GUARANTEEING MESSAGE LATENCIES ON CONTROL AREA NETWORK (CAN)[1]

Ken Tindell, Alan Burns
Real-Time Systems Research Group,
Department of Computer Science,
University of York, YO1 5DD, England

ABSTRACT

A generally perceived problem with CAN is that it is unable to guarantee the timing performance of lower priority messages. Recent work has developed analysis to bound message latencies under arbitrary error rate assumptions. Messages are permitted to be periodic or sporadic, with few restrictions on deadlines. The analysis can give optimal identifier orderings, and be used to ask "what if" questions. In this paper we derive the analysis and apply it to an SAE 'benchmark', assuming the Intel 82527 stand-alone CAN controller is used.

1. INTRODUCTION

A common misconception within the automotive industry is that while CAN is very good at transmitting the most urgent data, it is unable to provide guarantees that deadlines are met for less urgent data [6, 2]. This is not the case: the dynamic scheduling algorithm used by CAN is virtually identical to scheduling algorithms commonly used in real-time systems to schedule computation on processors. In fact, the analysis of the timing behaviour of such systems can be applied almost without change to the problem of determining the worst-case latency of a given message queued for transmission on CAN.

This paper reproduces the existing processor scheduling analysis, and shows how this analysis is applied to CAN. In order for the analysis to remain accurate, details of the implementation of CAN controllers must be known, and therefore in this paper we assume an existing controller (the Intel 82527) to illustrate the application of the analysis. We then apply the analysis to the SAE 'benchmark' for class C automotive systems (safety critical control applications) [1]. We extend the CAN analysis to deal with some fault tolerance issues.

The paper is structured as follows: the next section outlines the behaviour of CAN (as implemented by the Intel 82527) and the assumed system model. Section 3 applies the basic processor scheduling analysis to the 82527. Section 4 then applies this analysis to the standard benchmark, using a number of approaches. Finally, section 5 discusses some outstanding issues, and offers conclusions.

2. SYSTEM MODEL

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

---

[1]The authors can be contacted via e-mail as `ken@minster.york.ac.uk`; copies of York technical reports cited in this paper are available via FTP from `minster.york.ac.uk` in the directory `/pub/realtime/papers`

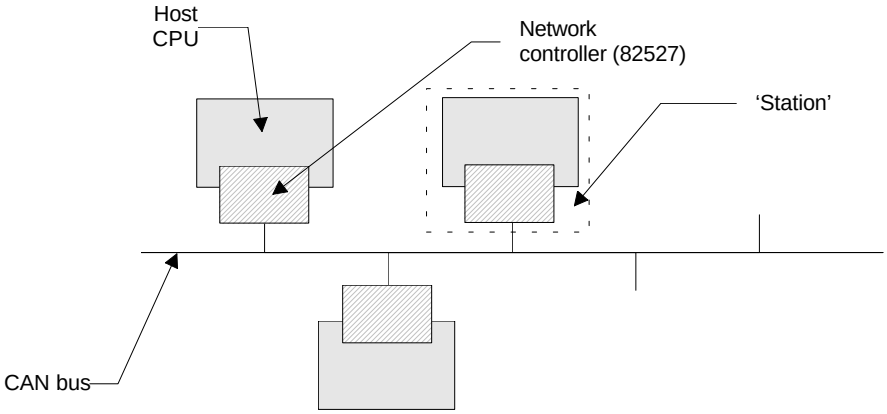

Figure 1: CAN architecture

A data source is transmitted as a *message*, consisting of between 1 and 8 bytes ('octets'). A message 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. The identifier servers two purposes: filtering messages upon reception, and assigning a priority to the message. As we are concerned with closed control systems we assume a fixed set of messages each having a unique priority and temporal characteristics such as rate and (maximum) size.

As well as data messages, CAN also permits 'remote transmission request' (RTR) messages. These messages are contentless and have a special meaning: they instruct the station holding a data message of the same identifier to transmit that message. RTR messages are intended for quickly obtaining infrequently used remote data. However, the 'benchmark' [1] does not require RTR messages, and so we do not discuss these types of messages further.

We can only apply the analysis to a particular controller, since different controllers can have different behaviours (for example, the Philips 82C200 controller can have a much worse timing performance that the 'ideal' behaviour [12]). A controller is typically connected to the host processor via dual-ported RAM (DPRAM), whereby the CPU and the controller can access the memory simultaneously. A perfect CAN controller would contain 2032 'slots' for messages, where a given message to be sent is simply copied into the slot corresponding to the message identifier. A received message would also be copied into a corresponding slot. However, the amount of memory required for this would be too expensive for many systems. The Intel 82527 makes the compromise of giving just 15 slots. One of these slots is dedicated to receiving messages; the remaining 14 slots can be set to either transmit or receive messsages. Each slot can be mapped to any given identifier; slots programmed to receive can be set to receive any message matching an identifier mask. Each slot can be independently programmed to generate an interrupt when receiving a message into the slot, or sending a message from the slot. This enables 'handshaking' protocols with the CPU, permitting a given slot to be multiplexed between a number of messages. This is important when controlling the dedicated receive slot 15: this special slot is 'double buffered' so that the CPU has time to empty one buffer whilst the shadow buffer is available to the controller. In this paper we assume that slots are statically allocated to messages, with slot 15 used to receive messages that cannot be fitted into the remaining slots. The 82527 has the quirk that messages stored in the slots are entered into arbitration in slot order rather than identifier (and hence priority) order. Therefore it is important to allocate the messages to the slots in priority order.

We now outline a 'system model' for message passing that we are able to analyse. A system is deemed to be composed of a static set of hard real-time messages, each statically assigned to a set of stations connected to the bus. These hard real-time messages are typically control messages, and have deadlines that must be met, or else a serious error is said to occur. Messages will typically be queued by a software *task* running on the host CPU (the term 'task' encompasses a number of activities, ranging from interrupt handlers, to heavyweight processes provided by an operating system). A given task is assumed to be invoked by some event, and to then take a bounded time to queue the message. Because this time is bounded instead of fixed, there is some variability, or *jitter*, between subsequent queuings of the message; we term this *queuing jitter*. For the purposes of this paper, we

assume that there is a minimum time between invocations of a given task; this time is termed the *period*[2]. If the given task sends a message once every few invocations of the task, then the message *inherits* a period from the sending task.

A given message is assigned a fixed identifier (and hence a fixed priority). We assume that each given hard real-time message must be of bounded size (*i.e.* contain a bounded number of bytes). Given a bounded size, and a bounded rate at which the message is sent, we effectively bound the peak load on the bus, and can then apply scheduling analysis to obtain a latency bound for each message.

We assume that there may also be an unbounded number of soft real-time messages: these messages have no hard deadline, and may be lost in transmission (for example, the destination processor may be too busy to receive them). They are sent as 'added value' to the system (*i.e.* if they arrive in reasonable time then some quality aspect of the system is improved). In this paper we do not discuss special algorithms to send these, and for simplicity instead assume that they are sent as "background" traffic (*i.e.* assigned a priority lower than all hard real-time messages)[3].

As mentioned earlier, the queuing of a hard real-time message can occur with *jitter* (variability in queuing times). The following diagram illustrates this:

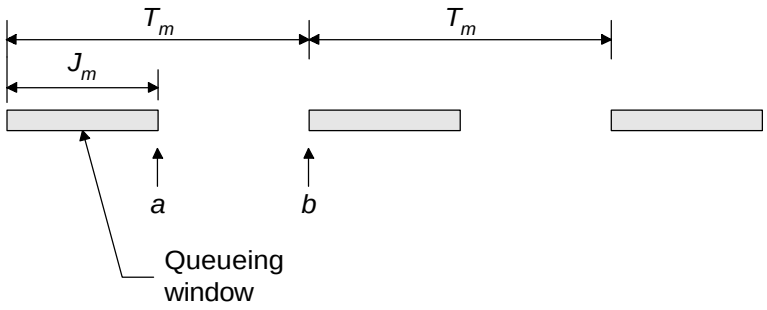

Figure 2: message queuing jitter

The shaded boxes in the above diagram represent the 'windows' in which a task on the host CPU can queue the message. Jitter is important because to ignore it would lead to insufficient analysis. For example, ignoring jitter in Figure 2 would lead to the assumption that message *m* could be queued at most once in an interval of duration (*b* – *a*). In fact, in the interval (*a* .. *b*] the message could be queued twice: once at *a* (as late as possible in the first queuing window), and once at *b* (as early as possible in the next queuing window). Queuing jitter can be defined as the difference between the earliest and latest possible times a given message can be queued. In reality, it may be possible to reduce the queuing jitter if we know where in the execution of a task a message is queued (for example, there may be a minimum amount of computation required before the task could queue a message, and therefore event *b* in the diagrams would occurs later than the start of the task); other work has addressed this [11].

The diagram above also shows how the *period* of a message can be derived from the task sending the message. For example, if the message is sent once per invocation of the task, then the message inherits a period equal to the period of the task.

To keep the queuing jitter of a message small, we might decompose the task generating the message into two tasks: the first task calculates the message contents, and the second 'output' task merely queues the message. The second task is invoked a fixed time after the first task, such that first task will always have completed before the second task runs. Since the second task has very little work to do, it can typically have a short worst-case

[2]Of course, the task could be invoked once only (perhaps in response to an emergency), and would therefore have an infinite period.

[3]There are a number of algorithms that could potentially lead to very short average response times for soft real-time messages; the application of these algorithms to CAN bus is the subject of on-going research.

response time, and the queuing jitter inherited by the message will therefore be small (this general technique is discussed in more detail elsewhere [10]).

We briefly discuss how a message is handled once received. At a destination station the results of an incoming message must be made available. If the message is a sporadic one (*i.e.* sent as the result of a 'chance' event) then there is a task which should be invoked by the message arrival. In this case, the message arrival should raise an interrupt on the host CPU (and hence be assigned to slot 15 on the Intel 82527 bus controller). Of course, it is possible for the arrival of the message to be polled for by the task, but if the required end-to-end latency is small then the polling period may have to be unacceptably high. The arrival of a periodic message can be dealt with without raising an interrupt: the message can be statically assigned to a slot in the 82527 and then be picked up by the application task. This task could be invoked by a clock (synchronised to a notional global clock) so that the message is guaranteed to have arrived when the task runs.

As can be seen, the sole requirement on the communications bus is that messages have bounded latencies. We now proceed to develop analysis to give these. Clearly, this analysis will form a key part of a wider analysis of the complete system to give end-to-end timing guarantees; such end-to-end analysis is the subject of on-going research at York.

Having established the basic model for CAN and the system we are now able to give analysis bounding the timing behaviour of a given hard real-time message.

3. ANALYSIS OF 82527 CAN

In this section we present analysis that bounds the worst-case latency of a given hard real-time message type. The analysis is an almost direct application of processor scheduling theory [9, 3, 4]. However, there are some assumptions made by this analysis: Firstly, the deadline of a given message m (denoted D_m) must not be more than the period of the message (denoted T_m). Secondly, the bus controller must not release the bus to lower priority messages if there are higher priority messages pending (*i.e.* the controller cannot release the bus between sending one message and entering any pending message into the arbitration phase; note that some CAN controllers fail to meet this assumption).

The *worst-case response time* of a given message m is denoted by R_m, and defined as the longest time between the start of a task queuing m and the latest time that the message arrives at the destination stations. Note that this time includes the time taken for the sender task to execute and queue message m, and is at first sight a curious definition (measuring the time from the queuing of the message to the latest arrival might seem better). However, the contents of the message reflects the results of some action undertaken by the task (itself triggered in response to some event), and it is more desirable to measure the wider end-to-end time associated with an event.

The jitter of a given message m is denoted J_m, and is derived from the response time of the tasks on the host CPU. If these tasks are scheduled by fixed priority pre-emptive scheduling then related work can bound the time taken to queue the message [5, 9] and hence determine the queuing jitter.

We mentioned earlier how CAN operates a fixed priority scheduling algorithm. However, a message is not fully pre-emptive, since a high priority message cannot interrupt a message that is already transmitting. The work of Burns *et al* [4] allows for this behaviour, and from other processor scheduling work [3] we can bound the worst-case response time of a given hard real-time message m by the following:

$$R_m = J_m + w_m + C_m \qquad (1)$$

The term J_m is the queuing jitter of message m, and gives the latest queuing time of the message, relative to the start of the sending task. The term w_m represents the worst-case queuing delay of message m (due to both higher priority messages pre-empting message m, and a lower priority message that has already obtained the bus).

The term C_m represents the longest time taken to physically send message m on the bus. This time includes the time taken by the frame overheads, the data contents, and extra stuff bits (the CAN protocol specifies that the message contents and 34 bits of the overheads are subject to bit stuffing with a stuff width of 5). The following equation gives C_m:

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

The term s_m denotes the bounded size of message m in bytes. The term τ_{bit} is the bit time of the bus (on a bus running at 1 Mbit/sec this is 1μs).

The queuing delay is given by:

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

The set $hp(m)$ is the set of messages in the system of higher priority than m. T_j is the period of a given message j, and J_j is the queuing jitter of the message. B_m is the longest time that the given message m can be delayed by lower priority messages (this is equal to the time taken to transmit the largest lower priority message), and can be defined by:

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

Where $lp(m)$ is the set of lower priority messages. Note that if there are an unbounded number of soft real-time messages of indeterminate size, then B_m is equal to $130\tau_{bit}$.

Notice that in equation 2 term w_m appears on both the left and right hand sides, and the equation cannot be re-written in terms of w_m. A simple solution is possible by forming a recurrence relation:

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

A value of zero for w_m^0 can be used. The iteration proceeds until convergence (*i.e.* $w_m^{n+1} = w_m^n$).

The above equations do not assume anything about how identifiers (and hence priorities) are chosen. However, from work on processor scheduling [7, 3] we know that the optimal ordering of priorities is the *deadline monotonic* one: a task with a short value of $D - J$ should be assigned a high priority.

We can now apply this analysis to the SAE benchmark [1].

4. THE SAE 'BENCHMARK'

The SAE 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 messages, some of which contain sporadic signals, and some of which contain control data sent periodically. A periodic message has a fixed period, and implictly requires the latency to be less than or equal to 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.

The reader is referred to the work of Kopetz [6] for a more detailed description of the benchmark. Note that Kopetz is forced to 'interpret' the benchmark specification, giving sensible timing figures where the benchmark fails to

specify them (for example, the latency requirement of 20ms for driver-initiated messages is a requirement imposed by Kopetz rather than the benchmark). There is still some unspecified behaviour in the benchmark: the system model assumed by this paper requires that even sporadic messages are given a period (representing the maximum rate at which they can occur), but no periods for the sporadic messages in the benchmark can be inferred (Kopetz implicitly assumes that sporadic messages have a period of 20ms). Like Kopetz, we are forced to assume sensible values. We also hypothesise queuing jitter values.

The following table details the requirements of the messages to be scheduled. There are a total of 53 messages, some simple periodic messages, and some 'chance' messages (*i.e.* queued sporadically in response to an external event).

Signal Number	*Signal Description*	*Size /bits*	*J /ms*	*T /ms*	*Periodic /Sporadic*	*D /ms*	*From*	*To*
1	Traction Battery Voltage	8	0.6	100.0	P	100.0	Battery	V/C
2	Traction Battery Current	8	0.7	100.0	P	100.0	Battery	V/C
3	Traction Battery Temp, Average	8	1.0	1000.0	P	1000.0	Battery	V/C
4	Auxiliary Battery Voltage	8	0.8	100.0	P	100.0	Battery	V/C
5	Traction Battery Temp, Max.	8	1.1	1000.0	P	1000.0	Battery	V/C
6	Auxiliary Battery Current	8	0.9	100.0	P	100.0	Battery	V/C
7	Accelerator Position	8	0.1	5.0	P	5.0	Driver	V/C
8	Brake Pressure, Master Cylinder	8	0.1	5.0	P	5.0	Brakes	V/C
9	Brake Pressure, Line	8	0.2	5.0	P	5.0	Brakes	V/C
10	Transaxle Lubrication Pressure	8	0.2	100.0	P	100.0	Trans	V/C
11	Transaction Clutch Line Pressure	8	0.1	5.0	P	5.0	Trans	V/C
12	Vehicle Speed	8	0.4	100.0	P	100.0	Brakes	V/C
13	Traction Battery Ground Fault	1	1.2	1000.0	P	1000.0	Battery	V/C
14	Hi&Lo Contactor Open/Close	4	0.1	50.0	S	5.0	Battery	V/C
15	Key Switch Run	1	0.2	50.0	S	20.0	Driver	V/C
16	Key Switch Start	1	0.3	50.0	S	20.0	Driver	V/C
17	Accelerator Switch	2	0.4	50.0	S	20.0	Driver	V/C
18	Brake Switch	1	0.3	20.0	S	20.0	Brakes	V/C
19	Emergency Brake	1	0.5	50.0	S	20.0	Driver	V/C
20	Shift Lever (PRNDL)	3	0.6	50.0	S	20.0	Driver	V/C
21	Motor/Trans Over Temperature	2	0.3	1000.0	P	1000.0	Trans	V/C
22	Speed Control	3	0.7	50.0	S	20.0	Driver	V/C
23	12V Power Ack Vehicle Control	1	0.2	50.0	S	20.0	Battery	V/C
24	12V Power Ack Inverter	1	0.3	50.0	S	20.0	Battery	V/C
25	12V Power Ack I/M Contr.	1	0.4	50.0	S	20.0	Battery	V/C
26	Brake Mode (Parallel/Split)	1	0.8	50.0	S	20.0	Driver	V/C
27	SOC Reset	1	0.9	50.0	S	20.0	Driver	V/C
28	Interlock	1	0.5	50.0	S	20.0	Battery	V/C
29	High Contactor Control	8	0.3	10.0	P	10.0	V/C	Battery
30	Low Contactor Control	8	0.4	10.0	P	10.0	V/C	Battery
31	Reverse and 2nd Gear Clutches	2	0.5	50.0	S	20.0	V/C	Trans
32	Clutch Pressure Control	8	0.1	5.0	P	5.0	V/C	Battery
33	DC/DC Converter	1	1.6	1000.0	P	1000.0	V/C	Battery
34	DC/DC Converter Current Control	8	0.6	50.0	S	20.0	V/C	Battery
35	12V Power Relay	1	0.7	50.0	S	20.0	V/C	Battery
36	Traction Battery Ground Fault Test	2	1.7	1000.0	P	1000.0	V/C	Brakes

Signal Number	*Signal Description*	*Size /bits*	*J /ms*	*T /ms*	*Periodic /Sporadic*	*D /ms*	*From*	*To*
37	Brake Solenoid	1	0.8	50.0	S	20.0	V/C	Brakes
38	Backup Alarm	1	0.9	50.0	S	20.0	V/C	Brakes
39	Warning Lights	7	1.0	50.0	S	20.0	V/C	Ins.
40	Key Switch	1	1.1	50.0	S	20.0	V/C	I/M C
41	Main Contactor Close	1	0.3	50.0	S	20.0	I/M C	V/C
42	Torque Command	8	0.2	5.0	P	5.0	V/C	I/M C
43	Torque Measured	8	0.1	5.0	P	5.0	I/M C	V/C
44	FWD/REV	1	1.2	50.0	S	20.0	V/C	I/M C
45	FWD/REV Ack.	1	0.4	50.0	S	20.0	I/M C	V/C
46	Idle	1	1.3	50.0	S	20.0	V/C	I/M C
47	Inhibit	1	0.5	50.0	S	20.0	I/M C	V/C
48	Shift in Progress	1	1.4	50.0	S	20.0	V/C	I/M C
49	Processed Motor Speed	8	0.2	5.0	P	5.0	I/M C	V/C
50	Inverter Temperature Status	2	0.6	50.0	S	20.0	I/M C	V/C
51	Shutdown	1	0.7	50.0	S	20.0	I/M C	V/C
52	Status/Malfunction (TBD)	8	0.8	50.0	S	20.0	I/M C	V/C
53	Main Contactor Acknowledge	1	1.5	50.0	S	20.0	V/C	I/M C

A simple attempt at implementing the problem on CAN is to map each of these messages to a CAN message. Sporadic messages generally require a latency of 20 ms or less (although Kopetz gives a required latency of 5ms for one sporadic message). These messages may be queued infrequently (for example, it is reasonable to assume that there at least 50 ms elapses between brake pedal depressions). The benchmark does not give periods for these messages, and so we assume a period of 50ms for all sporadic messages.

We mentioned in section 2 how a special 'output task' could be created for each message with the job of merely queuing the pre-assembled message, and we assume this model is adopted for the benchmark system analysed here. The following table lists the messages in order of priority (*i.e.* in $D - J$ order), and gives the worst-case latencies as computed by the analysis of the previous section. The signal numbers in **bold** indicate that the signal is a sporadic one. The symbol × indicates that the message fails to meet its latency requirements (*i.e.* the message is not guaranteed to always reach its destinations within the time required); the symbol '—' indicates that no valid response time can be found because the message is not guaranteed to have been sent before the next is queued (*i.e.* $R > D - J$).

Signal N°.	*Size /bytes*	*J /ms*	*T /ms*	*D /ms*	*R (125Kbit/s)*	*R (250Kbit/s)*	*R (500Kbit/s)*	*R (1Mbit/s)*
14	1	0.1	50.0	5.0	1.544	0.772	0.386	0.193
9	1	0.2	5.0	5.0	2.048	1.024	0.512	0.256
49	1	0.2	5.0	5.0	2.552	1.276	0.638	0.319
42	1	0.2	5.0	5.0	3.056	1.528	0.764	0.382
8	1	0.1	5.0	5.0	3.560	1.780	0.890	0.445
7	1	0.1	5.0	5.0	4.064	2.032	1.016	0.508
43	1	0.1	5.0	5.0	4.568	2.284	1.142	0.571
11	1	0.1	5.0	5.0	× 5.072	2.536	1.268	0.634
32	1	0.1	5.0	5.0	× —	2.788	1.394	0.697
29	1	0.3	10.0	10.0	× 10.112	3.040	1.520	0.760
30	1	0.4	10.0	10.0	× —	3.292	1.646	0.823
53	1	1.5	50.0	20.0	× 25.232	3.544	1.772	0.886
48	1	1.4	50.0	20.0	× 29.768	3.796	1.898	0.949
46	1	1.3	50.0	20.0	× 39.344	4.048	2.024	1.012
44	1	1.2	50.0	20.0	× 39.848	4.300	2.150	1.075
40	1	1.1	50.0	20.0	× —	4.552	2.276	1.138
39	1	1.0	50.0	20.0	× —	4.804	2.402	1.201
27	1	0.9	50.0	20.0	× —	7.072	2.528	1.264

Signal N°.	*Size /bytes*	*J /ms*	*T /ms*	*D /ms*	*R (125Kbit/s)*	*R (250Kbit/s)*	*R (500Kbit/s)*	*R (1Mbit/s)*
38	1	0.9	50.0	20.0	× —	7.324	2.654	1.327
37	1	0.8	50.0	20.0	× —	7.576	2.780	1.390
52	1	0.8	50.0	20.0	× —	7.828	2.906	1.453
26	1	0.8	50.0	20.0	× —	8.080	3.032	1.516
35	1	0.7	50.0	20.0	× —	8.332	3.158	1.579
51	1	0.7	50.0	20.0	× —	8.584	3.284	1.642
22	1	0.7	50.0	20.0	× —	8.836	3.410	1.705
34	1	0.6	50.0	20.0	× —	9.088	3.536	1.768
20	1	0.6	50.0	20.0	× —	9.340	3.662	1.831
50	1	0.6	50.0	20.0	× —	9.592	3.788	1.894
31	1	0.5	50.0	20.0	× —	9.844	3.914	1.957
47	1	0.5	50.0	20.0	× —	12.616	4.040	2.020
28	1	0.5	50.0	20.0	× —	12.868	4.166	2.083
19	1	0.5	50.0	20.0	× —	13.120	4.292	2.146
25	1	0.4	50.0	20.0	× —	13.372	4.418	2.209
17	1	0.4	50.0	20.0	× —	13.624	4.544	2.272
45	1	0.4	50.0	20.0	× —	13.876	4.670	2.335
24	1	0.3	50.0	20.0	× —	14.128	4.796	2.398
16	1	0.3	50.0	20.0	× —	14.380	4.922	2.461
18	1	0.3	50.0	20.0	× —	14.632	6.056	2.524
41	1	0.3	50.0	20.0	× —	14.884	6.182	2.587
23	1	0.2	50.0	20.0	× —	17.152	6.308	2.650
15	1	0.2	50.0	20.0	× —	17.404	6.434	2.713
6	1	0.9	100.0	100.0	× —	17.656	6.560	2.776
4	1	0.8	100.0	100.0	× —	17.908	6.686	2.839
2	1	0.7	100.0	100.0	× —	18.160	6.812	2.902
1	1	0.6	100.0	100.0	× —	18.412	6.938	2.965
12	1	0.4	100.0	100.0	× —	18.664	7.064	3.028
10	1	0.2	100.0	100.0	× —	18.916	7.190	3.091
36	1	1.7	1000.0	1000.0	× —	19.168	7.316	3.154
33	1	1.6	1000.0	1000.0	× —	19.420	7.442	3.217
13	1	1.2	1000.0	1000.0	× —	19.672	7.568	3.280
5	1	1.1	1000.0	1000.0	× —	22.444	7.694	3.343
3	1	1.0	1000.0	1000.0	× —	22.696	7.820	3.406
21	1	0.3	1000.0	1000.0	× —	22.948	7.946	3.469

There is a problem with the approach of mapping each signal to a CAN message approach: the V/C subsystem transmits more than 14 message types, and so the 82527 cannot be used (recall that there are 15 slots in the 82527, one of which is a dedicated receive slot). We will return to this problem shortly.

As can be seen, at a bus speed of 125Kbit/s the system cannot be guaranteed to meet its timing constraints. To see the underlying reason why, consider the following table:

Bus Speed	*Message Utilisation*	*Bus Utilisation*	α
125 Kbit/s	15.91%	125.29%	—
250 Kbit/s	7.95%	62.64%	1.14
500 Kbit/s	3.98%	31.32%	3.09
1Mbit/s	1.99%	15.66%	5.79

The ‘message utilisation’ is calculated using the number of data bytes in a given CAN message. The ‘bus utilisation’ is calculated by using the total number of bits (including overhead) in a given CAN message. The column headed α details the *breakdown utilisation* [8] of the system for the given bus speed. The breakdown utilisation is the largest value of α such that when all the message periods are divided by α the system remains schedulable (*i.e.* all latency requirements are met). It is an indication of how much slack there is in the system: a value of α close to but greater than 1 indicates that although the system is schedulable, there is little room for increasing the load. The symbol ‘—’ for α for the bus speed of 125Kbit/s indicates that no value for breakdown utilisation can be found, since even $\alpha = 0$ still results in an unschedulable system.

As can be seen, there is a large difference between the message and bus utilisations. This is because of the relatively large overhead of a CAN message. At a bus speed of 125Kbit/s the bus utilisation is greater than 100%, and it is therefore no surprise that the bus is unschedulable.

One way of reducing the bus utilisation (and the message utilisation) is to ‘piggyback’ messages sent from the same source. For example, consider the Battery subsystem: this periodically sends four single byte messages each with a period of 100 ms (message numbers 1, 2, 4, and 6). If we were to collect these into a single message then we could sent one four byte message at the same rate. This would reduce the overhead, and hence the bus utilisation. Another advantage with piggybacking is that the number of slots required in the bus controller is reduced (we have only 14 slots available with the 82527 CAN controller, and the V/C subsystem has more than 14 signals).

It is also possible to piggyback signals that are not neccesarily generated together (for example, sporadic signals). The approach we take is to send a ‘server’ message periodically. A sporadic signal to be sent is stored in the memory of the host CPU. When the ‘server’ message is to be sent, the sender task polls for signals that have occurred, and fills the contents of the message appropriately. 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 or longer, a server message with a period of 10ms and a worst-case response time of 10ms would be sufficient. Alternatively, a server message with a period of 15ms and a worst-case response time of 5ms could be used.

We transform the set of messages to include these server messages. We choose server messages with periods of 10ms and latency requirements of 10ms. The following table lists the messages in the system:

Signal N^{o}s	*Size /bytes*	*J /ms*	*T /ms*	*D /ms*	*R* (*125Kbit/s*)	*R* (*250Kbit/s*)	*R* (*500Kbit/s*)	*R* (*1Mbit/s*)
14	1	0.1	50.0	5.0	1.544	0.772	0.386	0.193
8,9	2	0.1	5.0	5.0	2.128	1.064	0.532	0.266
7	1	0.1	5.0	5.0	2.632	1.316	0.658	0.329
43,49	2	0.1	5.0	5.0	3.216	1.608	0.804	0.402
11	1	0.1	5.0	5.0	3.720	1.860	0.930	0.465
32,41	2	0.1	5.0	5.0	4.304	2.152	1.076	0.538
31,34,35,37,38,39,40,44,46,48,53	6	0.2	10.0	10.0	5.192	2.596	1.298	0.649
23,24,25,28	1	0.2	10.0	10.0	8.456	2.848	1.424	0.712
15,16,17,19,20,2226,27	2	0.2	10.0	10.0	9.040	3.140	1.570	0.785
41,43,45,47,49,50,51,52	3	0.2	10.0	10.0	9.696	3.468	1.734	0.867
18	1	0.2	50.0	20.0	10.200	3.720	1.860	0.930
1,2,4,6	4	0.3	100.0	100.0	19.088	4.088	2.044	1.022
12	1	0.3	100.0	100.0	19.592	4.340	2.170	1.085
10	1	0.2	100.0	100.0	20.096	4.592	2.296	1.148
3,5,13	3	0.4	1000.0	1000.0	28.904	4.920	2.460	1.230
21	1	0.3	1000.0	1000.0	29.408	6.552	2.586	1.293
33,36	1	0.3	1000.0	1000.0	29.912	6.804	2.712	1.356

There are two sporadic signals that remain implemented by sporadic messages: Signal 14 has a deadline that is too short to meet by polling. Signal 18 is the only sporadic sent from the Brakes subsystem, and cannot therefore be piggybacked with other sporadic signals.

The following table gives the utilisation and breakdown utilisation of the above system:

Bus Speed	*Message Utilisation*	*Bus Utilisation*	α
125 Kbit/s	18.46%	84.44%	1.011
250 Kbit/s	9.23%	42.22%	1.981
500 Kbit/s	4.62%	21.11%	3.812
1Mbit/s	2.31%	10.55%	7.082

As can be seen, the piggybacking of messages leads to a reduction in overheads, and hence a reduction in bus utilisation. In general, this in turn leads to increased real-time performance. For all four bus speeds, all messages will meet their deadlines.

5. DISCUSSION AND CONCLUSIONS

The analysis reported in this paper enables the CAN protocol to be used in a wide range of real-time applications. Indeed its use of system-wide priorities to order message transmissions makes it an ideal control network. The use of a global approach to priorities also has the advantage that the wealth of scheduling analysis developed for fixed priority processor scheduling can be easily adapted for use with CAN. An important extension to the analysis presented here is the inclusion of phasing relationships between messages sent from the same and from different stations; this combined with global time would enable the engineering of systems with shorter worst-case response times.

Tools already exist that embody processor scheduling, and similar tools could be developed for CAN. These would not only accurately predict the worst case message latencies (for all message classes in the system) but could also be used, by the systems engineer, to ask “what if” questions about the intended application (indeed, a such a tool was used to transform the SAE ‘benchmark’ so that it would fit into a system running at 125 Kbit/sec).

By applying the analysis to an existing benchmark an assessment of its applicability has been made. However, the benchmark does not illustrate all of the advantages the full flexibility of CAN can provide when supported by priority based analysis. In particular, sporadic messages with tight deadlines but long inter-arrival times can easily accommodated. It is also possible to incorporate many different failure models and to predict the message latencies when different levels of failure are being experienced: because of space limitations in this paper we have been unable to include details of how to incorporate the delays due to re-transmission of messages after a detected error. However, the report [12] describes the required extensions to the analysis, and shows how the SAE ‘benchmark’ behaves under certain error rate assumptions.

In this short paper we have dispelled some of the major misconceptions about the real-time performance of CAN, and have shown that the SAE ‘benchmark’ could be easily accommodated in a CAN-based system.

6. REFERENCES

[1] “*Class C Application Requirement Considerations*”, SAE Technical Report J2056/1 (June 1993)

[2] “*Survey of Known Protocols*”, SAE Technical Report J2056/2 (June 1993)

[3] 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)

[4] Burns, A., Nicholson, M., Tindell, K. and Zhang, , “*Allocating and Scheduling Hard Real-Time Tasks on a Point-to-Point Distributed System*”, Proceedings of the The Workshop on Parallel and Distributed Real-Time Systems, pp. 11-20, Newport Beach, California (April 13-15 1993).

[5] Burns, A., Tindell, K., Wellings, A., “*Fixed Priority Scheduling with Deadlines Prior to Completion*”, Proceedings Sixth Euromicro Workshop on Real-time Systems, IEEE Computer Society Press (June 1994)

[6] Kopetz, H., "*A Solution to an Automotive Control System Benchmark*", Institut für Technische Informatik, Technische Universität Wien, research report 4/1994 (April 1994)

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

[8] Lehoczky, J., Sha, L. and Ding, Y., "*The Rate Monotonic Scheduling Algorithm: Exact Characterisation and Average Case Behaviour*", *Proceedings of the Real-Time Systems Symposium* (December 1989).

[9] Tindell, K., "*Fixed Priority Scheduling of Hard Real-Time Systems*", YCST 94/03, DPhil Thesis, Department of Computer Science, University of York (1993).

[10] Tindell, K., and Clark, J., "*Holistic Schedulability Analysis for Distributed Hard Real-Time Systems*", *Microprocessors and Microprogramming* **40**, pp. 117-134 (1994)

[11] Tindell, K., Burns, A., Wellings, A., "*Analysis of Hard Real-Time Communications*", *Real-Time Systems* (to appear); also appears as technical report YCS 222, Department of Computer Science, University of York (January 1994)

[12] Tindell, K., Burns, A., "*Guaranteed Message Latencies for Distributed Safety Critical Hard Real-Time Networks*", Technical report YCS 229, Department of Computer Science, University of York, England (May 1994)