Analysing Real-Time Communications: Controller Area Network (CAN)*

K. W. Tindell†, *H. Hansson*
Department of Computer Systems,
University of Uppsala, Sweden

A. J. Wellings
Department of Computer Science,
University of York, England

Abstract

The increasing use of communication networks in time critical applications presents engineers with fundamental problems with the determination of response times of communicating distributed processes. Although there has been some work on the analysis of communication protocols, most of this is for idealised networks. Experience with single processor scheduling analysis has shown that models which abstract away from implementation details are at best very pessimistic and at worst lead to unschedulable system being deemed schedulable. In this paper, we derive idealised scheduling analysis for the CAN network, and then study two actual interface chips to see how the analysis can be applied.

1. Introduction

One of the fundamental difficulties in engineering hard real-time systems is the development of *analysis* to bound the timing behaviour of the system. Much work in recent years has been developing this analysis for a run-time dispatching algorithm known as *fixed priority pre-emptive scheduling*. This work has recently addressed the scheduling of messages on shared broadcast buses [6], and in particular token-passing and 'priority pre-emptive' buses. The work makes certain assumptions about the ideal behaviour of the interface between the host processor and the communications adapter for these buses. However, a given implementation may not meet these assumptions, and so recent research has examined a particular bus protocol and implementations from a number of different manufacturers. This paper reports on this analysis, and shows how small differences in the implementation of an interface can have dramatic effects on the worst-case timing performance of messages.

The real-time bus we examine in this paper is called *Controller Area Network* (CAN) [1]. In particular we examine in detail two interface chips: the 82527 controller from Intel, and the 82C200 controller from Philips. We show how the Intel controller has a very much better worst-case timing performance than the Philips controller.

CAN is a broadcast bus designed to operate at speeds of up to 1 Mbit/sec. Data is transmitted in *messages* containing between 0 and 8 bytes of data. An 11 bit number is associated with each message. The identifier is required to be unique, in the sense that two simultaneously active messages originating from different sources must have distinct identifiers (typically, an identifier corresponds to a particular type of message from a specific source). The identifier serves two purposes: (1) assigning a priority to the message, and (2) enabling receivers to filter messages. A station filters messages by only receiving messages with particular bit patterns (typically using comparitors and mask registers). Thus CAN messages have no explicit destination, since any station with an appropriate filter may receive a message.

The use of the identifier as priority is the most important part of CAN with respect to real-time performance. Like Ethernet, CAN is a collision-detect broadcast bus, but takes a much more systematic approach to contention. The identifier field of a CAN message is used to control access to the bus after collisions by taking advantage of certain electrical characteristics of a CAN bus: if multiple stations are transmitting concurrently and one station transmits a '0', then all stations monitoring the bus will see a '0'. Conversely, only if all stations transmit a '1' will all processors monitoring the bus see a '1'. In effect, the CAN bus acts like a large AND-gate, with each station able to see the output of the gate. This behaviour is used to resolve collisions: each station waits until bus idle (as with Ethernet). When silence is detected, each station begins to transmit the highest priority message held in its output queue whilst monitoring the bus. The identifier is the first part of the message to be transmitted; the identifier is transmitted from most-significant to least-significant bit. If a station transmits a recessive bit ('1'), but monitors the bus and sees a dominant bit ('0'), then it stops transmitting since it knows that the message it is transmitting is not the highest priority message in the system. Because identifiers are deemed unique within the system, a station transmitting the last bit of the identifier without detecting a collision must be transmitting the highest priority queued message, and hence can start transmitting the body of the message.

The CAN message format contains 47 bits of protocol control information (the identifier, CRC data,

*This work was supported in part by the U.K. EPSRC, grant number 06R00456
†Department of Computer Systems, P.O. Box 325, S-751 05 Uppsala, Sweden (E-mail: ken@docs.uu.se)

acknowledgement and synchronisation bits, *etc.*). The data transmission uses a bit stuffing protocol which inserts a 'stuff bit' after five consecutive bits of the same value.

Because the number of inserted stuff bits depends on the bit pattern of a message, a given message type can vary in size, *e.g.* a CAN message with 8 bytes of data (and 47 control bits) is transmitted with between 0 and 19 stuff bits.

2. Communications model & notation

We define a message to be either a *data message*, or a *remote* transmission *request message*. A message has a size (between zero and eight bytes), and an identifier (as described earlier). The set of all messages in the system is denoted *messages*.

In a typical system, a message is queued by an application task. We assume that each task is invoked repeatedly (a task is said to have *arrived* when invoked by some action). Each task has a minimum inter-arrival time termed the *period*. Note that the period is a minimum time between subsequent arrivals, rather than a strict fixed interval. If the message queued by a given task is potentially sent each time the task is invoked, then the message inherits a period equal to the period of the task. We denote as T_m the period of a given message m.

A given task i has a worst-case response time, denoted R_i, which is defined as the longest time between the arrival of a task and the time it completes some bounded amount of computation. Existing analysis for single processors is able to determine this worst-case response time.

In general, the queuing of a message can occur with *jitter* [2] (variability in queuing times). Correct analysis requires that jitter be taken into account. Queuing jitter can be defined as the difference between the earliest and latest possible times a given message can be queued.

As with the period, the jitter of a given message m may be inherited from the sender task. For an application task i (with worst-case response time R_i) sending message m, this queuing window is no more than R_i in duration (*i.e.* the difference between the earliest and latest queuing times of the message). The jitter of a given message m is denoted J_m. In any realistic system all messages will have some queuing jitter.

The worst-case response time of a given message m is denoted R_m and defined as the longest time taken for the message to reach the destination stations, measured relative to the arrival time of the sender task.

The longest time taken to transmit a given message m we denote as C_m. For an eight byte message (the largest message permitted with CAN) transmitted on a 1 Mbit/sec network, C is 130μs (64 bits for the data, 47 bits of overhead — CRC and identifier fields, *etc.* — and up to 19 stuff bits). Figure 1 illustrates the timing of a CAN message.

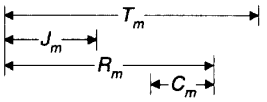

Figure 1: Timing model for CAN messages

3. Basic processor scheduling theory

Scheduling messages on a CAN bus is analogous to scheduling tasks by fixed priorities. It is possible to take existing analysis and apply it to CAN messages. We therefore take a brief detour into scheduling theory for fixed priority scheduling of tasks on a single processor.

Audsley *et al* [2] and Burns *et al* [3] show how the analysis of Joseph and Pandya [4] can be updated to include blocking factors introduced by periods of non-pre-emption, release jitter, and accurately take account of a task being non-pre-emptive for an interval before termination. The following equations represent this analysis:

$$R_i = J_i + w_i + C_i \tag{1}$$

where w_i is given by:

$$w_i^{n+1} = B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n + J_j + \tau_{res}}{T_j} \right\rceil C_j \tag{2}$$

Where $hp(i)$ is the set of tasks of higher priority than task i, C_i is the worst-case computation time required by a given task i, and T_j is the period of a given task j. B_i is the blocking factor of task i (a bound on the time that a lower priority task can execute and prevent the execution of task i); the priority ceiling protocol [5] controls this 'priority inversion' and defines how B_i can be computed. τ_{res} is the resolution with which we measure time. On CAN bus we deal with time units as multiples of the bit-time, which we denote as τ_{bit}; with a 1 Mbit/sec bus this is equal to 1 μs.

J_i is the release jitter of task i, analogous to the queuing jitter of a message. Note that if a task is invoked by an incoming message, then the task inherits a release jitter (and period) from the message (in just the same way as a message inherits a queuing jitter from a sending task); this is known as *attribute inheritance*, and leads to an approach known as *holistic scheduling* (see [7] for a full treatment).

The feasibility of a given task can be trivially assessed by comparing the worst-case response time of the task against its deadline. Note that the deadline of a given task i, denoted D_i is assumed to be less than or equal to T_i. Another assumption is that a task cannot voluntarily suspend itself (and hence the processor cannot be idle when tasks have work to do).

Equation 2 describes a recurrence relation, where the $(n + 1)$th approximation to the value of w_i is found in terms of the nth approximation, with the first approximation set to zero. A solution is reached when the $(n + 1)$th approximation equals the nth.

Having introduced this analysis we can apply it to CAN bus scheduling. We do this by first deriving ideal CAN analysis, and then discussing how the analysis is affected by the behaviour of implemented hardware. We discuss two implemented CAN controllers: the Intel 82257 and the Philips 82C200.

4. Analysis for ideal CAN

In this section we will derive scheduling analysis to bound the worst-case response time of a given message. The analysis of the previous section can be applied to ideal CAN by the analogy between task scheduling and message scheduling: a task is released at some time (*i.e.* is placed in a priority-ordered queue of runnable tasks), and contends with other tasks (both lower and higher priority tasks) until it becomes the highest priority runnable task.

Because of the operation of the priority ceiling protocol, a task need only contend with at most one lower priority task. In addition, it contends with all higher priority tasks until these have all completed and the processor is freed. With the model of Burns *et al* [3], the task is then dispatched and runs until completion. Upon completion it is returned to the waiting queue until next made runnable.

The same behaviour holds for CAN messages: a message is queued at some time, and contends with other messages until it becomes the highest priority message. It commences transmission, and is transmitted without interruption until completion. Note that this assumes that the bus cannot become idle between the transmission of messages if there are pending messages (this is analogous to the assumption that a task must not voluntarily suspend itself). This assumption does not hold with the Philips 82C200 controller, and we examine the ramifications of this in section 6.

The worst-case response time of a queued data message, measured from the arrival of the queuing task to the time the message is fully transmitted, is given from the analogy (equation 1) by:

$$R_m = J_m + w_m + C_m \tag{3}$$

J_m is the queuing jitter of message m, inherited from the worst-case response time $R_{sender(m)}$ (where $sender(m)$ denotes the task queuing message m).

The term w_m represents the worst-case queuing delay — the longest time between placing the message in a priority-ordered queue, and the message commencing transmission. By analogy with equation 2:

$$w_m = B_m + \sum_{\forall j \in hp(m)} \left\lceil \frac{w_m + J_j + \tau_{bit}}{T_j} \right\rceil C_j \tag{4}$$

where the term B_m is the worst-case blocking time of message m, and is analogous to the blocking factor defined by the analysis of the priority ceiling protocol. B_m is equal to the longest time taken to transmit a lower priority message, and given by:

$$B_m = \max_{\forall k \in lp(m)} (C_k) \tag{5}$$

$lp(m)$ is the set of messages in the system of lower priority than message m. If m is the lowest priority message then B_m is zero (just as the lowest priority task has a blocking factor of zero with the priority ceiling protocol).

τ_{bit} is the time taken to transmit a bit on the bus and $hp(m)$ is the set of messages in the system of higher priority than message m.

C_m is the longest time taken to transmit message m. As mentioned earlier, CAN has a 47 bit overhead per message, and a stuff width of 5 bits. Only 34 of the 47 bits of overhead are subject to stuffing, so C_m can be defined by:

$$C_m = \left(\left\lfloor \frac{34 + 8s_m}{5} \right\rfloor + 47 + 8s_m \right) \tau_{bit} \tag{6}$$

where s_m is the number of data bytes in the message. Equation 4 above can be solved in the same way as equation 2.

We next examine how the analysis copes with the implementation details of different controllers, starting with the Intel 82527.

5. Real-time behaviour of the 82527

The queuing of messages in the Intel 82527 is undertaken in the controller and interfaced to the host processor via dual-ported RAM. The intention is to map permanently message identifiers to memory locations (termed *slots*), so that both outgoing and desired incoming messages are assigned unique slots.

A slot is tagged with a message identifier, and marked as an incoming or outgoing slot. If a message is received with the same identifier as a slot marked as incoming then the message contents are stored in that slot (the slot also contains an interrupt enable flag so that an interrupt can be raised when the message arrives). If the host processor wishes to initiate the transmission of the message then it is able to mark the message as ready for transmission.

Because of hardware limitations, only 15 slots are available for outgoing and incoming messages (instead of the ideal 2032 — the full range of CAN identifiers). However, these 15 slots can be programmed to map to any CAN identifier. The controller will transmit messages in order of slot number, rather than the message identifier, and therefore it is important that the messages are allocated to the slots in identifier order. It should be noted that in most envisaged automotive systems, 15 messages per station is sufficient [8].

There is also a dedicated double-buffered receive buffer: when a message has been received in the controller without errors, a "message received" interrupt may be raised on the host processor. If the identifier of the message does not match the identifier in one of the slots in the controller then the interrupt handler must copy the contents of the message from the buffer and store it in main memory. The handler then issues a "removed message" signal to the controller, indicating that the receive buffer is free. This is needed because the receive buffer is double buffered: while the host processor is removing data from one buffer, the controller may be placing data in the other buffer. The controller needs to synchronise with the host processor in order to place data in a free buffer.

There is an implicit deadline on handling the "message received" interrupt: if the host processor fails to remove the data and signal "removed message" before the controller has received the subsequent message then any further incoming messages may be lost (the smallest time between two successive messages is 47 τ_{bit}).

In many ways the dual-ported memory approach is an elegant way of implementing a controller, but one drawback is that there is an implicit restriction on message deadlines: a message cannot be queued if the previous queuing of the same message has not yet been transmitted. Therefore, we must have $D_m \leq T_m$ (an assumption also made by the scheduling analysis in this paper).

Apart from the limitations discussed, the Intel 82527 controller behaves as an ideal CAN controller with respect to the analysis derived in this paper.

6. Real-Time behaviour of the 82C200

In this section we discuss the behaviour of the Philips 82C200 CAN controller, and show its worst-case real-time properties are poor (space limitations preclude the development of analysis for this controller).

The Philips controller is a simple controller, with two message buffers on-chip: a single 10 byte transmission buffer, and a 10 byte double-buffered receive buffer. The controller is typically interfaced to the processor as a memory mapped I/O device, and can raise two interrupts: "message received", and "message sent". The controller accepts three signals from the host processor: "send message", "abort message", and "removed message". The controller requires messages to be held on the host processor, and software drivers to copy the messages from the processor to the controller when appropriate.

To send a message, the host processor fills the transmit buffer with up to eight bytes of data, the identifier of the message, and some control bits, and then sends a "transmit message" signal to the controller. We denote the longest time to do this as τ_{copy}. The controller attempts to transmit the message according to the CAN protocol; when the message has been sent, a "message sent" interrupt is raised on the host processor.

The reception of messages is very similar to the Intel 82527 controller: when a message has been received in the controller without errors a "message received" interrupt is raised on the host processor and the interrupt handler must copy the 10 bytes of message data from the controller and store it in main memory.

The signal "abort message" is to aid in the writing of software device drivers for pre-emptive queuing. Without the signal, the real-time performance of the controller would be very poor indeed. Consider the situation where there is a low priority message in the transmit buffer of the controller, and a high priority message has just been queued by the host processor software. If the host processor were unable to remove the low priority message, then the high priority message would blocked until the low priority message is sent. The low priority message will only be sent when all other higher priority traffic on the bus has finished; this could be very long[‡].

Instead of succumbing to this problem, the device driver should abort the transmission of the low priority message, and copy the high priority message to the transmit buffer. The controller will only abort the message if it has not yet begun transmission. This is a sensible approach, since if the low priority message has begun transmission then there will be only a short delay (equal to the transmission time of the message) before the transmission buffer is freed.

There remains a major problem with the management of the transmission buffer: the time between "message sent" and the host processor copying the next message to the transmit buffer is non-zero (although short if the host processor is fast). In this short interval the bus could be claimed by a low priority message from another station and defer the transmission of the newly copied message. This problem also occurs when a message is pre-empted: the short interval between a lower priority message being aborted, and the higher priority message being copied into the buffer, releases the bus to low priority traffic.

For every pre-emption (*i.e.* when a message is aborted, and replaced by a higher priority message) in an interval, the bus may potentially be claimed twice by lower priority traffic at other stations: once when the higher priority message pre-empts, and once when the message has been transmitted.

To illustrate this, consider the following scenario: a message *M* is to be sent from a given station. Also sent from this station are a high priority message *H* and a low priority message *L1*. Other stations also have low priority traffic to send (messages *L2*, *L3*, and *L4*). In this scenario, message *M* can be delayed four times by lower priority

[‡]The software drivers supplied by Motorola for the 82C200 appear to exhibit this priority inversion problem.

messages whilst being pre-empted just once. This is solely a result of the buffer management mechanism.

The first delay occurs when message M is queued: as mentioned earlier, the 82C200 controller is not able to abort a message if the message has begun transmission. Therefore message M can be delayed by $L1$. After the message has been sent, the host processor copies message M to the transmission buffer (taking at most τ_{copy}). In this time the bus is released and may become idle, or may be claimed by lower priority messages from other stations. When message M has been copied to the buffer, and is ready for transmission, it may be delayed by a lower priority message that has just started transmission from another station ($L2$). Just before message M starts transmitting, a higher priority message H can pre-empt M: the 82C200 controller aborts message M, and copies the higher priority message to the transmission buffer. Again, the bus is released, and again lower priority message can be transmitted ($L3$), delaying both message H and message M. When message H has been transmitted the host processor copies message M back to the transmission buffer. Again, the bus is released, and again message M can be delayed (by $L4$).

It is straightforward to bound the delays due to this priority inversion, and the delays due to copying messages. This priority inversion can be very large, and lead to very poor worst-case performance of the controller. The following table details a set of messages based on the above scenario. They conform to the 'rate monotonic' model of deadlines equal to periods.

Message	*T*	*D*	*C*	*util*^{n}
H	605	605	47	7.8%
M	610	610	47	7.8%
L1	100000	100000	130	0.13%
L2	100000	100000	130	0.13%
L3	100000	100000	130	0.13%
L4	100000	100000	130	0.13%

In the above table, all times are in microseconds. Messages are assumed to be queued with zero jitter.

A small value for τ_{copy} is assumed: large enough to release the bus to lower priority messages when copying a message to the transmission buffer, but not large enough to form a significant part of the response time of a message (in practice, such a small value would be unattainable).

The example message set is unschedulable: in the scenario described, we find that the response time of message M is 614 µs. The bus utilisation in this example is just under 16%. By comparison, the worst-case response time of M with the Intel controller is 224 µs. It is possible to find unschedulable scenarios with bus utilisations as low as 11%. Clearly using the Philips controller could lead to very poor resource utilisation.

Note that in the situation where there is a large amount of low priority 'soft' real-time traffic on the bus, the impact of higher priority traffic on lower priority traffic sent from the same station is at least trebled when compared to the ideal CAN behaviour (and when compared to the behaviour of the Intel 82527), and that worst-case response times will therefore be very much larger.

7. Conclusions

This paper has derived scheduling analysis for the CAN communication protocol. In particular, it has bounded message response times for ideal behaviour. It has shown that this ideal model can be inadequate, and that it is necessary to consider the actual behaviour of the controller technology. Two controllers have been considered: the Intel 82527 and the Philips 82C200. The Intel controller performs ideally, whereas the Philips controller potentially leads to a large amount of priority inversion (and hence requires the ideal analysis to be updated). This paper has considered only message delivery, but another aspect of real-time communication is the processing overheads incurred when sending and receiving message. Analysis bounding such overheads can be developed (see Tindell *et al* [6] for a full discussion).

8. References

[1] "Road Vehicles — Interchange of Digital Information — Controller Area Network (CAN) for High Speed Communication", ISO/DIS 11898 (Feb. 1992).

[2] Audsley, N., Burns, A., Richardson, M., Tindell, K. and Wellings, A., "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Engineering Journal* 8(5), pp. 285-292 (Sept. 1993).

[3] Burns, A., Nicholson, M., Tindell, K. and Zhang, N. "Allocating and Scheduling Hard Real-Time Tasks on a Point-to-Point Distributed System", *Proc. Workshop on Parallel and Dist. Real-Time Syst.*, pp. 11-20 (Apr. 1993)

[4] Joseph, M. and Pandya, P., "Finding Response Times in a Real-Time System," *Computer J.* 29(5), pp.390-395 (Oct. 1986)

[5] Sha, L., Lehoczky, J. P., Rajkumar, R., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Trans. on Computers*, 39(9), pp. 1175-1185 (Sept. 1990)

[6] Tindell, K., "Analysis of Hard Real-Time Communications," YCS 222, Dept. Computer Science, Univ. of York (1994) (to appear in *Real-Time Systems*)

[7] Tindell, K., and Clark, J., "Holistic schedulability analysis for distributed hard real-time systems", *Microprocessing and Microprogramming*, 40(2-3), pp. 117-134 (Apr. 1994)

[8] Tindell, K., Burns, A., "Guaranteed Message Latencies for Distributed Safety Critical Hard Real-Time Networks," YCS 229, Dept. Computer Science, Univ. of York (1994).