CALCULATING CONTROLLER AREA NETWORK (CAN) MESSAGE RESPONSE TIMES

K. TINDELL, A. BURNS, and A. WELLINGS

University of York, Department of Computer Science, York, YO1 5DD, England

Abstract: Controller Area Network (CAN) is a well designed communications bus for sending and receiving short real-time control messages at speeds of up to 1Mbit/sec. One of the perceived drawbacks to CAN has been the inability to bound accurately the worst-case response time of a given message (*i.e.* the longest time between queueing the message and the message arriving at the destination processors). This paper presents analysis to bound such response times, including the costs of error handling and re-transmission.

Key Words: real-time systems; real-time communications; scheduling theory; scheduling analysis; distributed systems

1. INTRODUCTION

The Controller Area Network (CAN) [3] is a well designed communications bus for sending and receiving short real-time control messages. The bus is designed to connect control systems over a small area (such as automobiles), operating in a noisy environment at speeds of up to 1Mbit/sec. One of the perceived problems of CAN is the inability to bound the response times of messages. To show how this problem can in fact be easily solved, we apply analysis developed for fixed priority pre-emptive real-time processor scheduling [1, 6, 7] to the problem of message scheduling on a CAN bus. Before we proceed further we describe briefly the architecture of CAN, and make some general observations and assumptions about the implementation.

CAN is a broadcast bus where a number of processors are connected to the bus via an interface (Fig. 1).

A data source is transmitted as a *message*, consisting of between 1 and 8 bytes ('octets'). A data source may be transmitted periodically, sporadically, or on-demand. So, for example, a data source such as 'road speed' could be encoded as a 1 byte message and broadcast every 100 milliseconds. The data source is assigned a unique *identifier*, represented as an 11 bit number (giving 2032 identifiers — CAN prohibits identifiers with the seven most significant bits equal to '1'). The identifier servers two purposes: filtering messages upon reception, and assigning a priority to the message.

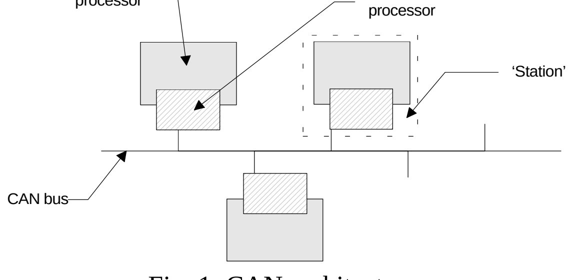

Fig. 1: CAN architecture

A station on a CAN bus is able to receive a message based on the message identifier: if a particular host processor needs to obtain the road speed (for example) then it indicates the identifier to the interface processor. Only messages with desired identifiers are received and presented to the host processor. Thus in CAN a message has no destination.

The use of the identifier as priority is the most important part of CAN regarding real-time performance. In any bus system there must be a way of resolving contention: with a TDMA bus, each station is assigned a pre-determined time slot in which to transmit. With Ethernet, each station waits for silence and then starts transmitting. If more than one station tries to transmit together then they all detect this, wait for a randomly determined time period, and try again the next time the bus is idle. Ethernet is an example of a carrier-sense broadcast bus, since each station waits until the bus is idle (*i.e.* no carrier is sensed), and monitors its own traffic for collisions. CAN is also a carrier-sense 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.

With CAN, if multiple stations are transmitting concurrently and one station transmits a ‘0’ bit, 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 CAN terminology, a ‘0’ bit is termed *dominant*, and a ‘1’ bit is termed *recessive*. 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 queue whilst monitoring the bus. The message is coded so that the most significant bit of the identifier field is transmitted first. If a station transmits a recessive bit of the message identifier, but monitors the bus and sees a dominant bus then a collision is detected. The station knows that the message it is transmitting is not the highest priority message in the system, stops transmitting, and waits for the bus to become idle. If the station transmits a recessive bit and sees a recessive bit on the bus then it may be transmitting the highest priority message, and proceeds to transmit the next bit of the identifier field. Because CAN requires identifiers to be unique within the system, a station transmitting the last bit (least significant 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 (if identifiers were not unique then two stations attempting to transmit different messages with the same identifier would cause a collision after the arbitration process has finished, and an error would occur).

There are some general observations to make on this arbitration protocol. Firstly, a message with a smaller identifier value is a higher priority message. Secondly, the highest priority message undergoes the arbitration process without disturbance (since all other stations will have backed-off and ceased transmission until the bus is next idle). The whole message is transmitted without interruption.

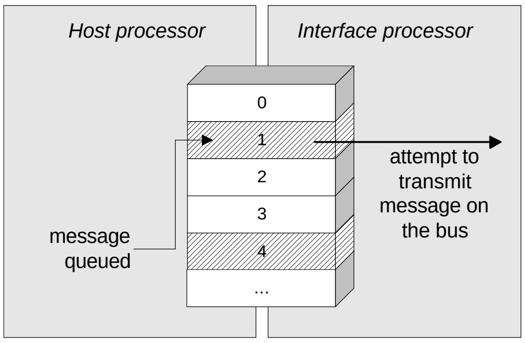

Fig. 2: Interface between host processor and CAN processor

From these observations, the worst-case time from queueing the highest priority message to the reception of that message (*i.e.* the worst-case response time of the message) can be calculated easily: the longest time a station must wait for the bus to become idle is the longest time to transmit a CAN message (we term this delay the *blocking time* of a message). The largest CAN message (8 bytes) takes 130 microseconds to be transmitted (at 1Mbit/sec transmission speed, with a ‘bit stuffing’ width of 5 bits), and hence the blocking time of a CAN message is 130 microseconds. The worst-case response time of the highest priority CAN message is therefore 130 microseconds plus the time taken to transmit the message. For a lower priority message, the worst-case response time cannot be found so easily, leading to the generally perceived problem that only the highest priority message can be guaranteed on CAN. We will give analysis in this paper that bounds the response time of all CAN messages, including the lowest priority message. The existence of this analysis makes CAN eminently suitable as a bus for hard real-time applications.

Before we proceed to develop such analysis we need to discuss briefly how CAN messages are queued in the stations. Fig. 2 depicts a typical interface.

In Fig. 2 the host processor is queueing a message into the slot for identifier ‘1’; the slot for identifier ‘4’ is already occupied with another message. The slots are typically implemented as dual-port memory shared between the processors. The interface processor will attempt to transmit message ‘1’ when the bus next becomes idle. There is no queue of messages for a given identifier: in Fig. 2, if message ‘1’ is being transmitted when another message with the same identifier is queued then the message in the slot is overwritten and destroyed. This is important, since it implies a deadline for a message queued periodically: a given message must be transmitted before the message for the next period can be queued. So, returning to the example of a message containing ‘road speed’, we can see that the message must be transmitted within 100 milliseconds to avoid being overwritten by the contents of the message corresponding to the next measurement. In effect, we have a *deadline* on the transmission of any message: the message must be transmitted before the subsequent message can be queued (of course, we may have a deadline on the message that is much shorter than the period).

2. ANALYSIS OF A SIMPLE CAN MODEL

In this section we develop simple analysis for the CAN model outlined above. In reality, CAN is more complex than described, and later sections will extend the analysis to cover these complexities.

There has been much work in the field of real-time systems analysis recently: at the University of York we have developed analysis for systems where activities are dispatched according to fixed priorities [1, 6, 7, 5]. Because CAN is primarily a priority-based bus, much of this analysis can be applied directly. In this paper we will show the application of the analysis; the reader is referred elsewhere for a more formal derivation of the general theory [1].

Before introducing the analysis we first define some terms. A *message* is a CAN message assigned a unique identifier and consisting of between 1 and 8 bytes of data. A given message is assumed to be queued cyclicly (*i.e.* at intervals, the source of the message queues messages of the same size and with the same identifier). A given message is queued at a station within a *queueing window*, with a minimum interval between subsequent queueing windows (messages do not have to be strictly periodic: a message can be sporadic, but there must be a minimum time between the queueing of the message). This is illustrated in Fig. 3.

The period of a given message m is denoted as T_m. The width of the queueing window for message m (*i.e.* the jitter on the queueing of the message) is denoted J_m. The term b_m defines the number of bytes in the message; C_m denotes the worst-case time taken to physically transmit the message on the bus. This does not including the delays because of contention on the bus; it does include the time taken to transmit the identifier field, other message fields (such as cyclic redundancy checks), and the message data itself. Thus C_m is a function of b_m.

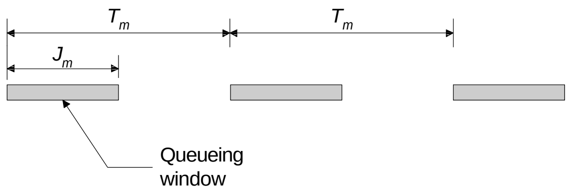

Fig. 3: Periodic message queueing

The blocking time on CAN is defined as the longest time that a message can take to be physically transmitted on the bus. This is equal to C for a message of 8 bytes, and is 130 microseconds for a transmission speed of 1Mbit/sec.

The *worst-case response time* of a given message m is the longest time between the queueing of a message and the time the message arrives at destination stations, and is denoted R_m. The *deadline* of the message is denoted D_m; a message is said to be *schedulable* if and only if:

$$R_m \le D_m$$

We have a restriction on the worst-case response time: a queued message must be sent before the next queueing of the message (we want to prevent the overwriting of a message). Thus we must also have:

$$R_m \le T_m - J_m$$

From this we can see that the message queueing window (*i.e.* the message queueing jitter) must be less than the periodicity of the message. We now develop analysis to determine the worst-case response time of a given message m.

We define the worst-case response time is composed of two delays: the *queueing delay* and the *transmission delay*. The queueing delay is longest time that a message can be queued in a station and be delayed because other higher and lower priority messages are being sent on the bus. We denote this time as t_m. The transmission delay is the time taken to actually send the message on the bus. As we said earlier, this time is denoted C_m (and is a function of b_m, the number of bytes in message m). The worst-case response time is thus defined as:

$$R_m = t_m + C_m \qquad (1)$$

The queueing delay, t_m is itself composed of two times: the longest time that any lower priority message can occupy the bus, and the longest time that all higher priority messages can be queued and occupy the bus before the message m is finally transmitted. Earlier we termed these times the *blocking time*, and denoted it as B. The latter time is termed the *interference*. From earlier scheduling theory [1], the interference from higher priority messages over an interval of duration t is:

$$\sum_{\forall j \in hp(m)} \left\lceil \frac{t + J_j + \tau_{bit}}{T_j} \right\rceil C_j$$

The set *hp*(*m*) is composed of all the messages in the system of higher priority than message *m*. The term τ_{bit} is the time taken to transmit a bit on CAN. Note that the set *hp*(*m*) defines a priority ordering. From other work we know that the optimal priority ordering is *deadline monotonic* [2]. In fact, in the presence of queueing jitter, the optimal ordering is to select priorities on the basis of:

$$D_m - J_m$$

That is, the smaller the value of $D - J$ the higher the message priority [1]. From the above description we can see that the queueing delay is given by:

$$t_m = B + \sum_{\forall j \in hp(m)} \left\lceil \frac{t_m + J_j + \tau_{bit}}{T_j} \right\rceil C_j \qquad (2)$$

We desire the smallest value of t_m satisfying the above equation. Unfortunately, the above equation cannot be re-arranged to give a solution for t_m. However, a recurrence relation can be formed:

$$t_m^{n+1} = B + \sum_{\forall j \in hp(m)} \left\lceil \frac{t_m^n + J_j + \tau_{bit}}{T_j} \right\rceil C_j$$

Because the recurrence relation is monotonically increasing in t_m, we need to start the iteration with a value of t_m^0 that is smaller than the smallest value of t_m satisfying equation 2. A value of zero is suitable, but a better value (*i.e.* one that leads to shorter iteration) is to choose the value of t_n where *n* is a message of higher priority than *m*.

3. EXTENDING THE MODEL: ERROR HANDLING AND 'RTR' MESSAGES

In the previous sections we described briefly the CAN architecture and protocol. However, we made two simplifications: we ignored error handling, and we did not address a special type of message (called Remote Transmission Request messages). In this section we describe a model for error handling, discuss remote transmission request (RTR) messages, and extend the analysis to handle the full CAN model.

CAN has an effective error detection mechanism: an error detected by either the sender of a message, or receiver stations of the message, is signalled to the sender station. The sender then re-transmits the message. In the worst-case, upon detection of an error the recovery process requires the transmission of up to 29 bits (plus the re-transmission of the message). To include the costs of error handling in the analysis of the previous section, we define the function *E*(*t*): the most probable bound on the overheads due to errors in an interval of duration *t*. We include in this function the costs of retransmission. This function can be defined using statistical analysis based on observed error characteristics of a given configuration of CAN in a given environment. Each detected error implies the re-transmission of a message. We assume that as soon as the sending station detects an error in the transmission of a message it immediately re-queues the message for transmission. The assumption is an important one for the following reason: if the message is not immediately re-queued then the bus may become idle and a lower priority message attain access to the bus (and then begin transmission). This means that the message being re-transmitted may be again delayed by a lower priority message. In general, therefore, a given message *m* would be delayed by lower priority messages for up to time (*n* + 1)*B*, where *n* is the number of re-transmissions of message *m*. This would needlessly add to the worst-case response time of the message.

A probable bound on the error recovery overheads before a message *m* arrives at the destination is:

$$E(R_m)$$

Now that we have defined the overheads due to error handling for the transmission of a given message *m*, we can include these overheads in the analysis developed in the previous section. We update equation 2 to:

$$t_m = E(t_m + C_m) + B + \sum_{\forall j \in hp(m)} \left\lceil \frac{t_m + J_j + \tau_{bit}}{T_j} \right\rceil C_j \qquad (3)$$

Note that we have re-written R_m as $t_m + C_m$ (from equation 1).

We now describe CAN remote transmission request (RTR) messages. This message is a special CAN message with a zero length data field. It is interpreted by stations to mean "please transmit the message with the same identifier as this message". Because identifiers are unique within the system, there can only be one station that responds to this message (if no stations respond by transmitting the requested message then an error is flagged). We make the assumption that a station responding to an RTR message will immediately queue the requested message for transmission such that no lower priority message can be transmitted first (for the same reasons described earlier for the assumption that re-transmissions occur immediately). Of course, if a higher priority message has been queued since the transmission of the RTR message, then the higher priority message will be transmitted after the RTR message has been sent and before the requested message is sent.

A number of stations may transmit RTR messages, and one station may transmit the corresponding requested message; all of these messages have the same identifier, and hence priority. This complicates the analysis slightly: previously, messages were assumed to have unique identifiers, and the set *hp*(*m*) for a given message *m* indicated all the messages that could win the arbitration process and delay the transmission of *m*. However, with the introduction of RTR messages this is no longer true. This problem is addressed by CAN in two ways. Firstly, although a number of stations can simultaneously attempt to transmit RTR messages with the same identifier, no collision results. This is because RTR messages with the same identifier have identical bit patterns (recall that RTR messages are zero byte messages): no station will see other than the data it transmitted. Secondly, the CAN message arbitration gives priority to requested messages over RTR messages with the same identifier.

One way to address the problem of interference between RTR and requested data messages all with the same identifier is to change the set *hp*(*m*) to include all messages of higher *or the same* priority. However, this is pessimistic, since it is possible for CAN to prevent the interference if we are careful with how we define the semantics of an RTR message. We say that the worst-case response time of an RTR message is defined as the longest time between queueing the RTR message and the *requested* message arriving at destination stations. This definition means that if an RTR message is queued at some time, but that before the RTR message is transmitted the requested data message is received (in response to an earlier RTR message, say), then the response time is the time between the still-untransmitted RTR message and the reception of the requested data message. Thus the RTR message could be satisfied before it has been transmitted! (a sensible implementation of CAN would then remove the untransmitted RTR message).

Because of this definition of RTR response time we do not have to consider the interference between data messages and RTR messages of the same priority. We now continue, and define new notation: the time $C_{RTR(m)}$ is the value of C for the requested message, where m is a given message. If m is not an RTR message then we define $C_{RTR(m)}$ to be zero.

Because the worst-case response time of an RTR message includes the time taken to transmit the requested message, we must re-define the equation for the worst-case response time of a given message *m* (from equation 1):

$$R_m = \begin{cases} t_m + C_{RTR(m)} & \text{if } m \in RTR \\ t_m + C_m & \text{otherwise} \end{cases} \tag{4}$$

Where *RTR* is the set of RTR messages. The term t_m represents the queueing delay for message *m* (as before), but this queueing delay must also include the time taken to transmit the RTR message. The interference from higher priority RTR messages must include the transmission of the corresponding data message. Equation 3 is therefore updated to:

$$t_m = C_m + E\left(t_m + C_{RTR(m)}\right) + B + \sum_{\forall j \in hp(m)} \left\lceil \frac{t_m + J_j + \tau_{bit}}{T_j} \right\rceil \left(C_j + C_{RTR(j)}\right) \tag{5}$$

Thus we have completed the analysis of CAN.

4. CONCLUSIONS

Hitherto a perceived problem with CAN for use in distributed real-time control applications was that it was impossible to determine the worst-case response time of a given message. In this short paper we have shown how to find the worst-case response time of a given message queued for transmission across a CAN bus. Basic analysis has been developed for a simple CAN model, and then extended to include the costs of error handling and remote transmission request messages.

In order to apply the analysis to a real implementation we must examine in more detail the behaviour of the given CAN controller to see if it meets the assumptions outlined in this paper. The reader is referred to [4] for a detailed case study, using the Intel 82527 CAN controller on a large ‘benchmark’ automotive control problem.

5. REFERENCES

[1] 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. 284-292 (September 1993)

[2] Leung, J., and Whitehead, J., “On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks” *Performance Evaluation* **2**(4), pp. 237-250 (December 1982)

[3] "Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High Speed Communication", *ISO DIS 11898* (February 1992)

[4] Tindell, K. and A. Burns, "Guaranteed Message Latencies for Distribute Safety-Critical Hard Real-Time Networks", *YCS 229*, Department of Computer Science, University of York (June 1994)

[5] Tindell, K., "Analysis of Hard Real-Time Communications", *YCS 222*, Department of Computer Science, University of York (January 1994)

[6] Tindell, K. and Clark, J., "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems," *Micrprocessors and Microprogramming* (Special Issue on Parallel Embedded Real-Time Systems) (March 1994)

[7] Tindell, K., Burns, A., and Wellings, A., "*An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks,*" *Real-Time Systems* **6**(2) pp. 133-151