Pergamon

Control Eng. Practice, Vol. 3, No. 8, pp. 1163-1169, 1995

0967-0661/95 $9.50 + 0.00

0967-0661(95)00112-3

CALCULATING CONTROLLER AREA NETWORK (CAN) MESSAGE RESPONSE TIMES

K. Tindell*, A. Burns and A.J. Wellings****

**Institutionen för Datorteknik, Uppsala Universitet, Sweden*
***Department of Computer Science, University of York, UK*

(Received October 1994; in final form May 1995)

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 an analysis to bound such latencies. A benchmark is used to illustrate the application of this analysis.

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

1. INTRODUCTION

Controller Area Network (CAN) (ISO, 1993) 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, analysis developed for fixed-priority preemptive real-time processor scheduling (Audsley *et al.*, 1993; Tindell *et al.*, 1994*a*) is applied to the problem of message scheduling on a CAN bus. Before proceeding further, the architecture of CAN is described briefly, and some general observations and assumptions about the implementation are made.

CAN is a broadcast bus where a number of processors are connected to the bus via an interface. 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. 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 at the same time 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 the 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 bit 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. 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 easily be calculated: the longest time a station must wait for the bus to become idle is the longest time to transmit a CAN message (this delay is termed 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. An analysis is given 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 proceeding to develop such analysis there is a need to discuss briefly how CAN messages are queued in the stations. Fig. 1 depicts a typical interface.

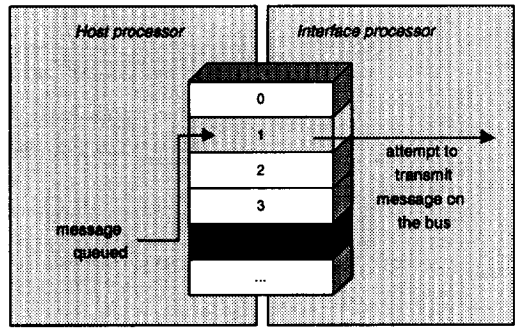

Fig. 1. CAN controller architecture

In Fig. 1 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: 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', it can be seen 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, there is a *deadline* on the transmission of any message: the message must be transmitted before the subsequent message can be queued (of course, there may be a deadline on the message that is much shorter than the period).

2. ANALYSIS OF A SIMPLE CAN MODEL

In this section a simple analysis is developed for the CAN model outlined above. In reality, CAN is more complex than described; Tindell and Burns (Tindell and Burns, 1994) have extended 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 an analysis has been developed for systems where activities are dispatched according to fixed priorities (Audsley *et al.*, 1993; Tindell and Clark, 1994; Tindell *et al.*, 1994*a*; Tindell *et al.*, 1995).

Because CAN is primarily a priority-based bus, much of this analysis can be applied directly. In this paper the application of the analysis will be shown; the reader is referred elsewhere for a more formal derivation of the general theory (Audsley *et al.*, 1993).

Before introducing the analysis some terms are defined. 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 cyclically (*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 instances of the message). This is illustrated in Fig. 2.

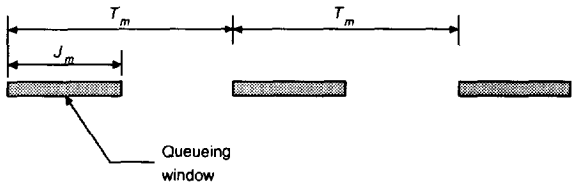

Fig. 2. Queueing windows

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 transmit the message physically on the bus. This does not include 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. 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:

$$R_m \leq D_m.$$

There is a restriction on the worst-case response time: a queued message must be sent before the next queueing of the message (it is important to prevent the overwriting of a message). Thus the following must hold:

$$R_m \leq T_m - J_m.$$

From this it can be seen that the message queueing window (*i.e.* the message queueing jitter) must be less than the periodicity of the message. An analysis is now developed to determine the worst-case response time of a given message m.

The worst-case response time is defined to be composed of two delays: the *queueing delay* and the *transmission delay*. The queueing delay is the longest time that a message can be queued within a station and be delayed because other higher- and lower- priority messages are being sent on the bus. This time is denoted as t_m. The transmission delay is the actual time taken to send the message on the bus. As was 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:

$$R_m = t_m + C_m. \tag{1}$$

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 these times were termed the blocking time, and denoted as B. The latter time is termed the *interference*. From earlier scheduling theory (Audsley *et al.*, 1993), 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 it is known that the optimal priority ordering is *deadline monotonic* (Leung and Whitehead, 1980). That is, the smaller the value of D, the higher the message priority. From the above description it can be seen that the queueing delay is given by:

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

The smallest value of t_m satisfying the above equation is desired. 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, the iteration needs to start 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. IMPLEMENTED CAN CONTROLLERS

The previous section derived an analysis assuming an 'ideal' CAN controller. In the real world CAN controllers are not so ideal. This section describes briefly two implementations of a CAN controller and shows how only one of them meets the assumptions of ideal behaviour.

The Intel 82527 is a 'FullCAN' controller: the buffering of messages is done on-chip via dual-ported memory (as described earlier in this paper). However, instead of allowing the full 2032 possible messages, the controller allows only fourteen (actually, it permits fifteen, but the fifteenth 'slot' is dedicated to receiving messages). Although there are only fourteen slots, the identifiers of the messages stored in the slots are not restricted. Internal arbitration is not by message identifier, but by slot number. It is therefore important that messages are allocated to slots in priority order. Apart from the limit on the number of messages, the Intel 82527 behaves as the analysis assumes.

The Philips 82C200 is a 'BasicCAN' controller: there is very little buffering done within the controller. Messages are stored in the main memory of the CPU, and transferred to and from the controller in response to "message sent" and "message received" interrupts. Unfortunately, the analysis in this paper is not applicable to this controller. The reason is that there are additional delays caused by handling the 'handshaking' interrupts. Also, during the handling of a "message sent" interrupt the controller does not enter into arbitration on the bus (having no valid message to send until the CPU has copied the next one across). This means that between each pair of outgoing messages, arbitration could be won by a low-priority message from another station. The cumulative delay of these low-priority messages is not accounted for by the analysis of this paper. Tindell *et al.* discuss this in more detail (Tindell *et al.*, 1994*b*), and give a pathological example where the worst-case latency of a medium-priority message is over five times higher with the Philips 82C200-style of controller than with the Intel 82527-style of controller. The rest of this paper assumes the use of a CAN controller which behaves 'ideally'.

4. AN SAE BENCHMARK

The Society of Automotive Engineers (SAE) provided a 'benchmark' application to be used to evaluate different multiplexing communications technologies (SAE, 1993). The benchmark report describes a set of signals sent between seven different subsystems in a prototype electric car. Although the car control system was engineered using point-to-point links, the set of signals provide a good example to illustrate the application of CAN bus to complex distributed real-time control systems.

The seven subsystems are: the batteries ('Battery'), the vehicle controller ('V/C'), the inverter/motor controller ('I/M C'), the instrument panel display ('Ins'), driver inputs ('Driver'), brakes ('Brakes'), and the transmission control ('Trans'). The network connecting these subsystems is required to handle a total of 53 signals, some of which are sporadic and some of which contain control data sent periodically. A periodic signal has a fixed period, and implicitly requires the latency to be no more than this period. Sporadic messages have latency requirements imposed by the application: for example, all messages sent as a result of a driver action have a latency requirement of 20ms so that the response appears to the driver to be instantaneous. Table 1 details the signals. The destinations of the signals have been omitted because CAN does not use destination addresses.

If a simple approach is taken (where each signal is assigned its own CAN message) then the system is unschedulable at a bus speed of 125 kbit/s. This is not surprising, since the bus utilisation at that speed is 125%. A further problem exists: the V/C subsystem transmits 17 different signals, more than many CAN implementations can store on-chip (recall how the Intel 82527 holds at most 14 messages for transmission).

One way to reduce the bus utilisation, and the number of messages sent from a station, is to 'piggyback' messages from the same source. For example, the Battery subsystem sends four single byte signals every 100ms (signals 1, 2, 4, and 6). These can be collected into a single message with size 4 bytes. This reduces the bus utilisation because the overhead for three messages is saved (with CAN this overhead is relatively large: a single-byte message can require the transmission of 63 bits, where only eight of these contain data).

The piggybacking can extend to signals that are sent at different rates: signals 29, 30, and 32 (for example) can be sent in a message with period 5ms, even though two of those signals (29 and 30) have a period of 10ms (every second message sent

Table 1 Signals

Signal N°	Description	size/bits	T/ms	Type	D/ms	From
1	Traction Battery Voltage	8	100.0	P	-	Battery
2	Traction Battery Current	8	100.0	P	-	Battery
3	Traction Battery Temp, Average	8	1000.0	P	-	Battery
4	Auxiliary Battery Voltage	8	100.0	P	-	Battery
5	Traction Battery Temp, Max	8	1000.0	P	-	Battery
6	Auxiliary Battery Current	8	100.0	P	-	Battery
7	Accelerator Position	8	5.0	P	-	Driver
8	Brake Pressure, Master Cylinder	8	5.0	P	-	Brakes
9	Brake Pressure, Line	8	5.0	P	-	Brakes
10	Transaxle Lubrication Pressure	8	100.0	P	-	Trans
11	Transaction Clutch Line Pressure	8	5.0	P	-	Trans
12	Vehicle Speed	8	100.0	P	-	Brakes
13	Traction Batter Ground Fault	1	1000.0	P	-	Battery
14	Hi & Lo Contactor Open/Close	4	50.0	S	5.0	Battery
15	Key Switch Run	1	50.0	S	20.0	Driver
16	Key Switch Start	1	50.0	S	20.0	Driver
17	Accelerator Switch	2	50.0	S	20.0	Driver
18	Brake Switch	1	20.0	S	20.0	Brakes
19	Emergency Brake	1	50.0	S	20.0	Driver
20	Shift Lever	3	50.0	S	20.0	Driver
21	Motor/Trans Over Temperature	2	1000.0	P	-	Trans
22	Speed Control	3	50.0	S	20.0	Driver
23	12V Power Ack Vehicle Control	1	50.0	S	20.0	Battery
24	12V Power Ack Inverter	1	50.0	S	20.0	Battery
25	12V Power Ack I/M Contr.	1	50.0	S	20.0	Battery
26	Brake Mode	1	50.0	S	20.0	Driver
27	SOC Reset	1	50.0	S	20.0	Driver
28	Interlock	1	50.0	S	20.0	Battery
29	High Contactor Control	8	10.0	P	-	V/C
30	Low Contactor Control	8	10.0	P	-	V/C
31	Reverse & 2nd Gear Clutches	2	50.0	S	20.0	V/C
32	Clutch Pressure Control	8	5.0	P	-	V/C
33	DC/DC Converter	1	1000.0	P	-	V/C
34	DC/DC Converter Current Control	8	50.0	S	20.0	V/C
35	12V Power Relay	1	50.0	S	20.0	V/C
36	Traction Battery Ground Fault Test	2	1000.0	P	-	V/C
37	Brake Solenoid	1	50.0	S	20.0	V/C
38	Backup Alarm	1	50.0	S	20.0	V/C
39	Warning Lights	7	50.0	S	20.0	V/C
40	Key Switch	1	50.0	S	20.0	V/C
41	Main Contactor Close	1	50.0	S	20.0	I/M C
42	Torque Command	8	5.0	P	-	V/C
43	Torque Measured	8	5.0	P	-	I/M C
44	Fwd/Rev	1	50.0	S	20.0	V/C
45	Fwd/Rev Ack	1	50.0	S	20.0	I/M C
46	Idle	1	50.0	S	20.0	V/C
47	Inhibit	1	50.0	S	20.0	I/M C
48	Shift in Progress	1	50.0	S	20.0	V/C
49	Processed Motor Speed	8	5.0	P	-	I/M C
50	Inverter Temperature Status	2	50.0	S	20.0	I/M C
51	Shutdown	8	50.0	S	20.0	I/M C
52	Status/Malfunction	1	50.0	S	20.0	I/M C
53	Main Contactor Acknowledge	8	50.0	S	20.0	V/C

would contain null data for these signals).

It is also possible to piggyback signals that are not generated at the same time (for example, sporadic signals). Such signals can be sent in a periodic 'server' message, where the sending station polls for sporadic signals before queueing the message. With this approach a sporadic signal may be delayed for up to a polling period, plus the worst-case latency of the 'server' message. So, to piggyback a number of sporadic signals with latency requirements of 20ms (say), a server message with a period of 15ms and a worst-case latency of 5ms would be sufficient.

By using piggybacking the bus load for the signals in the case study can be reduced to 85%, and all the timing constraints met. Table 2 details the piggybacking arrangement, and the worst-case timing behaviour of the system. Note that there are two sporadic signals that remain mapped to sporadic messages: signal 14 has a deadline that is too short to meet by polling; signal 18 is the only sporadic signal sent from the Brakes subsystem and cannot sensibly be piggybacked with other signals. Notice how all the timing constraints are met.

5. EXTENDING THE ANALYSIS

The analysis given so far assumes that transmission is error free, and hence no error recovery procedures delay the delivery of messages. This assumption does not hold in reality, so it is important to update the analysis to allow for error recovery time. The first step towards this is to understand how CAN detects and recovers from errors.

There may be several sources of errors, detectable by different stations. For example, the transmitting station monitors the bus while transmitting, and if the monitored value differs from the transmitted value then the station begins recovery from the error (note that this monitoring is disabled during arbitration). A receiving station may detect a mismatch between the message contents and a CRC field appended to the message. Another important error is the violation of the bit stuffing protocol. It was mentioned earlier how the CAN protocol uses bit stuffing for transmitting the message stream: if six bits of the same sign are transmitted then an opposite bit is inserted after the fifth bit. Therefore, if a station receives six bits of the same sign it knows that an error has occurred.

If a station detects an error it begins the recovery process by immediately transmitting an *error flag*. The format of the error flag is designed to cause all other stations to detect a stuff error, and to begin recovery themselves (by transmitting error flags which superimpose). The purpose of this process is to abort the transmission of the erroneous message, and to ensure all stations are ready to enter into arbitration again (the aborted message can be re-entered into arbitration). The resynchronisation process takes at most 29 bit times (ISO, 1993).

To allow for the recovery overhead the *error function*, $E(t)$ is defined – this is the longest time that is spent on recovering from errors in any interval of duration t. A crude error function can be obtained by assuming that there can be at most n_{burst} errors in rapid succession, with a residual error period no shorter than T_{error}. A further assumption is that each of these errors occurs just at the end of the transmission of the erroneous message. This means that the overheads in any interval of duration t are bounded by:

$$\left(n_{burst} + \left\lceil \frac{t}{T_{error}} \right\rceil\right) C_{overhead}$$

where $C_{overhead}$ is the longest time taken to handle an error and re-send the largest message transmitted in the interval.

This formulation of the error function can be incorporated into the analysis presented so far. A given message m is susceptible to delay due to errors at any time between queueing and finally being received correctly. This interval was defined earlier as the worst-case latency of a message, and denoted R_m (also equal to $t_m + C_m$). The largest message that can be transmitted during this interval is given by:

$$\max_{\forall j \in (hp(i) \cup \{i\})} (C_j).$$

The overheads for handling an error do not amount to more than:

$$C_{overhead,m} = \max_{\forall j \in (hp(i) \cup \{i\})} (C_j) + 29\tau_{bit}.$$

Therefore an error function for a given message m can be defined as:

$$E_m = \left(n_{burst} + \left\lceil \frac{t_m + C_m}{T_{error}} \right\rceil\right) C_{overhead,m}. \quad (3)$$

Equation 2, giving the worst-case latency of a given message m, can be updated thus:

Table 2 Results of Timing Analysis

Signal N°s	size/bytes	T/ms	Type	D/ms	R/ms	From
14	1	1000.0	S	5.0	1.544	Battery
8,9	2	5.0	P	5.0	2.128	Brakes
7	1	5.0	P	5.0	2.632	Driver
43,49	2	5.0	P	5.0	3.216	I/M C
11	1	5.0	P	5.0	3.720	Trans
32,42	2	5.0	P	5.0	4.304	V/C
31,34,35,37,38,39,40,44,46,48,53	6	10.0	P	10.0	5.192	V/C
23,24,25,28	1	10.0	P	10.0	8.456	Battery
15,16,17,19,20,22,26,27	2	10.0	P	10.0	9.040	Driver
41,45,47,50,51,52	2	10.0	P	10.0	9.624	I/M C
18	1	100.0	S	20.0	10.128	Brakes
1,2,4,6	4	100.0	P	100.0	18.944	Battery
12	1	100.0	P	100.0	19.448	Brakes
10	1	100.0	P	100.0	19.552	Trans
3,5,13	3	1000.0	P	1000.0	20.608	Battery
21	1	1000.0	P	1000.0	29.192	Trans
33,36	1	1000.0	P	1000.0	29.696	V/C

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

6. 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 it has been shown how to find the worst-case response time of a given message queued for transmission across a CAN bus. A 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. This analysis has been applied to a realistic case study.

7. REFERENCES

Audsley, N., A. Burns, M. Richardson, K. Tindell and A. Wellings (1993). Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal* 8(5), 284–292.

ISO (1993). *Road Vehicles: Interchange of Digital Information: Controller Area Network (CAN) for High Speed Communication.* ISO 11898:1993.

Leung, J. and J. Whitehead (1980). On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation* **2**(4), 237–250.

SAE (1993). *Class C Application Requirement Considerations.* SAE J2056/1.

Tindell, K., A. Burns and A. Wellings (1994*a*). An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems* **6**(2), 133–151.

Tindell, K., A. Burns and A. Wellings (1995). Analysis of hard real-time communications. *Real-Time Systems (To Appear).*

Tindell, K. and A. Burns (1994). Guaranteed message latencies for distributed safety-critical hard real-time networks. Technical Report YCS 229. Department of Computer Science, University of York.

Tindell, K. and J. Clark (1994). Holistic schedulability analysis for distributed hard real-time systems. *Microprocessors and Microprogramming* **40**, 117–134.

Tindell, K., H. Hansson and A. Wellings (1994*b*). Analysing real-time communications: Controller area network. In: *Proceedings IEEE Real-Time Systems Symposium.* San Juan, Puerto Rico.