

# Flexible Time-Triggered Protocol for CAN – New Scheduling and Dispatching Solutions

J. Fonseca, E. Martins, L. Almeida, P. Pedreiras, P. Neves \*

**One of the possibilities to build robust communication systems with respect to their temporal behaviour is to use autonomous control based on the time-triggered paradigm. The FTT-CAN - flexible time-triggered protocol, relies on centralised scheduling but makes use of the CAN native distributed arbitration to reduce communication overhead. There, a planning scheduler is used within a master node to reduce the scheduling run-time overhead. On-line changes to the communication requirements can then be made under guaranteed timeliness. In addition FTT-CAN also allows an efficient combination of both time-triggered and event-triggered traffic with temporal isolation.**

**In this paper, recent evolutions of the initial protocol definition concerning transmission of synchronous and asynchronous messages are presented. These consist in a time division of the elementary transmission window which optimises the available bandwidth for asynchronous messages, keeping the timeliness of synchronous messages without jeopardising their transmission jitter. A novel solution for the planning scheduler is also presented. It consists in an FPGA-based coprocessor which implements the planning scheduler technique without imposing overhead to the arbiter CPU. With it, it is possible to reduce strongly the plan duration thus allowing on-line admission demanded by system elements and, also, to extend the protocol application to high-speed networks.**

## 1. Introduction

A main feature that distinguishes the recently introduced FTT-CAN protocol (Flexible Time-Triggered communication on CAN) [1,2] from other proposals concerning time-triggered communication on CAN [3], is that it supports dynamic communication requirements by using centralised scheduling with on-line admission control.

Centralised scheduling based on the planning approach [4] makes it easier to perform on-line changes to the communication requirements, while the CAN distributed access arbitration allows the communication overhead to be kept well below the levels typically found in centralised access control systems [1].

FTT-CAN supports both time-triggered and event-triggered traffic, with temporal isolation being achieved by means of a double-phase elementary cycle concept. The former type of traffic is handled in the synchronous phase and the latter in the asynchronous one. The distributed arbitration is also used to simplify the handling of asynchronous communication requests, resulting in reduced overhead and, consequently, in shorter response times.

This paper describes current developments concerning two aspects of the FTT-CAN protocol: the dispatching of synchronous and asynchronous traffic, and the

scheduling implementation.

A study of the relative positioning of the synchronous and asynchronous phases within the transmission window, is presented in section 2. The emphasis here is on the asynchronous messaging system. A discussion concerning the support of the event and the time-triggered paradigms in FTT-CAN, opens this section.

Next, in section 3, we present a preliminary implementation of a hardware scheduling coprocessor for FTT-CAN based in the planning technique. This coprocessor aims at decreasing the response time to changes in the system configuration or message parameters, opening also the possibility of automatic on-line changes requested by system nodes or human operators, thus improving system reactivity.

Finally, section 4 concludes the paper.

## 2. Combining Synchronous and Asynchronous Traffic in FTT-CAN

It is commonly accepted [3,5] that time-triggered communication is well adapted to control applications that typically require regular transmission of state data with low, or bounded, jitter (e.g. motion control, engine control, temperature control, position control). On the other hand, event-triggered communication is well adapted to the monitoring of alarm conditions that are

---

\* DET / IEETA, Universidade de Aveiro, P-3810-193 Aveiro, Portugal  
{jaf, evm, lda}@det.ua.pt; pedreiras@alunos.det.ua.pt; pneves@ua.pt

supposed to occur sporadically and seldomly, and also to support asynchronous non-real-time traffic e.g. for global system management.

However, many applications do require simultaneous support for cyclic transmission of state data, sporadic transmission of alarms as well as of non-real-time data for system management. Hence, a combination of both paradigms in order to share their advantages seems desirable, particularly if it is possible to enforce a temporal isolation of both sorts of traffic. Otherwise, the event-triggered traffic would spoil the properties of the time-triggered one. With such combination it is then possible to use a time-triggered approach to manage the periodic real-time traffic with controlled jitter and with high efficiency under worst-case requirements. Simultaneously, a portion of the network bandwidth is kept available for sporadic event-triggered traffic, either real-time (e.g. alarms) as well as non-real-time (e.g. on-line management and reconfiguration).

A typical solution enforcing temporal isolation between the two sorts of traffic, is to use the elementary cycle concept, i.e. a fixed duration window composed of two phases, one for each sort of traffic (e.g. discussed by Raja and Noubir [6]). The bus time is slotted in elementary cycles resulting in an alternate sequence of time-triggered and event-triggered phases. The maximum duration of each phase can be tailored to suit the needs of a particular application. If each sort of traffic is forced to remain within the respective phase then, temporal isolation is guaranteed. This concept is used, for example, in the WorldFIP fieldbus [7]. However, since this fieldbus uses a MAC protocol based on centralised arbitration, the handling of event-triggered (aperiodic) traffic is rather inefficient requiring a considerable amount of bandwidth to allow the master node (arbitrator) to become aware of aperiodic requests. In the Foundation Fieldbus [8], one of the 8 profiles of the new international fieldbus standard, a somewhat similar scheme is used. Instead of an elementary cycle, a particular node known as Link Active Scheduler contains the schedule for the time-triggered traffic. This node grants the other nodes, Link Masters, the permission to control the bus and transmit event-triggered messages during precise time windows, only, that do not overlap with the time used by the time-triggered messages. In many other fieldbus systems,

that do not use the elementary cycle concept, it is still possible to specify cyclic time-triggered data exchanges but with no temporal isolation from the event-triggered traffic, e.g. PROFIBUS [7], CAL [9], DeviceNet [10].

## 2.1- FTT-CAN Asynchronous Messaging System

In FTT-CAN a particular node, the master, generates a periodic message used to synchronize all other nodes in the network. The transmission of this message represents the start of one elementary cycle (EC) and is known as *EC trigger message*. The EC duration is fixed and set at pre-run-time. Within each EC, the protocol supports two types of traffic, synchronous and asynchronous. The former one is time-triggered and its temporal properties (i.e. period, deadline and relative phasing) are represented as integer multiples of the EC duration. The EC trigger message, in its data field, conveys the identification of the synchronous messages that must be transmitted by the producer nodes in that EC. The nodes that identify themselves as producers by scanning a local table containing the messages to be produced / consumed, transmit the respective synchronous messages in the synchronous phase of that EC (fig. 1). Collisions on bus access are resolved by the native distributed MAC protocol of CAN. The synchronous traffic is transmitted autonomously by the synchronous messaging system (SMS).

The FTT-CAN protocol also supports asynchronous traffic for event-triggered communication with external control. This sort of traffic is transmitted during the periods of the EC not used by the synchronous messages. However, depending on how the desired temporal isolation between these two sorts of traffic is enforced, the asynchronous messaging system (AMS) can operate in one of two modes. In *controlled mode* any asynchronous message is transmitted only if it is guaranteed not to interfere with the timeliness of the EC trigger message or of the synchronous messages. In this mode each station that produces asynchronous messages is allowed to transmit them only if there is enough bus-time left within the asynchronous window. This enforces a strict temporal isolation between synchronous and asynchronous traffic. On the other hand, two negative aspects can be identified, the insertion of bus idle-time when asynchronous messages do not fit in the respective window and the need to force all the nodes in the system to follow the synchronisation imposed by the EC trigger message in order to determine the exact duration of each phase in each EC.

In *uncontrolled mode*, stations wishing to transmit asynchronous messages can try to do it as soon as they receive the respective requests from the application. There is no need to synchronise with the EC trigger message and thus, even stations not engaged in the FTT-CAN protocol can coexist in the system and send asynchronous messages. Although these messages may

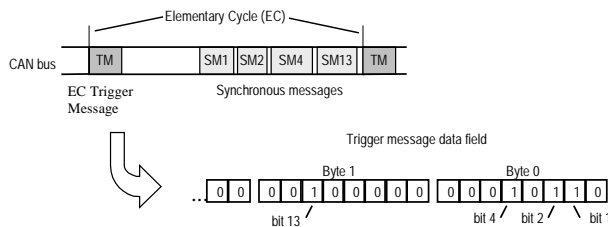
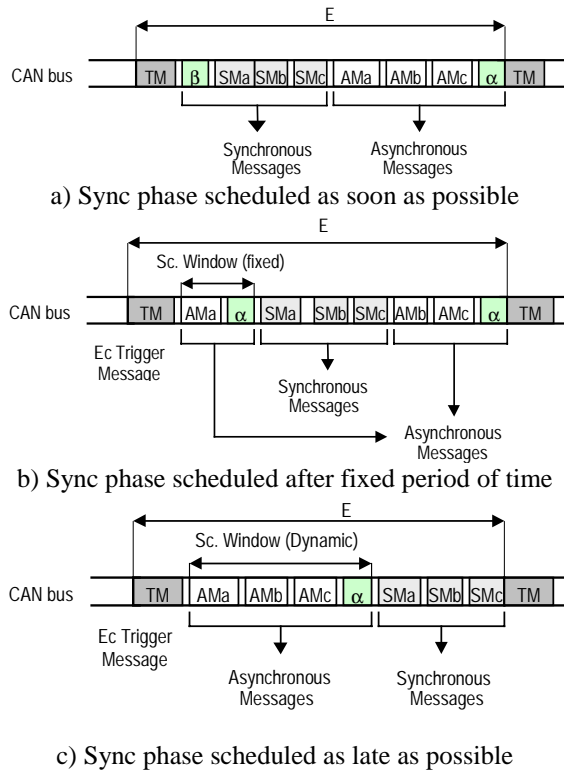


Fig. 1. EC Trigger Message data contents.



**Fig. 2.** Arrangement of sync/async phases within the EC.

now cause a certain blocking to the transmission of synchronous ones, such blocking can be upper bounded by using a proper choice of identifiers.

In FTT-CAN the message identifier range is divided in two parts. Higher priority identifiers are assigned to synchronous messages while lower priority identifiers are assigned to the asynchronous ones. In particular, the EC trigger message is assigned the highest priority identifier. Therefore, each group of consecutive synchronous messages, i.e., transmitted without releasing the bus arbitration process, can suffer blocking by, at most, one asynchronous message.

The advantages of this mode are twofold. Firstly, FTT-CAN can be installed in an existing system while keeping some of the legacy nodes not engaged in the new protocol. Secondly, under uncontrolled mode there is no inserted idle-time as in controlled mode, thus allowing for higher bus utilisation.

The price to pay is a bounded degradation of the synchronous and trigger message timings, i.e. increased jitter. How much is this degradation depends strongly on the particular bus controllers used. For example, when Full-CAN controllers (with multiple transmission buffers) are used, all the synchronous messages scheduled for one EC are transmitted without any intermediate blocking (at most one blocking in the beginning). On the other hand, if Basic-CAN controllers (with a single transmission buffer) are used,

then, each time a node produces more than one synchronous message in the same EC there may be extra blockings. In this case, the controlled mode might be more appropriate. Tindel *et al.* [11] clearly show the difference between these two types of controllers in terms of real-time performance.

In this paper, either the use of Full-CAN controllers, or Basic-CAN but with each node producing at most one synchronous message, will be considered.

## 2.2- Arranging Sync/Async Phases in the EC

When implementing the FTT-CAN protocol, it is important to define how the synchronous and asynchronous phases will be combined within the EC. In preliminary experimental work, two different options have been considered: the synchronous phase starting right after the EC trigger message and starting after a fixed period of time called *scanning window* [1]. In this section, these options, plus a third one considering that synchronous messages are always transmitted at the end of the EC, are analysed and compared. These three situations are referred to as: synchronous phase scheduled as soon as possible, scheduled after a fixed period of time (scanning window), and scheduled as late as possible.

### *Sync Phase Scheduled as Soon as Possible*

In this situation, the synchronous messages are produced as soon as they are identified by the producers, i.e. in the beginning of the EC, right after the trigger message (fig. 2-a). Notice, however, that before starting the transmission of any synchronous message all nodes require a certain time to scan their local tables and identify themselves as producers in that EC (appears as  $\beta$  in fig. 2-a). This time can be small (relative to the transmission time of one message) when using powerful microprocessors, or it can be larger when low processing-power microcontrollers are used<sup>1</sup>. In any case, all nodes engaged in the protocol release the bus during this period of time  $\beta$ . If there were asynchronous messages pending, at least one of them could gain access to the bus and eventually delay the start of transmission of the synchronous messages. Moreover, the amount of time  $\beta$  varies with the relative position of each synchronous message in the local table. Hence, to enforce a strict temporal isolation between both types of traffic, as desired in the asynchronous controlled mode, asynchronous messages are released for transmission only after the synchronous phase is over. Notice that, in this case, the amount of bus-time  $\beta$  is wasted, a situation that is particularly negative when

<sup>1</sup> In experiments with the 80C592 Philips controller (8051-based with Basic-CAN controller) clocked at 11MHz,  $\beta$  was close to the transmission time of a 4-data bytes message (@125Kbit/s).

using low processing-power microcontrollers. Furthermore, to strictly maintain the regularity of the EC trigger message, a variable amount of idle-time  $\alpha$ , upper bounded by the time to transmit one message, is inserted at the end of the asynchronous phase.

#### ***Sync Phase Scheduled After Fixed Period of Time***

In an attempt to make the time spent in scanning the local table ( $\beta$ ) usable by asynchronous traffic even under controlled mode, a *scanning window* of fixed duration was defined between the end of the trigger message and the start of the synchronous phase (fig. 2-b). The asynchronous phase is now divided in two parts, one corresponding to the scanning window and another one corresponding to the time left after the synchronous phase. In any of these parts, under controlled mode, messages are transmitted only if they can be guaranteed to fit in the respective windows. Otherwise, they are delayed to the following ECs. This corresponds to the insertion of idle-time ( $\alpha$ ), in each part, to enforce the regularity of the trigger and synchronous messages.

#### ***Sync Phase Scheduled as Late as Possible***

This option, also referred to as *dynamic scanning window*, has the major advantage over the previous ones of maximising the time available to asynchronous messages within one EC and keeping it all together (fig. 2-c). Therefore, insertion of idle-time ( $\alpha$ ) to enforce the timeliness of the synchronous messages is required at most once per EC, leading to a higher average capacity to handle asynchronous messages. This arrangement of the two phases can easily be accomplished by also using the EC trigger message to convey the duration of the asynchronous phase, e.g. using low order bits of the message identifier. This can be carried out with minimal extra run-time overhead.

Another interesting feature is that when a message transmission request is aborted because it does not fit in the current asynchronous window, it is delayed until the next EC. However, the request is resubmitted during the transmission of the trigger message of the next EC so that it re-enters arbitration in the very beginning of the next asynchronous window. This detail assures that there will be no further priority inversions whenever an asynchronous message is delayed from EC to EC until it is transmitted.

Furthermore, since the most regular edge of the synchronous phase, in terms of periodicity, is now the ending edge (near the next EC trigger message), the higher priority synchronous messages must be transmitted later within the respective phase in order to reduce jitter. This is achieved by assigning lower priority identifiers to higher priority synchronous messages (notice that the priority of the synchronous messages and the respective identifiers are different parameters [1]).

#### ***Choosing an Arrangement***

The three situations referred above were analysed concerning, particularly, their use under the asynchronous controlled mode. In such mode, scheduling the synchronous phase as late as possible seems to be more advantageous leading to a more efficient bus utilisation. This is particularly true when using low processing-power microcontrollers.

With uncontrolled mode the differences between those three situations are not so evident. In fact, in this case possible blockings caused by asynchronous messages are inevitable in any situation. Such blockings may occur if an asynchronous message is still being transmitted when the EC trigger message is released or when the synchronous window starts.

However, a previous study [12] showed that a worst-case response time analysis of the AMS is easier to carry out, and is more precise, when the block of synchronous and trigger messages is transmitted in a row, i.e. without preemption. This is achieved, only, when the synchronous phase is scheduled as late as possible. Therefore, this arrangement, with the asynchronous phase first, followed by the synchronous one, will be preferably considered in future developments of FTT-CAN.

### **3. Implementing the Planning Scheduler in Hardware**

As previously referred, scheduling of the time-triggered traffic is performed in FTT-CAN in a centralized way. The scheduler is an integral part of the SMS and runs in the master node generating the periodic EC trigger messages.

To meet the required level of operational flexibility, low overhead and timeliness guarantees, software-based schedulers are built following a dynamic table-based approach known as the *planning scheduler* [2].

In this section we describe our first experience in transferring this scheduling technique to a hardware implementation. The resulting coprocessor is described focusing on its functionality and interface with the master node CPU. Its internal architecture is briefly presented, together with some figures showing the feasibility of the proposed architecture.

#### **3.1. Scheduling in FTT-CAN - The Planning Scheduler**

Message scheduling on a fieldbus can be done statically or dynamically. Table driven and priority-based approaches such as the ones in FIP and CAN respectively, fall in the category of static scheduling while dynamic scheduling can be done using planning based or best effort approaches. Although dynamic planning-based schedulers are not commonly found in current standard fieldbuses, recent work on the subject [4], has shown

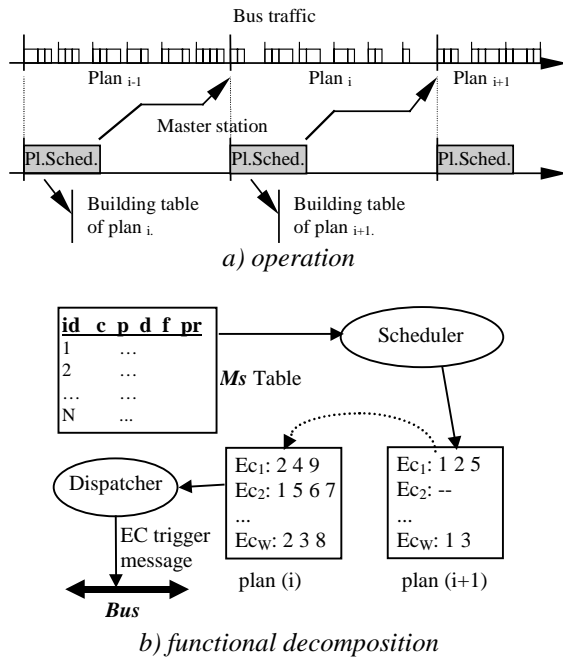


Fig. 3. The planning scheduler.

they could become a good compromise between the static and dynamic approaches.

The planning scheduler and an associated dispatcher can be implemented in fieldbus-based systems imposing an overhead compatible with the low-processing power microprocessors or microcontrollers used as typical nodes' CPUs. Also, it presents some degree of flexibility resulting from the possibility to change, from plan to plan, the message's set, adding or deleting messages or changing their parameters. The underlying concept is the reservation of resources into the future. So, when a new message is accepted, the additional bus bandwidth required is reserved. To do this, the scheduler builds static schedules for consecutive fixed duration periods of time called plans. The static schedules are called plan tables. The creation of a plan table is overlapped with the dispatching of the previous. In figure 3 the operation of the planning scheduler is illustrated. The dispatcher works with plan  $i$ , while the scheduler builds plan  $i+1$ .

Each plan includes a fixed number of ECs. Messages' periods are then restricted to an integer multiple of the EC time. Transmission time of the longest message is supposed to be less than the EC duration.

The simple mechanism of this scheduler reduces run-time overhead mainly because it is invoked fewer times. So, comparing with a dynamic scheduler, each time it is invoked, instead of determining the next message to be transmitted, only, it determines all the bus activity, for all the messages, for a certain period of time corresponding to the plan duration.

### 3.2- Scheduling with a Dedicated Coprocessor

#### Motivation

Experimental results [1] taken in a CAN-based system where the planning scheduler was implemented showed an exponential decrease of run-time overhead with the plan duration. Results have shown also that, for a typical EC duration of, say, 8.9ms, and 20-EC plans, the response time to a request of change in the message set is normally more than adequate when it comes from a human operator. Also, the response time can be reasonable for automatic changes during set-up or upgrade of the system. However, if more dynamic mechanisms are to be thought for the system operation, e.g., changing messages' periodicity to react to a bus overload or to adapt the sampling period of a distributed control system (operation following a QoS - quality of service model), then the response time is clearly insufficient. To overcome this limitation the plan duration should be reduced. Adding to the increased runtime overhead caused by the reduction of the plan, the implementation of automatic procedures to allow on-line changes in the communication parameters will also require relevant processing power at the arbiter node CPU.

Apart from the obvious solution of simply adopting a much more powerful CPU to keep up with all this processing needs, another interesting possibility is to use dedicated hardware to offload the node CPU in the scheduling task. The repetitive nature of the scheduling process, the robustness required for the arbiter node and the desire to reduce strongly the response time to changes led to choose the hardware coprocessor as the first solution to explore. This option was reinforced by the fact that the planning technique makes very easy the exchange of data between the coprocessor and the arbiter CPU, even when the worst case execution time of the scheduling process is not completely determined. The output of the scheduler is, in this case, a list of messages to be produced during several ECs. Although other solutions such as a scheduling coprocessor based in another CPU are yet to be studied in the future, the use of dedicated hardware is presently a good and easy option namely due to the availability of support tools [13].

#### Related Work

While virtually nothing has been reported on specialised hardware for message scheduling in fieldbuses, some recent papers have surfaced describing coprocessors aiming at improving the execution time and predictability of operating system functions.

The Real Time Unit (RTU) reported in [14] is a complete multitasking kernel implemented in an ASIC. It consists of a number of units which handle most of the time-critical functions of a typical real-time kernel. Task scheduling is based on the rate monotonic

algorithm. The RTU can handle a maximum of 64 tasks at 8 priority levels, and supports up to 3 application processors. The prototype described was used in a VME system with 3 CPU boards executing tasks. The interaction between the processors and RTU is through interrupts and registers which makes it easy to use the RTU with different types of processors.

The Spring Scheduling CoProcessor (SSCoP) [15] is a VLSI coprocessor dedicated only to the task of scheduling. It was designed to work together with the Spring kernel and supports also multiple processors. The SSCoP can use different scheduling algorithms, considering shared resource requirements and precedence constraints. The operating system writes the attributes of a set of tasks in the coprocessors registers. Using these attributes SSCoP tries to build a complete feasible schedule, which, if successfully created, can be read back by the operating system.

Finally, [16] describes a universal scheduling coprocessor for single processor systems. The coprocessor is provided with the task parameters and states, and gives back to the operating system the identification of the task that has to be executed next. The architecture approach is suited for the implementation of nearly every scheduling algorithm that is based on comparison of task parameters. The coprocessor was implemented in FPGA technology and its latest version uses the ELLF scheduling algorithm and supports up to 32 tasks.

### 3.3- The Planning Scheduler CoProcessor (PSCoP)

The coprocessor currently under development differs from the previous solutions because it directly follows the planning paradigm. PSCoP has then a limited amount of memory to store a scheduler plan i.e. the identification of the messages that must be transmitted each EC of the plan. PSCoP memory is divided in two banks allowing the coprocessor to generate one schedule plan while the CPU dispatches the other.

PSCoP is targeted to work specifically with FTT-CAN. The architectural solution described next presents some degree of scalability since the number of messages can be adapted depending on the operational needs.

#### The Node CPU Interface

To start working, PSCoP needs to be initialised first with the parameters of each variable to be scheduled. These include the variable's period (P), its initial phasing (Ph) and associated transaction duration (C). The parameters of each variable are written by the node CPU in a three register slot within PSCoP's interface. There are as many register slots as the maximum number of variables supported by the coprocessor.

In this experimental version there is no support for explicit deadline or priority parameters. The deadline of all variables is assumed to be the same as their period.

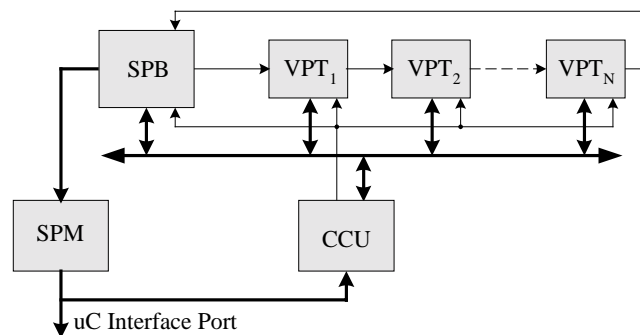
Relative priorities are dictated by the allocation of register slots. These are numbered 1 to N and have assigned decreasing priorities. The scheduling priority of a given variable is thus set by mapping its parameters to the appropriate register slot at initialisation time. Clearly, priorities are always static.

The interface includes also an EC register which must be initialised with the elementary cycle duration parameter. A control/status register allows the CPU to start or stop the coprocessor, and provides information about the current state of the scheduling operation. After instructed to begin PSCoP starts generating schedules. The message schedule for each EC in the plan is presented to the node CPU as an N-bit word which identifies the transactions that must be carried out during that EC. The coding scheme of this word is the same as the one used in the FTT-CAN trigger message data field (see again fig. 1), and it was adopted in order to reduce the dispatching overhead.

#### Architecture Overview

In devising a hardware structure where the planning scheduler functionality could be mapped, two separate activities were identified within the scheduler algorithm. One of them is performed in the context of each variable and acts basically as a timer, keeping track of the instants when the variable must be produced. The other concerns the placing of transactions in the respective ECs in the plan table. This partitioning of activities inspired the architecture depicted in figure 4. Here, the Variable's Production Timer (VPT) units are responsible for the first activity while the Schedule Plan Builder (SPB) takes care of the second activity.

Each variable to be scheduled is allocated to one VPT unit which holds the variable's period (P) and initial phase (Ph) parameters. Global timing information received from the SPB allows all VPTs to be synchronised while keeping track of the EC schedule currently being generated. When a VPT detects that the



**Fig.4.** PSCoP architecture. SPB - Schedule Plan Builder; VPT - Variable's Production Timer; SPM - Schedule Plan Memory; CCU - Configuration Control Unit.

scheduling for a particular EC where its variable should be produced has started, it signals the SPB requesting the allocation of the associated transaction. Based on the transactions' duration (C) and on the remaining EC time left, the SPB unit decides to allocate or reject the transaction. If the transaction is accepted, further requests for allocation in the same EC (from other VPTs) are received, otherwise the current EC schedule is finished and a new one is started.

Because more than one VPT can request allocation in the same EC, a mechanism must exist to help SPB to select which request to serve first. A daisy chain structure similar to the one commonly found in microprocessor-based systems is used with this purpose. The chain signal ripples through VPT<sub>1</sub> down to VPT<sub>N</sub>. When a VPT unit raises a request for allocation its chain signal output is deactivated. After this, the unit is allowed to communicate with SPB only if its chain signal input is true, which means that, in a contention situation, the leftmost VPT with a pending request is always the only one with the chain signal input set to true, and therefore the one which can engage communication with SPB.

Besides the VPTs and SPB the PSCoP architecture includes two other functional blocks, the Configuration Control Unit (CCU) and the Schedule Plan Memory (SPM). The former includes control and status registers and provides access to the parameter registers in the VPTs and SPB. The SPM unit is where SPB builds the plans with the EC schedules it generates.

#### ***Preliminary Feasibility Assessment***

The first prototype of PSCoP will be implemented on a XC4010XL FPGA. It will have 64 VPTs and a parameter resolution of 8-bits. The memory banks in SPM will support 20-EC plans, or, in other words, will be 20 x 64-bits FIFO memories. The prototype will be tested on a CAN master node based on a XS40 development kit from XESS<sup>®</sup> Corporation [17].

At the time of writing the coprocessor is still in the design entry stage, and so simulation results are not yet available. Nevertheless, an accurate estimate of performance was obtained by carrying out a step by step analysis of the various phases of the coprocessor's internal operation, counting the number of clock cycles required by each.

Each variable allocation takes 6 clock cycles. In the end of each EC, another 3 clock cycles are needed to transfer the schedule to the SPM unit and to begin the next schedule. The time taken by PSCoP to build a complete plan with W ECs,  $t_{sched}$ , can thus be expressed (in clock cycles) as written below, where  $Nv(EC_i)$  is the

$$t_{sched} = 3.W + 6. \sum_{i=1}^W Nv(EC_i)$$

number of variables allocated in EC<sub>i</sub>.

To calculate a worst case scheduling time in our prototype version, we shall assume a maximum number

of allocations in every EC of the plan. For this to occur all messages must have the smallest possible length, which, if we consider CAN2.0A format and a 1Mbit/s data rate, corresponds to a minimum transmission time of 44μs [18]. If we consider an EC duration of 1ms, then we can have at most 22 of these minimum length messages per EC, in every EC. Using the expression above, the scheduling time in this worst case scenario is computed as 2700 clock cycles. Since the FPGA in the development board is clocked at 12MHz, this translates to 0.22ms, or 1.1% of the time taken by the CPU to dispatch an entire plan.

## **4. Conclusion**

This paper focuses on the FTT-CAN protocol, which was designed primarily to support time-triggered communication on CAN in a flexible way. Two current developments of the protocol are presented, namely the allocation of the synchronous and asynchronous phases within the elementary cycle, and a coprocessor for scheduling of synchronous traffic in the master node.

The paper discusses first the event and time triggered paradigms in fieldbus communication systems, and the advantages that may arise from a combination of both in a way that enforces temporal isolation between the two types of traffic. This combination results in the synchronous messaging system (SMS) and the asynchronous messaging system (AMS), which co-exist both in FTT-CAN.

Allocating the synchronous phase as late as possible in the EC seems to be the solution that best promotes a more efficient bus utilisation, and at the same time facilitates the response time analysis, particularly when the AMS is used in the uncontrolled mode.

The second FTT-CAN development is PSCoP, a scheduling coprocessor that works according to the planning scheduler principle. The main goal here was simply to design a working coprocessor which could fit in a medium-sized FPGA, and be used as an initial testbed to obtain insight on the real performance gains and problems of the architecture.

A first implementation of PSCoP will be available shortly. As shown with the rough performance estimate given for this initial version working at a modest clock rate, PSCoP can easily create a plan table in a small fraction of an EC in a field-bus running at 1Mbit/s. This result is quite encouraging in the development of the coprocessor because it suggests that some of the performance room may be sacrificed in favour of a few design improvements and additional functionality.

At this point it is clear that one of these improvements must concern the static arbitration method used to resolve the contention between several VPTs. We are considering the adoption of a scheme relying on dynamic priority vectors in order to allow the implementation of various scheduling policies, like

RM, DM or priorities-based, between which the coprocessor can be switched dynamically.

## References

- [1] Almeida, L., J.A. Fonseca, P. Fonseca. A Flexible Time-Triggered Communication System Based on the Controller Area Network: Experimental Results. Proc. of FeT'99 (Int. Conf. on Fieldbus Technology). Magdeburg, Germany, September 1999.
- [2] Almeida, L. "Flexibility and Timeliness in Fieldbus-based Real-Time Systems". PhD Thesis, University of Aveiro, Portugal. November 1999.
- [3] Peraldi, M.A. and J.D. Decotignie. Combining Real-Time Features of Local Area Networks FIP and CAN. Proc. of ICC'95 (2<sup>nd</sup> Int. CAN Conference). CiA – CAN in Automation, 1995.
- [4] Almeida, L., R. Pasadas and J.A. Fonseca. Using a planning scheduler to improve the flexibility in real-time fieldbus networks. IFAC Control Engineering Practice, 7: 101-108, February 1999.
- [5] Thomesse, J.-P, M. Leon Chavez. Main Paradigms as a Basis for Current Fieldbus Concepts. Proc. of FeT'99 (Int. Conf. on Fieldbus Technology). Magdeburg, Germany. September 1999.
- [6] Raja, P. and G. Noubir. Static and Dynamic Polling Mechanisms for Fieldbus Networks. ACM Operating Systems Review, 27(3), 1993.
- [7] European standard EN 50170. General Purpose Fieldbus: Vol.1: P-Net; Vol.2: PROFIBUS; Vol.3: WorldFIP. CENELEC, European Committee for Electrotechnical Standardisation, 1996.
- [8] IEC Draft Standard 61158-3,4: Fieldbus standard for use in industrial control systems – part 3: Datalink service specification; - part 4: Data link protocol specification. IEC, Int. Electrotechnical Committee, 1998. This specification became profile 1 of the 61158 standard after January 2000.
- [9] CiA DS 201-207. CAN Application Layer for Industrial Applications. CiA, CAN in Automation International Users and Manufacturers Group, 1994.
- [10] DeviceNet Specification – release 2.0, Vol. I and II. ODVA – Open DeviceNet Vendor Association, Inc., USA, 1997.
- [11] Tindell K., H. Hansson and J. Wellings. Analysing Real-Time Communication: Controller Area Network (CAN). Proc. of RTSS'94 (15th IEEE Real-Time Systems Symposium), 1994.
- [12] P. Pedreiras, L. Almeida; "Combining Event-Triggered and Time-Triggered Traffic in FTT-CAN: Analysis of the Asynchronous Messaging System"; Proc. 3<sup>rd</sup> IEEE Intl. Workshop on Factory Communication Systems, Porto, Portugal, Sept. 2000.
- [13] Valery Sklyarov et. al.; "Development System for FPGA-Based Digital Circuits", Proc. FCCM'99: IEEE Symp. Field-Prog. Custom Computing Machines, USA, April de 1999.
- [14] J. Adomat et. al.; "Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems"; Proc. of Euromicro RTS '96, L'Aquila, Italy, 1996, pp.164-168.
- [15] D. Niehaus et. al.; "The Spring Scheduling Coprocessor: Design, Use, and Performance"; Proc. of the 14th IEEE Real-Time Systems Symposium, USA, 1993, pp.106-111.
- [16] J. Hildebrandt, F. Golasowski D. Timmermann; "Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems"; Proc. 11th Euromicro Conf. on Real-Time Systems, England, June, 1999, pp.208-215.
- [17] XESS Corporation, URL: <http://www.xess.com>.
- [18] K. Tindell, A. Burns, and A. Wellings; "Calculating Controller Area Network Message Response Times"; Proc. IFAC Workshop on Distributed Computer Control Systems, Toledo, Spain, Sept. 1994.