CAN-RT-TOP: Real-Time Task-Oriented Protocol over CAN for Analyzable Distributed Applications

Juan López Campos, J. Javier Gutiérrez, and Michael González Harbour

Departamento de Electrónica y Computadores, Universidad de Cantabria, 39005-Santander, SPAIN
{lopezju,gutierjj,mgh}@unican.es

Abstract

This paper presents the design and implementation of CAN-RT-TOP (Real-Time Task-Oriented Protocol over CAN), which is a high level protocol over CAN. Although the CAN Bus uses fixed priorities, some standard protocols over CAN assign the priorities to specific nodes by encoding it in the destination node identifier, which produces reduced schedulability. The protocol presented in this paper provides applications the capability of choosing the message priorities that each task wants to use, thus increasing schedulability. It has been implemented on a real-time kernel for embedded systems called MaRTE OS.

1. Introduction[1]

CAN, acronym of Controller Area Network is a serial communication protocol initially developed by BOSCH [1] and currently maintained by CAN in Automotion (CiA) [2], which supports efficiently distributed control tasks with real-time characteristics and strict reliability requirements. The CAN bus can be used to send and receive small control messages of up to 8 data bytes at speeds up to 1 Mbps even in noisy ambients. Each message has an associated identifier or fixed priority that is used to arbitrate access to the bus. Each bit of the message priority is arbitrated in sequence, so if all priorities are unique there are no collisions in accessing the bus.

To access the CAN bus from the application it is useful to use a high level protocol such as CANOpen [2], CANKingdom [3], OSEK-COM [4], or SDS [5]. In CANOpen there is a restriction that the priority of the message is assigned in a configurable but fixed manner to the destination communication node and function. This policy is usually appropriate for device-oriented communication. But in task-oriented communication, when a node is a processor executing many concurrent tasks, there may be different timing requirements imposed on the messages which should not depend on their destination node. Because in CANOpen, all the messages that are sent to the same destination object have to share the same priority there is a severe deviation from the optimum priority assignment, with the corresponding loss in schedulability.

CANKingdom has a much more flexible way of assigning priorities, but is a relatively complex protocol that requires a central master and an initialization phase to configure the network.

In OSEK-COM it is possible to assign deadlines to messages, but there is no explicit mechanism to assign priorities. In distributed transactions better results can be obtained by having the choice to freely assign priorities to the messages and tasks of the transaction [8].

The SDS high-level protocol associates the priority of the messages with the logical addresses of the CAN controller chips. This restriction does not allow a flexible priority assignment, and is precisely what we want to avoid.

With CAN-RT-TOP we have designed a simple protocol and its corresponding driver that is task-oriented, in the sense that multiple concurrent tasks executing in the same node can choose the priorities of the messages they send. In fixed priority scheduling it is a well established concept that priority is related to the urgency of an activity, not to its importance. With this approach we are able not only to communicate with sensors or actuators, but also to communicate among control tasks, as long as there are no requirements for high volumes of data that would not be appropriate for the CAN bus. The configuration is done in a static way to avoid the complexity of a dynamic initialization phase.

In the industrial automation applications for which the protocol is designed there is no need for message broadcast, but this would be a fairly simple extension for the future.

We have implemented this real-time communication protocol in MaRTE OS [6], which is a real-time kernel for embedded systems on which our research group has been working in the last few years. The protocol proposed in this work is, from a functional point of view, similar to RT-EP [7], a real-time protocol over ethernet also included in

1. This work has been funded in part by the *Comisión Interministerial de Ciencia y Tecnología* (CICYT) of the Spanish Government under grant number TIC2002-04123-C03-02 (TRECOM), and by the IST Programme of the European Commission under project IST-2001-34820 (FIRST).

MaRTE OS. Our measurements show that when the requirements for communications imply short messages, the CAN bus outperforms the RT-EP ethernet protocol.

The paper is organized as follows. Section 2 describes the CAN-RT-TOP protocol. In Section 3 we give some details about how it is implemented. Section 4 discusses the usage in a distributed application. Finally, Section 5 gives our conclusions.

2. CAN-RT-TOP Description

The objective of CAN-RT-TOP is to make a simplified real time protocol over CAN in which the applications involved can choose the priorities of their messages independently of the destination CAN node device. When designing the protocol, we decided to include the following information to be sent with each message:

- *Priority*: It is global priority used to encode the urgency of the message. It contains only a priority value, independent of the destination node and function. Because CAN priorities are ordered inversely to their integer value and user priorities go the opposite way, we have to perform the required mapping of priorities.
- *Destination Node*: When sending a message to a destination object it is important to specify the destination node, so that the CAN controller chips can filter out those messages that are not addressed to them. This field must have at least one distinct value for each CAN controller chip in the system.
- *Communication Channel*: Because at the receiving node there may be multiple concurrent tasks receiving messages, we need to identify the destination task. We do this by creating the concept of a communication channel. A channel would be a logical endpoint for the communication, so that the sender can specify the channel and the receiving task can choose the channel from which it wants to read messages.

We must determine where to place the three protocol information fields inside a CAN frame. These frames have four relevant fields for us, as shown in Figure 1.

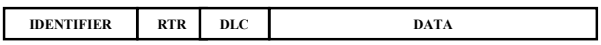

Figure 1. CAN frame format

The *Identifier* is used for the bus arbitration and has 11 or 29 bits, respectively in the standard or extended frame format. The *RTR* bit is used to request retransmission. The *DLC* bits are used to define the amount of data bytes of the *Data* field. Although some protocol information like the communication channel could be sent in the *Data* field, the amount of space for the user messages is so small that we decided to include all the protocol information inside the *Identifier* field. This also helps in configuring the identifiers, because there is a requirement that all identifiers are different, so that it is not possible to simultaneously try to transmit two messages with the same identifier.

Because arbitration in the CAN bus is done bit by bit beginning from the most significant bit of the identifier, it is important to place the *Priority* information in the most significant part.

Placement of the *Destination Node* information depends on the filtering capabilities of the CAN controller chips. By filtering out the messages that are not addressed to a particular controller we reduce the CPU overhead because we only accept messages that are addressed to that particular node. The filtering service depends on the manufacturer and is not standardized. In the chip we have used, Philips SJA1000, the filtering service is applied to all or several of the identifier bits, depending on the operating mode. In the most restrictive mode, the mask can be placed only for the 8 most significant bits.

The *Communication Channel* information does not need to be placed inside the range of bits where the CAN filtering service is applied. So the preferable order in the CAN-RT-TOP fields is as shown in Figure 2: *Priority*, *Destination Node*, and *Communication Channel*.

Identifier										
Bit number	11	10	9	8	7	6	5	4	3	2	1
Field	Message Priority				Destination Node			Communication Channel			

Figure 2. Fields in a CAN Frame with 11 bits identifier

The number of bits reserved for each field can be determined by the user via a configuration process in order to adapt the protocol to the specific application requirements. If the configuration with 11 bits identifiers is chosen we must make the best use we can out of them.

3. CAN-RT-TOP Implementation

We have implement CAN-RT-TOP in a character driver in MaRTE OS [6]. In order to make the protocol completely independent of the CAN controller chip used, the driver is divided in two well separated parts: one containing the protocol itself, and another one containing the functions needed to manage the CAN controller chip. These two parts are connected with an abstract interface, so in case another CAN controller chip would be used it would be only necessary to change this small part of the driver. This makes the protocol completely portable between CAN controller chips.

The driver has six functions available for the user: *open*, *close*, *read*, *write*, *read_bus_status* and *set_read_channel*. The four first functions are regular POSIX primitives,

while the last two are implemented via device control calls (*ioctl*). All of them are reentrant calls.

Inside the driver and as part of the protocol implementation, we have implemented two types of data structures for storing the messages in priority order: one is based on binary heaps, and the other one is an array of FIFO circular queues, one for each priority. Each of these data structures has its own advantages and drawbacks. The array of FIFO queues is faster when the number of priorities is not too high, but it needs much more memory. Binary heaps take less space but are slower in most practical cases. The particular choice of data structure is set in the driver configuration information. Both data structures are implemented such that equal-priority messages are dequeued in FIFO order.

Both data structures are used to implement the concept of the *Communication Channel*. There are two types of channels for every CAN controller chip, one for transmitting and one or more for receiving messages. Each channel is implemented by one priority queue.

The *open* call creates a file description data structure for the driver, and returns a file descriptor used for all the subsequent operations on the driver. The *close* call destroys that data structure.

The *write* call is used when a task wants to send a message through the CAN bus. In the case when the transmission channel is full, this function blocks the calling thread until it can write the message to the transmission channel. Messages are stored in priority order. Figure 3 shows a diagram with the elements involved in the *write* function call.

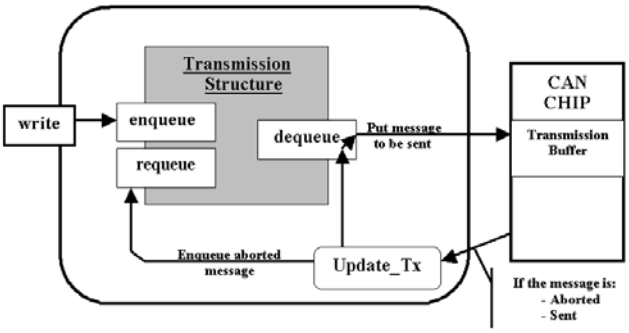

Figure 3. Transmission of messages in CAN-RT-TOP

The standard POSIX *read* function does not have parameters for specifying the communication channel from which it wants to read messages. For this reason, a task using the *read* call must configure the reception channel from which it wants to read using *set_read_channel*. Once configured, the task will read messages from that channel; besides, the configuration can be changed whenever it is necessary. It would have been possible to use a different file for each channel, but because the number of channels could be very large, this solution would lead to excessively large file tables. The *read* call gets the message with the highest priority value from the reception channel associated with the calling thread. If no messages are available, the calling thread is blocked until a message arrives. Figure 4 shows the elements involved in the *read* function call.

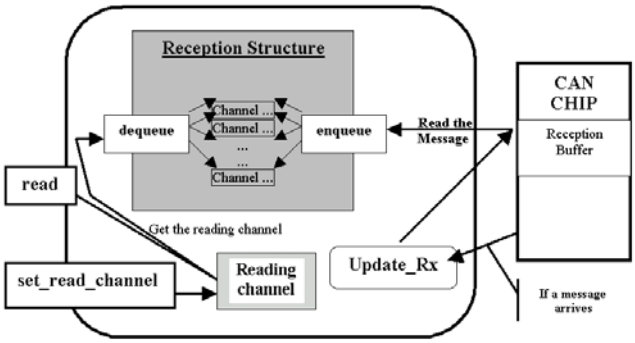

Figure 4. Reception of messages in CAN-RT-TOP

The *read_bus_status* call may be used to find out whether the associated driver has had any problem such as an error or overrun in the CAN controller chip or an overrun in the reception channel (more messages arrived than those expected when the system was configured).

When transmitting a message through the CAN controller chip, the problem known as inner priority inversion [13] may occur. Our chip (Phillips SJA1000) has only one transmission buffer, and the solution that we have implemented to solve this problem is to replace the message stored in the transmission buffer inside the controller chip with a higher priority message whenever it is necessary. This can be done using the chip's message abort feature: we can abort the message that is waiting at the transmission buffer, replace it with the new highest priority message, and requeue the old message into the transmission queue, as shown in Figure 3.

4. Application to a General-Purpose Distributed Platform

The protocol presented in this paper is fully implemented and available to be used by any application. In our group, we are introducing it as a communication protocol available in RT-GLADE [8] which is an implementation of the Distributed System Annex of ADA 95 based on GLADE [10], and optimized for achieving a better real-time behavior. An application using this platform can be modeled and analyzed [9][11].

To be able to use CAN-RT-TOP in RT-GLADE, or in other applications, it is necessary to add a partitioning layer that allows us to transmit messages larger than 8 bytes. To accomplish this we have implemented in RT-GLADE a general packet partitioning layer. This partitioning layer is also available to be used by our ethernet protocol RT-EP.

We need to define the information that is required to be transmitted with each packet to ensure the correct message recomposition. Because of the small amount of bytes that we have in CAN for each packet, this information should be restricted in size as much as possible.

One of the CAN requirements is that identifiers of simultaneously enqueued packets are different, to avoid collisions. We meet this requirement by assigning at configuration time different CAN identifiers to each task involved in message transmission. As a consequence, it is not necessary to explicitly identify which task is sending the message.

Since the CAN bus does not reorder the transmitted packets and the data structures that we have implemented in the CAN driver dequeue packets of the same priority in FIFO order, the only information that we need for the partitioning layer is for identifying the packets that conform the original message. In CAN, frames are not missed except in unusual circumstances [12], but because we want to use this layer with other networks also, we want to impose the additional requirement of being able to detect errors caused by missing packets in the message sequence. For this reason, we will include the following information in each packet:

- *Type*: Single, first, intermediate or last packet.
- *Sequence number*: a modular number of fixed size that lets us distinguish packets from different messages.
- *Number of remaining packets*: a modular number of fixed size that lets us distinguish whether the received packets are contiguous or not.

This information can be coded in just one byte, as shown in Figure 5. This byte would be the first user byte in the case of the 11-bit identifier, and would be the least significant bits of the 29-bit identifiers.

Bit number	8	7	6	5	4	3	2	1
Field	Type		Seq. Num.		Remaining packets			

Figure 5. Partitioning information byte

5. Conclusions

We have presented a high level protocol over CAN which is task oriented rather than device oriented, in order to be able to use CAN for general purpose communications. In this protocol, tasks can freely choose the priorities of their messages. Due to the characteristics of the CAN bus, the protocol is useful for those systems in which messages are not too large.

The protocol is fully implemented under MaRTE OS, and is being introduced in a general-purpose distributed programming environment called RT-GLADE. To overcome the limitation of only eight bytes per message, a packet partitioning layer has been implemented.

The protocol is easily configurable and is designed to ease portability among different CAN controller chips via a simple interface.

REFERENCES

[1] CAN Specification Version 2.0. 1991, Robert Bosch GmbH, Postfatch 30 02 40, D-70442 Stuttgart.

[2] CAN in Automation Web Page: `www.can-cia.org`

[3] CanKingdom High level CAN protocol web page : `www.cankingdom.org`

[4] OSEK-COM. “OSEK/VDX Communication. Version 3.0.1”, January 29, 2003.

[5] Smart Distributed System. “Application Layer Protocol Specification. Version 2.0” by Honeywell. April 6, 1999.

[6] M. Aldea and M. González. “MaRTE OS: An Ada Kernel for Real-Time Embedded Applications”. Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, Lecture Notes in Computer Science, LNCS 2043, May, 2001.

[7] J.M. Martínez, M. González Harbour, and J.J. Gutiérrez. “RT-EP: Real-Time Ethernet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel”. Proceedings of the 2nd Intl. Workshop on Real-Time LANs in the Internet Age, RTLIA 2003, Porto (Portugal), July 2003.

[8] Juan López Campos, J.Javier Gutiérrez and Michael González Harbour. “The Chance for Ada to Support Distribution and Real Time in Embedded Systems. Ada-Europe 2004, Palma de Mallorca, Spain. June 2004.

[9] M. González Harbour, J.J. Gutiérrez, J.C. Palencia and J.M. Drake. “MAST: Modeling and Analysis Suite for Real-Time Applications”. Proceedings of the Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001

[10] L. Pautet and S. Tardieu. “GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems”. Proc. of the 3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC'00), Newport Beach, USA, March 2000.

[11] K.Tindell, A. Burns and A. Wellings. “Calculating Controller Area Network (CAN) message response times”. Proc. of the 1994 IFAC Workshop on Distributed Computer Control Systems (DCCS), Toledo, Spain.

[12] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. “Fault-Tolerant Broadcasts in CAN”.In Digest of Papers, The 28th International Symposium on Fault-Tolerant Computing Systems, pages 150-159, Munich, Germany, June 1998. IEEE.

[13] Michiel Van Osch and Scott A. Smolka. “Finite-State Analysis of the CAN Bus Protocol”. Proceedings of Sixth IEEE International Symposium on High Assurance Systems Engineering (HASE 2001), October 2001.