REPORT ON THE SOFTWARE ARCHITECTURE OF PATH'S AUTOMATED VEHICLE CONTROL

by

Stavros Tripakis

Memorandum No. UCB/ERL M01/6

29 January 2001

REPORT ON THE SOFTWARE ARCHITECTURE OF PATH'S AUTOMATED VEHICLE CONTROL

by

Stavros Tripakis

Memorandum No. UCB/ERL M01/6

29 January 2001

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Report on the Software Architecture of PATH's Automated Vehicle Control

Stavros Tripakis

January 29, 2001

Abstract

We report on the software architecture of PATH's automated vehicle control. The architecture is responsible for the longitudinal and lateral control of each vehicle in a *platoon* (sequence of vehicles, closely spaced, at high speeds). The architecture consists of a set of processes running concurrently on a PC, reading data from various sensors (e.g., radar, speedometer, accelerometer, magnetometer), writing to actuators (throttle, brake and steering), and using radio to communicate data to other vehicles. The processes exchange data with each other using a *publish/subscribe* scheme.

We describe the architecture, and identify chains of computation that can be seen as real-time tasks. We estimate the task latencies and compute the total CPU utilization, which is found to be less than 70%. We also perform a more sophisticated schedulability analysis to check whether the deadlines of the tasks are met.

In the appendix, we describe the API for the Publish/Subscribe Library. We also give a list of the control variables used in the current architecture.

1 Introduction

PATH's [1] Advanced Vehicle Control and Safety Systems (AVCSS) project involves the design and implementation of automated vehicle control applications on a variety of vehicles, such as cars (Ford's, Buick's), trucks, or snow-plows.

In this document, we focus on the platoon application, [2] where the architecture is responsible for controlling a set of cars moving autonomously in a *platoon* formation (one car behind the other, with a small distance, e.g., 4-6 meters, between them), on the highway and at high speed (e.g., 65 miles/hour). The supporting highway infrastructure consists of a sequence of magnets placed on the center of a lane (typically 1.2 meters apart).

The control functions can be divided into lateral and longitudinal control. The *lateral* control is responsible for keeping the car in the center of the lane, by reading magnet relative position information from the car's magnetometer and controlling the steering. The *longitudinal* control is responsible for maintaining a safe but short distance between the cars and for keeping the platoon stable. It does this by controlling braking and acceleration, using input information from the car's radar and other sensors, as well as information about the speed and acceleration of the car in front and the lead car of the platoon. This information is distributed among cars in the platoon using wireless communication.

[1] PATH (Partners for Advanced Transit and Highways) is a research lab administered by the Institute of Transportation Studies (ITS), University of California, Berkeley, in collaboration with Caltrans [8].

[2] However, the pattern followed by this architecture is the same as for those of other types of vehicles and other applications.) The parts that change are the control software modules and pieces of hardware which may be specific to a particular application and type of vehicle.

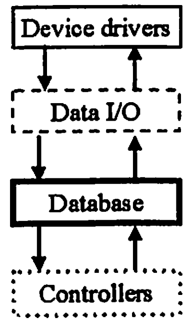

Figure 1: Process types in the automated vehicle control software architecture.

In this paper, we describe the software architecture of the above system, which consists of a set of processes running on the control computer (a PC) on each vehicle. All the software is written in C and runs on the QNX real-time operating system. The processes include: *device drivers*, *controllers*, and *data I/O processes.* The device drivers interact directly with the hardware. The data I/O processes transform data from the device drivers into high-level C structures to be read by the controllers, and also transform high-level output data written by the controllers into low-level data for the device drivers. The controllers read high-level sensor data and compute high-level actuator data.

The controllers interact with the data I/O processes via a *publish/subscribe* inter-process communication library. This is essentially a centralized database, providing to its clients (processes) the possibility to register/deregister, create/destroy variables, read/write variables, and ask to receive notifications when a variable is updated.

Figure 1 shows the interaction between the different types of processes and the database.

The purpose of this paper is two-fold. First, to present a real embedded software architecture, which has been successfully used to implement non-trivial control functions in non-trivial applications. Our interest is not in the hybrid controllers themselves, but rather in their implementation. We believe that this implementation follows a pattern found in many similar control applications, namely, the Publish/Subscribe scheme. This is not surprising, since this scheme has a number of features particularly attractive for control applications, such as loose coupling of producer/consumer processes, automatic over-writing of old data and update notifications.

The second objective of the paper is to study the properties of the current implementation. In particular, we are interested in verifying whether the architecture meets its real-time requirements, in terms of *deadlines*. We argue that an attempt to verify the architecture using formal method techniques, such as, for example, model checking, is extremely hard, mainly because of the complexity of modeling the operating system functions. Instead, we use a number of schedulability analysis results from fixed-priority scheduling theory. We find that the CPU utilization is below 70%, which is only a necessary condition for correctness, since the architecture does not follow the simple periodic task model. We therefore use more sophisticated types of schedulability analysis which cover synchronization constraints and varying priorities within a task. We identify a potential problem with the architecture, where the deadline of a task is not guaranteed.

We start by briefly presenting the hardware architecture (section 2). We then describe the Publish/Subscribe library in section 3. We present the software architecture in section 4. The analysis is contained in section 5. Section 6 contains the conclusion.

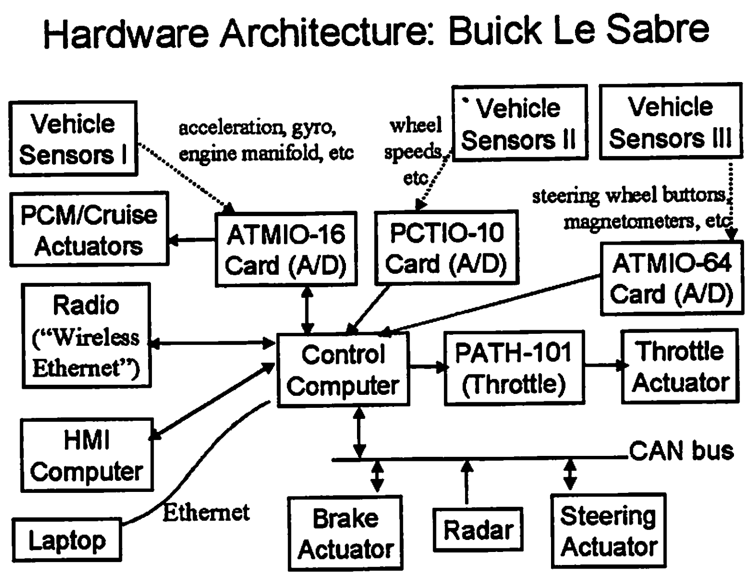

Figure 2: Automated vehicle control: hardware architecture

2 Hardware Architecture

For a better understanding of the software, we start by briefly presenting the hardware equipment of the Buick Le Sabre vehicles, which are the ones used for car automated control (Figure 2). The boxes represent different pieces of hardware. The arrows represent connections of these pieces, and the direction of the arrows represents data flow: for example, the control computer takes input from the radar but not vice-versa.

The control computer is a 166 MHz Pentium PC. The "sensors" boxes I, II, III, are analog circuits taking inputs from accelerometer, magnetometers, and so on. The ATMIO-16, ATMIO-64 and PCTIO-10 cards are essentially digital/analog converter boards, equipped also with timers. PATH-101 is a card developed at PATH to control the throttle actuator. The other two actuators, brake and steering are connected to the control computer through a CAN bus, through which they receive control messages and send back status information. The radar (installed in the front of the vehicle) is also connected to the CAN bus. The laptop is used for initialization. The Human Machine Interface (HMI) computer provides status display to the passengers in the car.

3 The Publish/Subscribe Architecture

In this section we briefly describe the Publish/Subscribe architecture, which is used for communication between data I/O and control processes, as mentioned in the introduction. The architecture is implemented as a C library on top of QNX. It has been used in various automated vehicle control projects (however, it is generic enough to be used in other applications as well).

The library offers the service of a *centralized database* to a set of processes running on the same host. The processes using the database are called *clients*. The database is a means for *asynchronous* inter-process communication, in the sense that a process producing data can write it to the database without worrying who the potential consumers might be, and at what pace they read the data. Consumers are also guaranteed to read the most recent value of data, which is of particular interest to control applications, where old data is often useless. Finally, the architecture is modular, in the sense that different software components built separately can interface in a clear way through the database.

The name Publish/Subscribe was chosen because in addition to typical database operations the library also offers the possibility for clients to request to be notified whenever a variable is updated: these notifications are called *triggers* and can be seen as messages that are sent to a client process from the database. The messages are buffered in FIFO order, until the client calls the QNX primitive `Receive()` to retrieve the first message in the buffer. If there is no pending message, the client blocks until a message arrives.

In summary, the services offered by the publish/subscribe library are:

- Register/deregister with the database (primitives `clt_login()`, `clt_logout()`).
- Create/destroy a variable (primitives `clt_create()`, `clt_destroy()`).
- Read a variable (primitive `clt_read()`).
- Write a variable (primitive `clt_update()`).
- Set/unset triggers for variables (primitives `clt_trig_set()`, `clt_trig_unset()`), receive notification messages (QNX system call `Receive()`) and check which variable they are meant for.

3.1 Semantics and Properties of the Publish/Subscribe Library

We can view the Publish/Subscribe primitives that interact with the database (e.g., `clt_create()`, `clt_read()` or `clt_update`) as requests that the clients of the service place to the server (the database). These requests are *atomic*, which means that the database will complete serving a request (receive the command, execute it, return the result) until it proceeds with the next request (that is, the database *serializes* the requests).

Atomicity ensures in particular database *integrity*, for example, that the value read by a client is not modified during the reading process.

Another property derived from atomicity is that `clt_update` always returns the most recent value of the variable in question.

Conceptually, the Publish/Subscribe library does not offer any *fairness* guarantees to clients. This will generally depend on the underlying operating system and in particular its scheduling policy. For example, in a priority based scheduling policy (such as the one used in QNX), it is possible that some high priority processes monopolize the database, so that a low priority process *starves* (i.e., never gets to place a request).

Another thing to notice is the possibility of having more than one trigger messages buffered. Since process execution depends on the scheduler, a variable might be updated more than once before a process that has set a trigger for this variable is waken up. This means that when this process wakes up, it may have more than one trigger messages pending in its input buffer.

3.2 Implementation of the Publish/Subscribe Library

The Publish/Subscribe library is implemented using the blocking message-passing facilities provided by the QNX microkernel, through the system calls `Send()`, `Receive()`, `Reply()`. Quoting from [11]:

- A process that issues a `Send()` to another process will be blocked until the target process issues a `Receive()`, processes the message, and then issues a `Reply()`.
- If a process executes a `Receive()` without a message pending, it will block until another process executes a `Send()`.
- These primitives copy data directly from process to process without queuing.

The database of the Publish/Subscribe library is implemented as a QNX process. This process executes the following loop: call `Receive()` and block waiting for requests from clients; upon reception of a request, process that request; send back the result using `Reply()` and return to the beginning of the loop.

A request such as `clt_login`, `clt_create`, `clt_read` and so on, is implemented, from the clients side, as a `Send()` to the database process.

Triggers are implemented using the `Trigger()` system call of QNX. This is the *non-blocking* version of `Send()`. That is, a process calling `Trigger()` sends a message to another process and continues execution as normal. If the other process is in the Receive-blocked state, it will be waken up, otherwise, the message will be buffered until that process calls `Receive()`. Whenever the database receives a `clt_update` request, it updates the variable in question, and then goes through the (possibly empty) list of processes that have set a trigger for this variable. For each process in that list, it calls `Trigger()`. After going through the entire list, the database sends a `Reply()` to the process that originated the update.

4 Software Architecture

A first diagram of the set of processes and their interaction appears in Figure 3. The device drivers are `pctio10` (PCTIO-10 card), `atmio16` (ATMIO-16 card), `atmioe` (ATMIO-64 card), `path101` (PATH-101 card), `cani` (CAN bus interface), and `radiodriver` (not shown in the figure).

The data I/O processes are the ones that deal with data acquisition, processing and output. They retrieve data from the device drivers, process it and store it in the database in a format that the control processes can use (i.e., C structures). They also retrieve from the database the control output produced by the control processes and write it to the device drivers. The data I/O processes talk to the device drivers using *synchronous message passing* [3]. That is, the device driver blocks waiting for a read/write message from a data I/O process, receives such a message, process it by writing to the hardware, and replies back. One the other direction, some device drivers have associated interrupt handlers which get invoked whenever a hardware interrupt is raised by the device, and send an asynchronous (non-blocking) message to a data I/O process. The latter can then read data from the device. The data I/O processes are `veh_iols`, `canread`, `canbrake`, `cansteer`, `veh_lat`, `radio` and `hmi`.

The control processes are `eng_spdls` (longitudinal control) and `hst` (lateral control). The process `buttons` can also be seen as a control process, since it only interacts with the database. This process retrieves steering-wheel button activation data and current button status data from the database, computes new button status data and writes it back into the database.

Figure 3 also shows the variables exchanged by data I/O and control processes. These variables are actually created and stored in the database. Each arrow labeled with a variable means that the originator of the arrow updates the variable in the database, and the target of the arrow reads the variable from the database. Notice that there is a single producer for (process that updates) each

[3] Implemented by `Send()`, `Receive()`, `Reply()` system calls.

variable. The exact information contained in the variables is not important for this document. For example, long_radar contains the range (in meters) to the nearest object in the front of the vehicle (presumably car in front), long_brake contains requested and achieved brake pressure, long_input contains acceleration (in meters/sec^2), engine speed (in rpm), and so on.

All processes are implemented following the same pattern: an infinite loop which starts with a blocking Receive call, waiting for a message; once the message is received, the process wakes up, performs its function, and then goes back at the beginning of the loop. The source of the message can be either a timer or the database. Accordingly, we classify processes into *time-driven* (in fact, periodic) and *trigger-driven*.

Time-driven processes wake up and perform their function periodically. In Figure 3, time-driven processes are labeled with a period in msec. The periodic source can be either the operating system (e.g., canbrake sets a software timer asking the operating system to be sent a message every 8 ms), or external hardware that raises an interrupt (e.g., atmio16 receives an interrupt generated by a timer on the ATMIO-16 card every 20 ms), or the CAN bus or wireless interface (e.g., cani receives a message on the CAN bus from the radar every 20 ms, from the steering actuator every 8 ms, and from the brake actuator every 10 ms).

Trigger-driven processes wait for triggers for one or more variables in the database. In Figure 3, each trigger-driven process has a dashed-arrow pointing to it, labeled with the name of the variable the process sets a trigger for. For example, eng_spdls sets triggers for long_input and long_track.

Notice that the hmi process is both time driven and trigger driven: it sets a trigger for hmi_display but also wakes up periodically every 200 ms.

A final important feature of the software architecture is process scheduling. The QNX operating system uses priority scheduling [10]. Each process is assigned a priority, from 0 (lowest) to 31 (highest). At any time, a highest-priority process is chosen to run among the *ready* (i.e., non-blocked) processes [4]. The priorities are usually assigned as follows: The database process runs at priority 25. canbrake and cansteer run at priority 25. Device drivers run at priority 19 (hardware interrupt handlers are part of the device drivers, so they inherit their priority). The lateral control process hst runs at priority 18. All other processes run at priority 10 (default).

5 Analysis of the Software Architecture

The requirements of embedded software are typically described in the form of *deadlines*: a *task* must complete its execution at most x seconds after it becomes ready. In our case, we look at a task not as a single process, but as a *time-triggered chain of execution, that involves multiple processes.* Such tasks can be identified by looking at Figure 3. For example, the *lateral input* task is initiated by an interrupt from the ATMIO-64 card every 2 ms, and consists of the following execution chain: the interrupt handler (running as part of atmioe) sends a message to veh_lat; veh_lat unblocks, reads the ATMIO-64 device, and computes and updates the lat_input_mag variable in the database; this update triggers a message to be sent from the database to hst, which unblocks, reads variables lat_input_mag, lat_input_sensors and button_status, and computes and updates variables lat_output and marker_pos.

In total, we identify 11 periodic tasks: lateral input task, *steering output* task (initiated by cansteer every 4 ms), *brake output* task (initiated by canbrake every 8 ms), *steering input* task (initiated by the steering actuator every 8 ms), *brake input* task (initiated by the brake actuator

[4] If there are more than one ready processes with the same priority, then a selected scheduling algorithm will be used to divide the CPU and all ready processes with the same priority. This algorithm is specified per process, and can be one of the following three: FIFO scheduling, round-robin scheduling, or adaptive scheduling (the default). See [10] for more details.

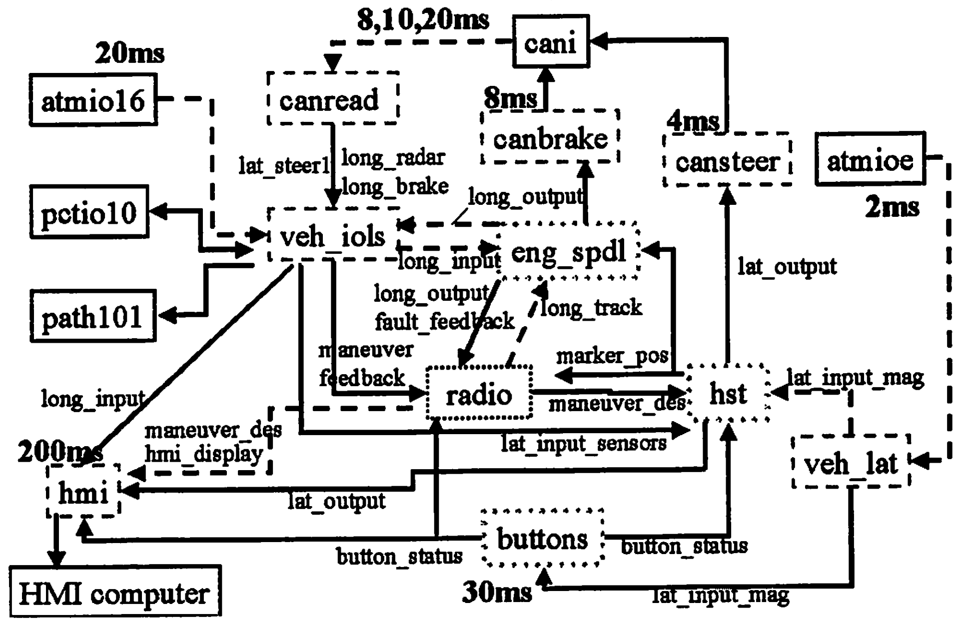

Figure 3: Automated vehicle control software architecture.

every 10 ms), *radar input* task (initiated by the radar every 20 ms), *longitudinal* task (initiated by ATMIO-16 every 20 ms), *communication input* task (initiated by messages transmitted by the other vehicles twice every 20 ms), *communication output* task (initiated by `radio` every 20 ms), *buttons* task (initiated by `buttons` every 30 ms), *human-machine interface (HMI)* task (initiated by `hmi` every 200 ms). Due to lack of space, we do not detail the operation of these tasks here. Looking at Figure 3, one can derive most of the information. Notice that the same process might be invoked twice in a task, e.g., `veh_iols` is invoked twice in the longitudinal task, first by a message from `atmio16`, then by a trigger for `long_output`.

For each of the above tasks, we impose a deadline equal to its period. For example, we require that no interrupt be raised by the ATMIO-64 card before the lateral input task triggered by the previous interrupt has fully executed [5].

It is not obvious that the software architecture meets the deadline requirements we specified above. The question then arises, how can we verify that the requirements are met? One possibility could be to use a formal verification tool such as a model-checker. However, this would require modeling the operating system scheduling, interrupt handling, message passing and other functions in great detail. We believe this to be a very hard, if not impossible, task. It is also quite possible for such an approach to suffer from the *state-explosion* problem. [6]

Instead, we engage in different types of schedulability analysis. First, we derive rough estimates of the various latencies involved in the execution of the tasks. Based on that, we compute the

[5] In general, stricter deadlines might be required: for example, it might be important for a controller to read inputs from sensors and output data to actuators immediately after the inputs become available, even if they become available not very often.

[6] Although the operating system is deterministic, the properties would have to be verified with respect to all initial time phasings of tasks, which would introduce a high degree of non-determinism.

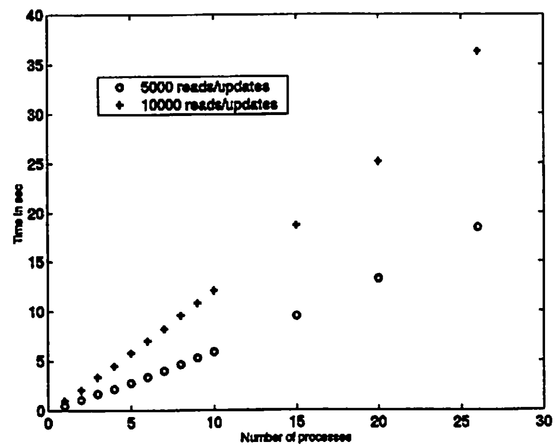

Figure 4: Performance of the Publish/Subscribe library on a 166 MHz QNX PC.

total CPU utilization induced by all tasks. This is merely a sanity check: we find that the CPU utilization is less than 70%, which is only a necessary condition for the deadlines to be met. We then perform a fixed-priority schedulability analysis introduced by a number of researchers in the real-time scheduling field, e.g., [6, 5, 2, 3, 4, 12].

Estimating execution times and other latencies: We first estimate the performance of the basic database primitives, namely, `clt_read` and `clt_update`. We conduct the following experiments, on a 166 MHz Pentium PC.[7] We run the database, then spawn a number of client processes (all with the same priority, lower than that of the database). Each client executes 20 iterations, where each iteration involves 10000 or 5000 calls to `clt_read` or `clt_update` or both (one after the other) of a large database variable (approximately 120 bytes). The total time taken to execute these calls is then divided by 10000 and averaged among processes. The results are shown in Figure 4. We see that performance grows almost linearly with the number of processes, although the slope is larger than 1. The extra overhead is probably due to context switching.

Based on the above and other measurements for reads and writes separately, we estimate the performance of the database under large load (number of clients) to be as follows:

- A `clt_read()` call takes approximately 35μsecs.
- A `clt_update()` call takes approximately 115μsecs.

We denote these latencies r and w respectively.

Apart from reads and updates to the database, tasks involve also the following latencies [8]:

- h: latency to handle a hardware interrupt.
- p: latency to send a synchronous message between processes.

[7] In fact, the experiments were done on a PC running the TCP/IP protocol stack, which adds considerable overhead. This stack does not run on the control computer in the car.

[8] We ignore floating point computation, since it is very small. In experiments we conducted, 20 million floating point operations took approximately 0.12 seconds on the 166 MHz Pentium machine. This averages to less than 1 microsecond for 1000 operations.

- t: latency to send an asynchronous trigger from the database to a client.
- c: context switching delay (includes scheduling).
- hw: hardware write (this includes sending a message to the device driver).
- hr: hardware read (this includes sending a message to the device driver).

We use the following estimates [9]: $h = 5\mu s$, $p = 50\mu s$, $t = 50\mu s$, $c = 30\mu s$, $hw = 50\mu s$, $hr = 50\mu s$. Notice that r and w already include context switching overhead, so this is not added for these operations.

CPU utilization: Based on the above estimates, we compute the total latency induced by each task. For example, let x_{li} be the total latency induced by the lateral input task. Since this task includes one hardware interrupt handler, three reads, three updates, one message sent from `atmioe` to `veh_lat`, one trigger, and three context switches, we have $x_{li} = h + 3r + 3w + p + t + 3c = 635\mu s$. Since this task is invoked every 2000 μs, the partial CPU utilization induced by it is $\frac{635}{2000} = 0.3175$. Similarly, we find the latencies induced by all other tasks: $x_{so} = x_{bo} = r + hw + p + 2c$, $x_{si} = x_{bi} = x_{ri} = h+w+hr+p+2c$, $x_{lon} = 5r+3w+2hr+p+2t+2hw+4c$, $x_{ci} = h+2hr+3w+2r+hw+8p+t+9c$, $x_{co} = 5r + 5hw + 7p + t + 8c$, $x_{but} = 2r + w + p + c$, $x_{hmi} = 5r + p + hw + c$.

Then, we can compute the total CPU utilization:

$$U = 10^{-3} \cdot \left(\frac{x_{li}}{2} + \frac{x_{so}}{4} + \frac{x_{bo}}{8} + \frac{x_{si}}{8} + \frac{x_{bi}}{10} + \frac{x_{ri}}{20} + \frac{x_{lon}}{20} + \frac{2x_{ci}}{20} + \frac{x_{co}}{20} + \frac{x_{but}}{30} + \frac{x_{hmi}}{200}\right) \approx 0.691$$

We see that $U < 1$. In fact, $U < 0.693$, which a sufficient condition for a set of periodic tasks scheduled according to the *rate-monotonic* algorithm not to miss their deadlines [6]. However, our tasks do not fit the simple model of the rate-monotonic algorithm. First, they consist of processes (subtasks) which run at different priorities. Second, they synchronize (block) during their execution on a shared resource: the database. Therefore, the above condition is merely a necessary condition for schedulability, and not a sufficient one.

Schedulability analysis: We now perform a more sophisticated schedulability analysis, taking into account the synchronization of the tasks, as well as the fact that each task is a sequence of subtasks running at different priorities. For example, the lateral input task can be viewed as a sequence of 13 subtasks, with priorities: $19 \rightarrow 10 \rightarrow 25 \rightarrow 18 \rightarrow 25 \rightarrow 18 \rightarrow 25 \rightarrow 18 \rightarrow 25 \rightarrow 18 \rightarrow 25 \rightarrow 18 \rightarrow 25$. The subsequence $18 \rightarrow 25 \rightarrow \cdots \rightarrow 25$ represents the interaction of `hst` with the database, namely reading three variables and updating two variables.

As far as synchronization is concerned, we observe the following. Since the priority of the database is set to the highest value, the database clients execute essentially the *priority ceiling protocol* [13]. In this protocol, the priority of a process that accesses a mutually-exclusive resource is temporarily raised to the priority of the resource. Here, the resource is the database, which can serve only one request at a time (hence the mutual exclusion). And the fact that when a process executes a read or update, control is passed to the database process, is equivalent to raising temporarily the priority of the task to 25.

It was shown in [13] that the priority ceiling protocol ensures absence of deadlocks, and also that a process can be blocked by a lower-priority process for at most the duration of one critical section (in our case, at most $\max\{r, w\}$).

Regarding the fact that a task consists of subtasks, we will use the so-called *HLK analysis* [2]. This technique extends the *completion time test* introduced in [5] for the basic rate-monotonic model.

[9] We believe these to be conservative. They are based on information from [11].

Due to lack of space, we will not present these techniques in detail, but only explain the intuition through our case study.

The completion time test is a necessary and sufficient condition for a set of tasks to be schedulable. Consider first the simple case of n periodic tasks with periods $T_1, ..., T_n$, execution times $C_1, ..., C_n$, blocking times $B_1, ..., B_n$ (B_i is the longest duration of blocking that can be experience by task i due to synchronization on a mutually-exclusive resource) and decreasing priorities. Define $W_i(t) = \Sigma_{j=1}^{i} C_j \lceil \frac{t}{T_j} \rceil$. Intuitively, $W_i(t)$ represents the cumulative demand for processing by all tasks up to i, in the time interval $[0, t]$. Given task i, define the series $S_0 = \Sigma_{j=1}^{i} C_j$, and $S_{k+1} = W_i(S_k) + B_i$. Then, the completion time test says that if for some k, $S_k = S_{k+1} \leq T_i$, then task i meets its deadline. If instead, $T_i \leq S_k$ for some k, then task i is not schedulable.

In the more complicated model, where tasks are sequences of varying-priority subtasks, a similar test applies, but with some modification in the definition of the above parameters. We illustrate that by performing the test for the steering output task. This task involves the sequence $25 \rightarrow 25 \rightarrow 25 \rightarrow 19$. [2] showed that the completion time of such a task is always the same as the completion time of its *normalized form*, which has unique priority 19 (intuitively, this means that since the task can be blocked while it is executing its last subtask of priority 19, it does not matter whether the previous subtasks have higher priority).

Now, we examine the *relative priorities* of each of the other tasks with respect to the normalized form of steering output. For example, the lateral input task has relative priorities $H \rightarrow L \rightarrow H$, where H denotes higher or equal and L lower priority. Similarly, the brake output task ($25 \rightarrow 25 \rightarrow 25 \rightarrow 19$) has relative priorities H, the radar input task ($19 \rightarrow 10 \rightarrow 25$) has relative priorities $H \rightarrow L \rightarrow H$, the longitudinal task has relative priorities $H \rightarrow L \rightarrow H$, and so on.

[2] showed that the maximum blocking time B for a task is the sum of the execution times of the first subtask for all tasks of the form $H \rightarrow L \cdots$, plus the maximum of the execution times of subtasks for all tasks of the form $L \rightarrow H \cdots$. In the case of steering output, its blocking time B_{so} is computed as follows: $B_{so} = \max\{h, 3r + 2w\} + 3\max\{h, w\} + \max\{h, hw, hr\} + \max\{h, r, w\} + \max\{r, hw\} + \max\{r, w\} + \max\{r, w\} = 1125\mu s$. [10]

Having computed the blocking time, we can perform the completion time test: $S_0 = x_{bo} + x_{so} + B_{so} = 1495\mu s$. $S_1 = S_0 \leq 4$ ms, therefore, the steering output task meets its deadline.

We can perform the above analysis for other tasks as well. Doing that, we find that the lateral input task does not meet its deadline. This is because `veh_lat` has priority 10, thus all other tasks have high relative priority, which means that the blocking time for lateral input is high. Notice that this is the worst-case blocking time, with respect to all possible phasings of tasks (thus, it is likely that it arises only once every several periods), and also, that it depends on latency estimates that may be too conservative.

In practice, missing this deadline has two implications. First, it means that the lateral control output might not be updated in time. Second, that there might be more than one messages in the input buffer of `veh_lat`, corresponding to multiple interrupts: `veh_lat` will consume these messages one after the other, resulting in a series of executions of the lateral input chain, whereas one would be enough. We do not know how often the above situation arises, and how negative its effect is on the control of the vehicle. It is certain that a noticeable effect on the behavior of the vehicle (the only debugging technique typically used) has not been observed to date.

[10] The first term represents the blocking effect due to lateral input task (hst interacting with the database), the second term the blocking effect due to brake, steering and radar input, and so on.

6 Conclusion

We have described the software architecture of a real automated vehicle control application, developed at PATH. We believe that it is necessary for such architectures to be studied carefully, if the implementation of hybrid controllers is to become tightly integrated to the design process from the early steps on, so that the properties of the design are maintained throughout the development of embedded software.

We have presented here preliminary schedulability analysis results, which identified potential problems in the architecture. We plan to continue our investigation in order to confirm the results.

We would also like to develop a general methodology (e.g., automatically assigning priorities) for developing software of the above kind, such that certain real-time requirements are met.

Finally, we would like to investigate other models and languages for embedded software development, and test how suitable they are for the type of control applications like the above. In particular, we would like to compare the Publish/Subscribe scheme which relies on *run-time* scheduling by the operating system, with *compile-time* scheduling schemes such as the ones used by Esterel [1], Lustre [7], or the time-triggered architecture [15].

Acknowledgments. I am grateful to Paul Kretz from PATH who provided much help in understanding the architecture. A large part of the code has been written by him. The rest of the software has been written by other engineers at PATH. I would like to thank them all for their clearly written code. I am also grateful to Raj Rajkumar, who has suggested to me the literature on extended fixed-priority schedulability analysis.

References

[1] Esterel: `http://www.esterel.org`.

[2] Harbour, Lehoczky, and Klein. Analysis of Tasks with Varying Fixed Priorities. Proc. 12th IEEE Real-Time Systems Symposium, 1991.

[3] M.G. Harbour, M.H. Klein, R Obenza, B. Pollak, T. Ralya. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems. Kluwer, 1993.

[4] Klein, Lehoczky, and Rajkumar. Rate-Monotonic Analysis for Real-Time Industrial Computing. IEEE Computer, Jan. 1994.

[5] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In Proceedings of 8th IEEE Real-Time Systems Symposium, pages 166-171. IEEE Computer Society Press, December 1989.

[6] C. L. Liu and J. Layland. Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment. Journal of the ACM, 20(1) pp 46-61, January 1973.

[7] Lustre: `http://www-verimag.imag.fr/SYNCHRONE/lustre-english.html`.

[8] PATH: `http://www.path.berkeley.edu/`.

[9] Mobies: `http://vehicle.me.berkeley.edu/mobies/`.

[10] QNX doc: `http://www.qnx.com/literature/qnx_sysarch/index.html`.

[11] QNX doc: `http://www.qnx.com/literature/whitepapers/archoverview.html`.

[12] Sha, Rajkumar and Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. IEEE Proc., Jan 1994.

[13] Sha, Rajkumar and Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Trans. Computers, Sep 1990.

[14] Teja: `http://www.teja.com`.

[15] Time-triggered architecture: `http://www.tttech.com`.

[16] P.Varaiya. Smart Cars on Smart Roads: Problems of Control. *IEEE Transactions on Automatic Control*, 38(2):195-207, February 1993.

A The Publish/Subscribe Library Primitives

A.1 Registering and deregistering

Each process that wants to use the database must register first. This is done by calling:

```
db_clt_typ *clt_login( char *pname, char *phost, char *pserv, COMM_QNX_XPORT );
```

where:

- `pname` is the name of the process requesting to register (need not be unique, used for debugging).
- `pserv` is the database process name.
- `phost` is the hostname where the database runs (or `NULL` if this is the local host).

If the call returns `NULL`, then the call has failed. Otherwise, a handle to the database is returned, to be used with the other primitives below.

To deregister, a process calls:

```
bool_typ clt_logout( db_clt_typ *pclt );
```

where:

- `pclt` is the handle to the database obtained upon registering.

`TRUE` is returned if the call succeeds, and `FALSE` if it fails.

A.2 Creating and destroying variables

What are variables: The database is a place that stores and allows access to *variables*. In the publish/subscribe library, variables are tuples of the form

(id, type, value),

where *id* is the variable identifier, *type* is the type of the variable and *value* is the current value of the variable.

The *id* of a variable is a number (an unsigned integer). The type of a variable is a pair *(typeid, size)* where *typeid* is the type identifier (an unsigned integer) and *size* is the size of the type in bytes (an unsigned integer). The value of a variable is an array of bytes, of length *size*. Notice that the type of a variable is used only for identification purposes. As far as the database is concerned, the value of each variable is simply an array of bytes. It is the responsibility of the client to interpret this array of bytes as a meaningful data structure (usually this is done by *casting*, see below the description of `clt_read`).

For the current automated vehicle control implementation at PATH, the following is to be noted (quoted from `clt_vars.h`):

```
/*
 *      As a convention, the variable name/type space is partitioned as
 *      follows:
 *
 *      0       to      99              Used by the system.
 *      100     to      199             Reserved.
 *      200     to      299             Permanent longitudinal variables.
 *      300     to      399             Permanent lateral variables.
 *      400     to      499             Permanent communications variables.
 *      1000 to 1099    Temporary variables.
 */
```

Dynamic creation and destruction of variables: Initially, the database is empty, i.e., contains no variables. Variables can be created and destroyed on-the-fly, by any process. To create a variable with id `var`, type id `type` and type size `size`, in the database with handle `pclt`, a process calls:

```
bool_typ clt_create( db_clt_typ *pclt, unsigned var,
                          unsigned type, unsigned size );
```

`TRUE` is returned if the call succeeds, and `FALSE` if it fails.

To destroy a variable, a process calls:

```
bool_typ clt_destroy( db_clt_typ *pclt, unsigned var, unsigned type );
```

`TRUE` is returned if the call succeeds, and `FALSE` if it fails.

A.3 Reading a variable

To read a variable with id `var` and type id `type`, from the database with handle `pclt`, a process calls:

```
bool_typ clt_read( db_clt_typ *pclt, unsigned var,
                        unsigned type, db_data_typ *pbuff );
```

`TRUE` is returned if the call succeeds, and `FALSE` if it fails. If successful, the call will fill-in the variable pointed to by `pbuff`, which is a generic `db_data_typ` structure. This C structure contains the current value of the variable, plus other information such as variable id and type id, last time the variable was updated, last command applied to the variable (e.g., create, read, or update). The value of the variable is contained in the field `value.user` of the `db_data_typ` structure.

Example: Assume the client wants to read a variable of id id and type id type from database db, and that the real value of the variable is a C structure mytype. Then, the client's program includes:

```
db_data_typ db_data;
mytype *myvalue;
...
if ( clt_read( db, id, type, &db_data ) != FALSE ) {
  myvalue = (mytype *) db_data.value.user;
  ...
}
else ...
```

Notice that in the above example, myvalue is an active pointer only within the scope that db_data lives.

A.4 Writing a variable

To write a variable with id var, type id type and type size size, in the database with handle pclt, a process calls:

```
bool_typ clt_update( db_clt_typ *pclt, unsigned var,
                     unsigned type, unsigned size, void *pvalue );
```

where pvalue is a pointer to a byte array of size at least size, containing the new value to be written. TRUE is returned if the call succeeds, and FALSE if it fails.

Example: Assume the client wants to update a variable of id id and type id type from database db, and that the real value of the variable is a C structure mytype. Then, the client's program includes:

```
mytype newval;
...
if ( clt_update( db, id, type, sizeof(mytype), (void *) &newval ) != FALSE )
  ...
```

A.5 Triggers

Triggers are notifications that a process requests for variable changes. A "variable change" is synonymous to the variable being updated (by a call to clt_update). That is, *it does not necessarily mean that the new value of the variable is different than its old value.*

To request notification for variable changes is to set a trigger for that variable. To cancel that request is to unset the trigger. To receive notification means to receive a *message*: the process that has requested notification can receive the related messages by calling a QNX system call, Receive (see below). In case a process is not waiting to receive a notification message (having called Receive) the message will be queued. [11] For each variable, the database keeps track of the processes that have a trigger set on this variable. Whenever this variable is written (by clt_update()), the database sends a message to all processes above.

[11] Triggers are implemented using the qnx_proxy_attach() and Trigger() QNX operating system calls. The QNX C-library manual says that up to 65535 notification messages can be pending.

Requesting/canceling notifications: Setting/unsetting a trigger for variable with id `var` and type id `type` in database with handle `pclt` is done by the following calls:

```
bool_typ clt_trig_set( db_clt_typ *pclt, unsigned var, unsigned type );
bool_typ clt_trig_unset( db_clt_typ *pclt, unsigned var, unsigned type );
```

In both cases, `TRUE` is returned if the call succeeds, and `FALSE` if it fails.

Receiving notifications: Notifications are received through the QNX system call `Receive()`, using a special type of messages, `trig_info_typ`, defined in the library. The client calls:

```
trig_info_typ trig_msg;    /* declare a placeholder for trigger messages */
...
Receive( 0, &trig_msg, sizeof( trig_msg ) );  /* block waiting for message */
```

and *blocks* waiting for a message. That is, the call does not return until a message is received. Notice that this message might be something other than a trigger, in case the client process uses other features of QNX inter-process communication through messages.

Checking which variable the trigger is for: Since a process may have set triggers for many different variables, it generally needs to check which variable the notification was for. This is done by a call to the macro `DB_TRIG_VAR`, which gives the id of the variable the notification was for.

```
if( DB_TRIG_VAR( &trig_msg ) == VAR_1 )  /* test which variable the message is for */
 ...
else if( DB_TRIG_VAR( &trig_msg ) == VAR_2 )
 ...
```

B Control Variables

The data I/O and control processes communicate through the following variables stored in the database:

- `long_radar`: contains range (in meters) to nearest object (presumably car in front, except for lead vehicle), range rate (in meters/sec), acceleration (in meters/sec^2), diagnostics (TBD), a wrap-around counter (1-1024) counting CAN messages from radar.
- `long_brake`: contains brake pressure requested (in psi), pressure achieved (in psi), mode and system status, error codes, a wrap-around counter (1-1024) counting CAN messages from brake.
- `long_track`: contains information for the leader (first car in the platoon) and the preceding vehicle. This information includes position in the platoon, time, distance, velocity and acceleration.
- `long_input`: contains sampling/control interval (in sec), platoon position, longitudinal acceleration (in meters/sec^2), measured manifold pressure (in kpa), master cylinder pressure (in psi), engine speed (in rpm), six wheel speeds (one for each wheel in meters/sec divided by 10, plus one for each of left-front and right-rear wheels, in meters/sec divided by 1), measured throttle angle (in degrees), decoded transmission position, overall transmission ratio, system status, mode status, car id, maneuver description id, counters for brake and radar (as above).

- `long_output`: contains desired throttle angle ([0.0 - 85.0 degrees]), optional user outputs to panel meters, a set of user defined data to be broadcast, car id, maneuver feedback id, a boolean to set the beeper on/off, throttle status, radar status, brake status, desired spacing gap (in meters), present spacing gap (in meters).
- `lat_input_sensors`: contains measured steering angle in degrees of handwheel, lateral acceleration (in meters/sec^2), yaw rate (in meters/sec), longitudinal velocity (in meters/sec), longitudinal velocity count (number of clock pulses between two gear teeth), error codes, a wrap-around counter (1-1024) counting CAN messages of type 5 from steering actuator.
- `lat_input_mag`: contains voltage readings from the six magnetometers' (left, center or right, front or back) x, y and z axes, magnetometer health status monitor, voltage from the steering wheel buttons, and tail light voltage.
- `lat_output`: contains desired steering angle in degrees of handwheel (17 degrees of handwheel equal 1 degree of roadwheel), steer status, lateral position, time and distance to destination.
- `lat_steer1`: contains steer status, error code from steering actuator, handwheel position in degrees, analog roadwheel position (not used currently), a wrap-around counter (1-1024) counting CAN messages of type 5 from steering actuator.
- `lat_steer2`: contains steering actuator motor current (in amps), analog roadwheel position (in degrees), a wrap-around counter (1-1024) counting CAN messages of type 6 from steering actuator.
- `marker_pos`: contains marker number (most recently seen marker), counter of number of markers, lane number, direction (south/north), Lateral error in cm, lateral controller maneuver id, time at marker position.
- `button_status`: contains current status of buttons for turning on lateral and longitudinal control, and of button for turning off both controls.
- `maneuver_feedback`: contains car id, number of maneuver feedback.
- `maneuver_des`: contains car id, number of requested maneuver.
- `fault_feedback`: contains car id, number of cars in platoon, type of fault.
- `hmi_display`: contains display state, position of car inside platoon, fault status for communications.