. From this graph the scheduler is made by selecting tasks (node-centric) or messages (netcentric) and adapting its dependences, which are known by through the graph, according to system timing requirements. This scheduler follows the approach specified in [4].

8 Final Remarks

This paper presents an ongoing work that proposes a holistic scheduler component, which will integrate an RTSJ-based API and a communication API. The selected communication protocol is the FTT-CAN. This choice was made mainly due to the characteristics of the FTT-CAN protocol, which can provide features such admission control and flexibility. Additionally, was implemented a wireless approach. Both modules, FTT-CAN and wireless, are integrated to RTFemtoJava processor and synthesized in a Virtex-II Pro Xilinx [5] FPGA.

A holistic scheduler was implemented, as well as the parameters describing communication characteristics of task. Currently the system is being validated through some case studies in order to ensure that the proposed scheduler meet all specified application requirements. In future work, we intend to complete the middleware integration to the platform and obtain results about jitter and latency.

References

1. L. Almeida, J. Fonseca, and P. Fonseca. The FTT-CAN Protocol: Why and How. *IEEE Transactions on Industrial Electronics*, 49(6), December 2002.
2. F.H. Athaide, C.E. Pereira, and V.F. Silva. A new approach for time-triggered phase in the FTT-CAN protocol a case study in an automotive system. In *Proc. of RTSS*, 2006.
3. P. Basanta-Val, L. Almeida, and M. Garca-Valls. Towards a synchronous scheduling service on top of a unicast distributed real-time Java. In *Proc. of Real Time and Embedded Technology and Applications Symposium*, 2007.
4. M. J. Calha. A holistic approach towards flexible distributed systems. Technical report, Universidade de Aveiro Departamento de Electrnica e Telecomunicaes, 2006.
5. http://www.xilinx.com.
6. S. Ito, L. Carro, and R.P. Jacobi. Making Java work for micro-controller applications. *IEEE Design & Test of Computers*, 18(5):100–110, 2001.
7. M. A. Wehrmeister, C. E. Pereira, and L. B. Becker. Optimizing the generation of object-oriented real-time embedded applications based on the real-time specification for Java. In *Proc. of DATE06*, pages 806–811, Munich, Germany, 2006.
8. M.A. Wehrmeister, L.B. Becker, and C.E. Pereira. Optimizing real-time embedded systems development using a RTSJ-based API. *Lecture Notes in Computer Science*, 3292:292, 2004.