

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

Stavros Tripakis

April 30, 2001

Abstract

We describe and analyze the software architecture of the automated vehicle control system [18] of PATH.¹ The software architecture consists of a set of processes running concurrently and communicating through a *publish/subscribe* database. We argue that such an architecture is suitable for many real-time control applications. We describe the implementation of one such application, responsible for the control of a number of vehicles traveling one behind the other in a *platoon* formation at close distance and at high speeds. We model the architecture in the framework of periodic tasks consisting of a sequence of sub-tasks with varying priorities [3, 4] and analyze the model, in order to verify whether the timing requirements of the application, expressed in the form of *deadlines*, 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.

In the introduction, we point out some crucial open problems which we believe could be addressed by Phase I groups participating in the *MoBIES* project.

Keywords: embedded middleware, publish/subscribe, scheduling and timing analysis, real-time control.

1 Introduction

PATH's 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, trucks, or snow-plows. One such application aims to increase the capacity of highways, by having vehicles travel in *platoons*, that is, groups of up to 10 vehicles moving one behind the other, at a close distance (e.g., 10 feet), and at high speeds (e.g., 65 miles/hour). This application has been implemented at PATH and has been successfully demonstrated numerous times, with human passengers traveling on the vehicles.

There are obviously many challenges in designing and building such a system, from providing the supporting highway infrastructure (e.g., magnets placed on the center of a

¹PATH (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.

lane to keep the vehicle in track), to designing the autonomous *lateral* and coordinated *longitudinal* controllers that operate in each vehicle.²

In this document, we focus on the embedded software architecture, which implements the above control design. Designing such an architecture is a challenge by itself. The architecture must permit an easy implementation of the controllers (especially since this implementation is usually done by control engineers). It must also facilitate modularity and re-use of components, since a variety of hardware equipment is used for different vehicles, each requiring different software components as well (e.g., device drivers), while some of the controller components must be fine-tuned as well, to meet the physical parameters of each vehicle. Finally, the architecture must also be amenable to analysis, to guarantee the timing requirements of the controllers.³

The objectives of this paper are three. First, to describe a real embedded software architecture of an automated vehicle control (AVC) system, which we believe is suitable for many similar real-time control applications. Second, to present a model of this architecture and state its timing requirements. Third, to perform a formal analysis of the architecture, in order to check whether these requirements are met. For this, we apply results from the real-time scheduling theory, and in particular [3, 4].

The AVC software architecture consists of a set of processes communicating through a *publish/subscribe* middleware. The latter is implemented in C on top of the real-time operating system QNX [11]. It is essentially a database which allows processes to create, read and write variables, as well as to request notification whenever a given variable is updated. In Section 3 we describe the publish/subscribe architecture and argue that it has a number of important properties with respect to the requirements discussed above, namely, it is modular, generic and inherently asynchronous (producers and consumers need not know about each other and can work at different rates). Therefore, it is especially suitable for real-time control applications such as automated vehicle control.

Although a remarkable piece of engineering, the AVC software architecture has not been designed with a formal model in mind and it has only been informally tested by collecting execution traces. In Section 4 we develop a model for the particular AVC application of automated platoons, mentioned above. Our model includes, except from the *physical* QNX processes that implement the application, a set of real-time *logical tasks*, which represent the formal requirements of the system. For instance, a (logical) task might be: “every 5ms,

²The *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.

³Another requirement is obviously fault-tolerance, since this is a safety-critical system. Our work does not deal with fault-tolerance. There are two ways in which this is achieved in the PATH system. First, the controllers are designed to monitor the sensors and diagnose faults in them (e.g., speed reported by the speedometer does not match speed calculated by speed of front vehicle, relative distance and acceleration), where-upon they go into a fault-mode (or abort automatic operation completely). Second, a human driver is present who can take over by manually switching off automatic control in case of a serious failure (e.g., of the control computer).

sample the radar data and compute new throttle output”. By essentially reverse-engineering the software, we identify a number of such tasks. All tasks are time-triggered and periodic. Each task is realized at the implementation level by a chain of execution of QNX processes, each running at its own priority. For instance, the task above might involve a process *A* to do the sampling and store the data in the database, a process *B* to read the data and compute the control output, and a process *C* to write the output to the actuator. Processes are triggered either periodically by hardware timers (e.g., process *A* above) or by other processes through the services of the database (e.g., process *A* triggers *B* and *B* triggers *C*).

In Section 5, we use results from real-time scheduling theory to cast our application into the model of periodic tasks consisting of sub-tasks with varying priorities of [3, 4]. We then check whether the *deadlines* of the logical tasks are met. The deadline of a task is set to be equal to its period, and specifies the requirement that an instance of a task must be executed completely before the next instance arrives.

In order to perform the analysis, we first estimate the execution times of the various computation and dataflow operations of the software. For the estimations, we use simple (and admittedly inaccurate) techniques, such as running an operation a number of times and measuring the average execution time. However, the analysis is independent on how the execution times are computed, and more elaborate techniques such as *worst-case execution time* analysis (e.g., see [9, 16]) can be used to obtain more accurate results.

Based on the execution times we compute the total CPU utilization and find that it is approximately 74%, which is only a necessary condition for schedulability. To check whether task deadlines are indeed met, we cannot use standard *rate-monotonic* theory [8, 7], since the logical tasks of our model are composed of a sequence of sub-tasks (i.e., physical processes), each running at its own priority. Instead, we apply the so-called *HKL analysis* [3, 4, 6] to check that the deadline of a logical task is always met.

In Section 6 we present our conclusions.

Challenges for the Phase I participants of the MoBIES project: We identify a number of challenges that can motivate the work of the Phase I groups participating in the MoBIES project.

- Can the analysis of Section 5 be automated?

Some tools claim to do this (e.g.[17], we contacted them for a trial license but received no answer yet), but we have no experience with using these tools.

- In case the schedulability analysis fails (tasks are found that miss their deadlines) it is important that some meaningful feedback is provided to the user (e.g., a counter-example trace).

Also, it would be nice to get some directives as to what, if possible, can be modified (e.g., the priorities of some processes) in order for schedulability to hold.

- Computing WCET (worst-case execution times) is a critical part of the analysis process. How accurately can it be done? Can it be automated?

Hardware Architecture: Buick Le Sabre

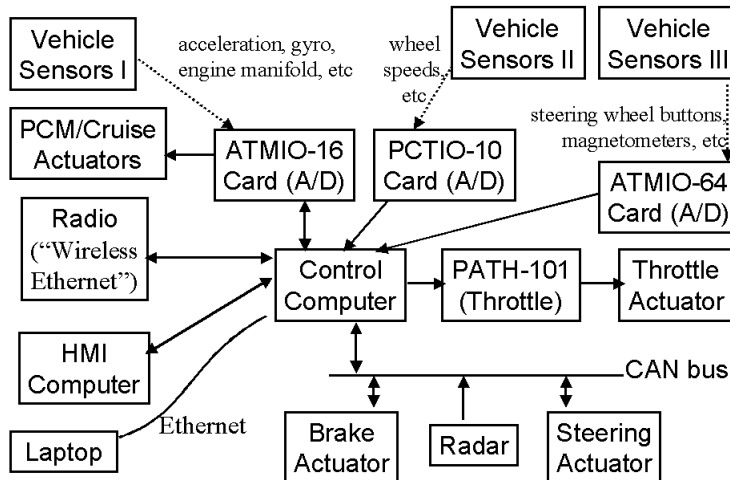


Figure 1: Automated vehicle control: hardware architecture

- Can the priorities be synthesized instead of checked?

2 Hardware Architecture

For a better understanding of the software, we start by briefly presenting the hardware equipment (Figure 1) of the Buick Le Sabre vehicles, which are the ones used in the platoon application. The boxes in the figure 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 via 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. A Lucent Wavelan 2 Mbits/sec “wireless Ethernet” interface (compliant to the IEEE 802.11 protocol) is used for inter-vehicle communication. 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 Embedded Software Architecture

The software architecture consists of a set of processes running on the control computer of each vehicle (a PC), and communicating through the *Publish/Subscribe database*. All the software is written in C and runs on the QNX real-time operating system [11].

We classify the processes (except the database process) into *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.

Figure 2 shows the interaction between the different types of processes and the database. Notice that only the data I/O and controller processes interact via the database. Device drivers and data I/O processes interact with *synchronous message passing*, that is, the reader blocks waiting for a message from the writer (the message may be generated by another process or by the handler routine of a hardware interrupt).

The publish/subscribe middleware consists of a *database server* (implemented as a QNX process) and a C library that *client* processes use to communicate with the server. The library contains primitives for a process to:

- 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()`). Setting a trigger for a variable means requesting notification whenever the variable is updated. To receive the notification messages, a process may use any of the synchronous or asynchronous `Receive()` system calls provided by QNX. There are also primitives to check which variable a specific notification refers to, in case the process has set triggers for more than one variables.

Regarding scheduling, the *static-priority scheduling* policy of QNX is used [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 database server always runs at the highest priority. In that way, the clients execute essentially the *priority ceiling protocol* [14]. In this protocol, the priority of a process that accesses a mutually-exclusive resource is temporarily raised to the priority of the resource.

⁴If 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.

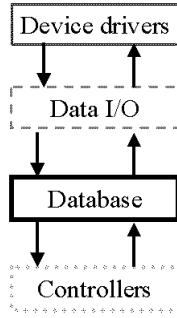


Figure 2: Automated Vehicle Control Embedded Software Architecture.

Here, the database can be seen as a resource, which can serve only one request at a time (hence the mutual exclusion). Since control is passed to the database process whenever a client executes a database request, we are effectively raising temporarily the priority of the client process to the highest priority level. It was shown in [14] 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. Here, a critical section corresponds to the database processing of a request.

The main characteristics of the software architecture are the following:

- It is modular. Data I/O and controller processes are *loosely* coupled. They do not have to know about the existence of other processes, and only interact with the database.⁵ As long as the interface with the database is respected, one or more processes in the architecture can be replaced (e.g., when updating some piece of hardware) without changing the rest of the system. Also, processes can be developed by different groups, and then integrated in a straightforward manner. (Integration is typically one of the most time-consuming tasks in the process of control software development.)
- It permits the design of *asynchronous* controllers, e.g., having different sampling rates. Since the producers and consumers of data only interact via the database, it is not a problem for the producer to be faster than the consumer (some values will be lost) or slower (some values will be read twice). This is exactly what is needed for control applications, where only the most “fresh” data are relevant.

Because of its inherent asynchrony, the architecture is particularly suitable for applications where communications are used, and where variable delays can create problems, especially in synchronous designs.⁶

In cases where synchronization is required, it can be implemented through the use of triggers. For example, a process A may request to be “waken up” whenever a process B

⁵Data I/O processes have to know the device drivers they are interacting with, but this is inevitable, since they are responsible for translating low-level into high-level data and vice versa, and the data format depends on the hardware device.

⁶Radio communication is an important piece of the platoon application, see section 4.

has updated a variable. Then B may do the same, and so on, which essentially allows the two processes to execute in *lock-step*.

- It is generic. The publish/subscribe library does not assume any fixed set of processes, or variables. It does not know the types of the variables (it only sees them as byte-strings). Therefore, it is also suitable in contexts where dynamic creation or destruction of processes is required. This is often the case in real-time control applications, where a change in the physical environment may require a change in the *control mode*. For example, failure of a sensor may require some processes to be stopped and others to be started.
- It is amenable to analysis. We perform such an analysis in section 5.

For the above reasons, we believe that the software architecture is particularly suited for many real-time control applications.

In the rest of this section, we give details on the semantics and implementation of the publish/subscribe middleware.

3.1 Semantics and Properties of the Publish/Subscribe Middleware

It is beyond the scope of this document to give formal semantics of the publish/subscribe architecture, since this would be unnecessarily complicated. We contend ourselves with an informal description.

We can view the Publish/Subscribe primitives that interact with the database (for example, `clt_create()`, `clt_read()` or `clt_update()`) as requests that the clients place to the server. 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 is ensured by the fact that the database server runs at the highest priority, therefore, cannot be interrupted by another process in the middle of execution.

In turn, atomicity ensures the *integrity* of the data stored in the database. For example, the value read by a client cannot be 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.

The Publish/Subscribe middleware does not offer any *fairness guarantees* to clients. This generally depends upon the scheduling policy of the underlying operating system and, in a static-priority scheduling case, also upon the priorities and implementation of the specific application. For example, it is possible that some high-priority processes monopolize the database, so that a low-priority process *starves* (i.e., never gets to execute).

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 Middleware

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.

Figure 4 displays an estimation of the performance of the Publish/Subscribe library on a 166 MHz QNX PC.

4 Embedded Software of the Platoon Application

We now present an instance of the embedded software architecture, for the platoon application mentioned in the introduction. In our description, we use the actual names of hardware components, software processes and variable names used in the application.

A diagram of the set of processes and their interactions appears in Figure 3. We omit the database from the figure. As mentioned before, interactions between device drivers and data I/O processes are direct, whereas interactions between data I/O processes and controllers are through the database (Figure 2).

Process types. In Figure 3, the device drivers are `pctio10` (PCTIO-10 card), `atmio16` (ATMIO-16 card), `atmioe` (ATMIO-64 card), `path101` (PATH-101 card), and `cani` (CAN

bus interface). The data I/O processes are `veh_iols`, `canread`, `canbrake`, `cansteer`, `veh_lat`, `radio` and `hmi`. The control processes are `eng_spdl` (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.

Dataflow. Figure 3 also shows the variables exchanged by data I/O and control processes. These variables are created and stored in the database. Each arrow labeled with a variable means that the origin 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 each variable, that is, each variable is updated by only one process (though it can be read by many processes).

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/sec²), engine speed (in rpm), and so on.

Time-driven and even-driven processes. All processes follow the same execution pattern, namely, 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 waiting for the next message. Messages can arrive periodically or asynchronously. Accordingly, we say that a process is *time-driven* or *event-driven*.

Time-driven processes wake up and perform their function periodically. In Figure 3, time-driven processes are labeled with a period in milliseconds. The periodic message source can be either the operating system (e.g., `canbrake` sets an operating-system timer which expires every 8 ms and generates a software interrupt which sends a message and resets the timer), or an external device that raises a hardware interrupt (e.g., `atmio16` receives an interrupt generated by a timer on the ATMIO-16 card every 20 ms, and `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).

Event-driven processes wait for triggers for one or more variables in the database. In Figure 3, each event-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_spdl` sets triggers for `long_input` and `long_track`.

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

Process priorities. The priorities of the processes have been assigned as follows. The database runs at priority 25 (highest). `canbrake` and `cansteer` run at priority 24. Device

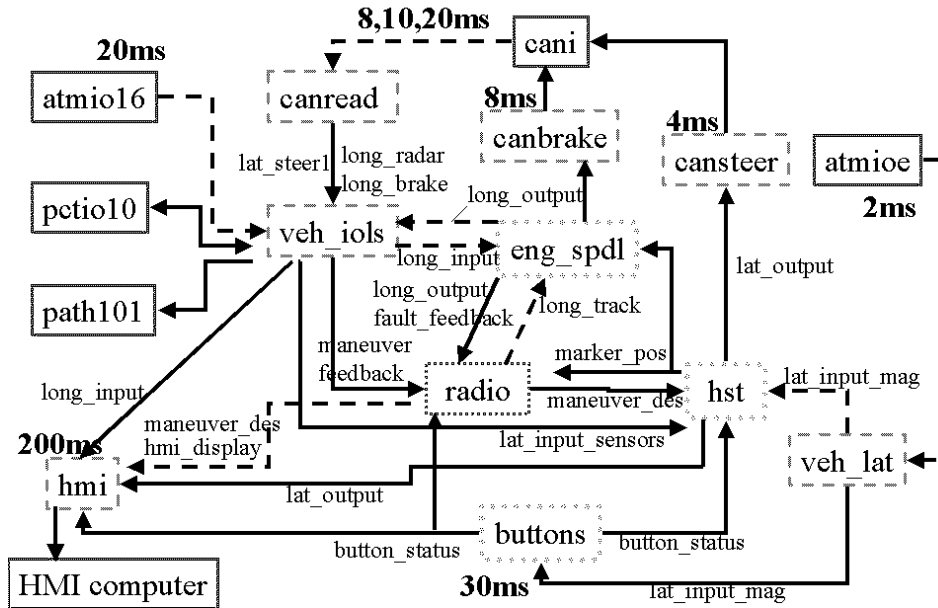


Figure 3: Automated vehicle control software architecture.

drivers run at priority 19 (hardware interrupt handlers are part of the device drivers, so they inherit their priority). `hst` and `veh_lat` run at priority 18. All other processes run at priority 10 (default).

Tasks. As mentioned in the introduction, we distinguish between the notions of *physical processes* and *logical tasks*. The former are the QNX processes that we have been presenting so far. The latter represent the formal requirements of the system, including functional and timing requirements. For instance, a task of the platoon application is: “every 2ms, sample the magnetometer data and compute new steering control outputs”.

Each task is realized at the implementation level by a “chain of execution” of different processes. For instance, the chain for the task above is as follows. Every 2 ms, the ATMIO-64 card generates a hardware interrupt, which is handled by `atmioe`. The interrupt handler sends a message to `veh_lat`. This triggers execution of `veh_lat`, which reads data from the ATMIO-64 card and updates the `lat_input_mag` variable in the database. This update triggers a message to be sent from the database to `hst`. The latter reads variables `lat_input_mag`, `lat_input_sensors`, `maneuver_des` and `button_status` from the database, and computes and updates variables `lat_output` and `marker_pos`.

By reverse-engineering the software implementation of the platoon application (i.e., browsing the code) we identify eleven tasks in total. These are listed in Table 1 and summarized below.

- Lateral input task: “every 2ms, sample the magnetometer data and compute new steering control outputs”.
- Steering output task: “every 4ms, read steering control outputs and write them to the steering actuator”.
- Brake output task: “every 8ms, read brake control outputs and write them to the brake actuator”.
- Steering input task: “every 8ms, read status data from the steering actuator and update steering control inputs”.
- Brake input task: “every 10ms, read status data from the brake actuator and update brake control inputs”.
- Radar input task: “every 20ms, read data from the radar and update radar inputs”.
- Longitudinal task: “every 20ms, sample various analog sensors and update the corresponding inputs; also use these inputs along with steering, brake and radar control inputs to compute throttle and brake outputs; finally, write throttle outputs to the throttle actuator”.
- Communication output task: “every 20ms, broadcast vehicle data (e.g., speed, acceleration)”.
- Communication input task: “every 20ms, receive vehicle data from the vehicle in front and from the lead vehicle, update radio inputs and display them on the human-machine interface”.
- Buttons task: “every 30ms, read steering data and update button control data”.
- Human-machine interface (HMI) task: “every 200ms, read display data and display them on HMI hardware”.

We do not detail here the execution chains for each of the above tasks, except for the lateral input chain (described above) and the longitudinal chain (described below). Table 1 summarizes the process chains for the other tasks.

The longitudinal chain is as follows. Every 20 ms, the ATMIO-16 card generates a hardware interrupt, which is handled by `atmio16`. The interrupt handler sends a message to `veh_iols`. The latter reads values from both the ATMIO-16 and PCTIO-10 cards, reads `long_canbrake`, `long_cansteer1` and `long_radar` from the database, and updates `long_input` and `lat_input_sensors`. The update of `long_input` causes a trigger to be sent to `eng_spdl`, which wakes up, reads `long_input`, computes `long_output` and updates it in the database. The update of `long_output` causes a trigger to be sent to `veh_iols`, which reads `long_output` from the database and writes to the throttle actuator PATH-101 and to the beeper actuator PCTIO-10.

Task	Chain
lateral input	atmioe, veh_lat, hst
steering output	cansteer, cani
brake output	canbrake, cani
steering input	cani, canread
brake input	cani, canread
radar input	cani, canread
longitudinal	atmio16, veh_iols, atmio16, pctio10, veh_iols, eng_spdl veh_iols, path101, pctio10
communication input	radio, eng_spdl, hmi
communication output	radio
buttons	buttons
HMI	hmi

Table 1: Tasks and chains in platoon application.

It is worth noting that all tasks are time-driven and periodic.⁷ Also notice that the same process might be invoked twice in a chain, e.g., `veh_iols` is invoked twice in the longitudinal chain, first by a message from `atmio16`, then by a trigger for `long_output`.

5 Analysis

Timing 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. For the platoon application, we set the deadline of a task to be equal to its period. The fact that the deadline of, say, the lateral input task, is 2ms ensures that no hardware interrupt will be missed and that control outputs will be computed using “fresh” inputs.⁸

It is not at all obvious that the software architecture meets its deadline requirements. In this section, we perform a formal analysis to check that this is the case. We use results from real-time scheduling theory, in particular, fixed-priority scheduling theory (e.g., see [8, 7, 5, 13, 1, 2, 15]) and the so-called *HKL* model and analysis [3, 4]. First, we cast the platoon application into the formal model. Then, we estimate the execution times and other latencies involved in the system, and compute the total CPU utilization. This is found to be about 74%, that is, less than 1, which is a necessary (but not sufficient) condition for schedulability. Finally, we apply HKL analysis to ensure that the deadlines are indeed met.

⁷A wireless TDMA (*time division multiple access*) protocol ensures that vehicles in a platoon communicate without collisions, so that packets arrive deterministically every 20ms. There are no retransmissions. If a packet is corrupted it is discarded. If three consecutive packets are missed, the control goes into a fault mode.

⁸In 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.

5.1 A formal model for the platoon application

We describe our application as a set of periodic tasks, each consisting of a sequence of sub-tasks with varying priorities. First we present our model, which is a special case of the one in [3, 4].

We have a set of tasks τ_1, \dots, τ_n . Each τ_i has a period $T_i > 0$ and a deadline $D_i = T_i$. Each τ_i is associated with a sequence $\tau_{i,1}, \dots, \tau_{i,m(i)}$ of *sub-tasks*. Each sub-task $\tau_{i,j}$ has an execution time $C_{i,j}$ and a priority $P_{i,j}$. We define C_i to be $\sum_{j=1}^{m(i)} C_{i,j}$, that is, the total execution time of τ_i . It is assumed that $C_i \leq T_i$, for all i .⁹

Each τ_i becomes *ready* for execution at times $t_i^k = \phi_i + kT_i$, $k = 0, 1, 2, \dots$, where ϕ_i is the initial *phase* of the task. At time t_i^k , the sub-task $\tau_{i,1}$ becomes *active* and remains active until it has executed for a total of $C_{i,1}$ time units, at which time it *finishes*. At that time, $\tau_{i,2}$ becomes active, and so on. When sub-task $\tau_{i,m(i)}$ finishes, the k -th *job* of τ_i finishes. At any time t the CPU executes one of the active sub-tasks that have the highest priority. If there are more than one such sub-tasks, ties are broken arbitrarily. We say that τ_i meets its deadline if for all $k = 0, 1, 2, \dots$, the k -th job of τ_i finishes at time $t_i^k + D_i$ at the latest. We say that the set of tasks is *schedulable* if for all possible initial phases and all possible ways to break ties, all tasks meet their deadlines. The problem is, given a set of tasks, to check whether it is schedulable.

We now show how to cast the platoon application in the above model. In short, a task will be a logical task and its sub-tasks will be the processes involved in its execution chain. As an example, consider τ_1 (the lateral input task). We have $D_1 = T_1 = 2$ (assumed to be in milliseconds). The sub-tasks of τ_1 are extracted from the lateral input process chain. Recall that the latter involves the execution of `atmioe`, `veh_lat` and `hst`, in that order. However, we cannot consider only three sub-tasks, because `veh_lat` and `hst` interact with the database, which is implemented as a process itself. Indeed, each interaction of a process A with the database D includes A sending a request-message to D , D processing the request and replying to A with a response-message, and A receiving this message and continuing execution. Thus, such an interaction can be modeled as a sequence of three sub-tasks: $A_s, D_{r,s}, A_r$, where A_s models A sending the request, $D_{r,s}$ models D receiving the request, processing and replying, and A_r models A receiving the response. Now, if A executed two requests (say, a read and an update), we get a sequence of five sub-tasks: $A_s, D_{r,s}, A_{r,s}, D_{r,s}, A_r$, since we can combine receiving the first response and sending the second request in a single sub-task $A_{r,s}$.

Apart from interactions of data I/O processes with the database, we must also model the interaction of these processes with the device drivers. For example, when `veh_lat` reads data from `atmioe`, this comes down to `veh_lat` sending a request to `atmioe` and the latter replying back, which can be modeled by a sequence $A_s, B_{r,s}, A_r$.

Coming back to our lateral input task example, we combine the above remarks as follows. Given that `veh_lat` reads from `atmioe` and requests one database update, and `hst`

⁹The model of [4] is more general, in that each sub-task $\tau_{i,j}$ has its own deadline $D_{i,j}$, such that $D_{i,1} \leq D_{i,2} \leq \dots \leq D_{i,m(i)} = D_i$, and also D_i can be smaller or greater than T_i . In our case, $D_{i,j} = D_i = T_i$.

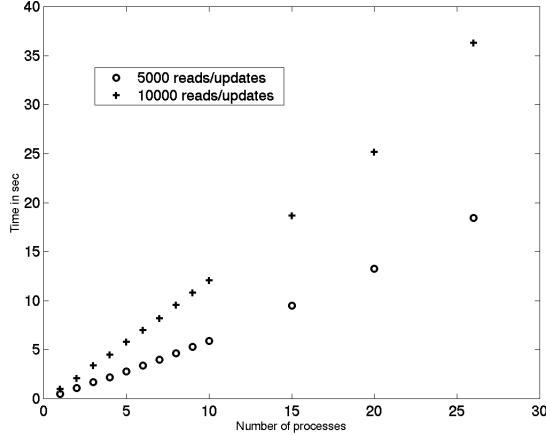


Figure 4: Performance of the Publish/Subscribe library on a 166 MHz QNX PC.

- h : latency to handle a hardware interrupt.
- p : latency to send a synchronous message between processes.
- t : latency to send an asynchronous trigger from the database to a client.
- c : context switching delay (includes scheduling).

We use the following estimates for the above latencies, based on information from [11]: $h = 10\mu s$, $p = 40\mu s$, $t = 40\mu s$, $c = 30\mu s$. Notice that r and w already include context switching overhead, so this is not added for these operations.

We ignore floating point computation latencies, since they are extremely small. In experiments we conducted, 20 million floating point operations took approximately 0.12 seconds on the 166 MHz Pentium machine. This averages to approximately 1 microsecond for 166 floating point operations. A typical control computation involves fewer such operations.

We can now estimate the execution time $C_{i,j}$ for each sub-task $\tau_{i,j}$. For example, consider the steering input task τ_4 , consisting of six sub-tasks. Sub-task $\tau_{4,1}$ models the interrupt handler of `cani` which sends a message to `canread`, thus, $C_{4,1} = h + p + c$ (we associate the context switch latency with the higher-priority process). Sub-tasks $\tau_{4,2}$ and $\tau_{4,3}$ model `canread` performing a hardware read operation, that is, the request message from `canread` to `cani` and the response message from `cani` to `canread`, thus, $C_{4,2} = C_{4,3} = p + c$. Sub-tasks $\tau_{4,4}$, $\tau_{4,5}$ and $\tau_{4,6}$ model `canread` performing a `clt_update`. Since we are interested in worst-case analysis, we estimate that the bulk of the latency in this operation is associated with $\tau_{4,5}$, i.e., the highest-priority database process. Therefore, we set $C_{4,4} = C_{4,6} \approx 0$ and $C_{4,5} = w$.

5.3 Total CPU utilization

Based on the above estimates, we can compute the total execution time C_i of each task τ_i . For example, τ_1 involves one hardware interrupt handler, a message from `atmioe` to `veh_lat`, a context switch, a hardware read operation, a database update, a trigger, another context switch, four database reads and 2 updates. In total, we have $C_1 = h + p + c + p + c + w + t + c + 4r + 2w = 740\mu\text{s}$. Since τ_1 is invoked every 2000 μs , the partial CPU utilization induced by τ_1 is $U_1 = \frac{C_1}{T_1} = \frac{740}{2000} = 37\%$. Similarly, we can compute C_i and U_i for all i . The results are shown in Table 2.

Then, we can compute the total CPU utilization:

$$U = \sum_{i=1}^{11} \frac{C_i}{T_i} = \sum_{i=1}^{11} U_i = 73.67\%$$

We see that $U < 1$, which is a necessary condition for tasks to be schedulable.

5.4 Schedulability analysis

A special case of our model, where each task consists of a single sub-task (i.e., $m(i) = 1$ for all i), is used by basic rate-monotonic analysis. In this simpler model, there are roughly two ways to check whether a set of tasks is schedulable.

- By computing the total CPU utilization U defined as above, and showing that $U \leq n(2^{1/n} - 1)$, where n is the number of tasks. This is only a sufficient condition for schedulability and assumes that priorities are assigned to the tasks according to the *rate-monotonic* policy, that is, priorities are inversely proportional to task periods. In fact this is an *optimal fixed-priority policy*, in the sense that if the tasks are schedulable with any other fixed-priority policy then they are also schedulable with the rate-monotonic policy [8].
- By performing the so-called *completion-time test* [7, 5, 6]. This is an exact test, that is, tasks are schedulable if and only if they pass the test.

We briefly describe the completion-time test for the simple model and then show how it is extended for the model of tasks with more than one sub-tasks. We try to make our description self-contained, although we obviously cannot give a thorough presentation. The reader is referred to [5, 3, 4, 6].

Completion-time test for periodic tasks with uniform priority. Given a task i , let $H(i) = \{j | P_j \geq P_i\}$ be the set of indices of tasks of priority higher than or equal to i . Define $W_i(t) = \sum_{j \in H(i)} C_j \lceil \frac{t}{T_j} \rceil$. $W_i(t)$ represents the cumulative demand of all tasks of priority at least as i , in the time interval $[0, t]$. Also define the series $S_0^i = \sum_{j \in H(i)} C_j$, and $S_{k+1}^i = W_i(S_k^i)$. This series is monotonically increasing, so that eventually we will find a $k(i)$ such that either $S_{k(i)}^i = S_{k(i)+1}^i \leq T_i$, or $S_{k(i)}^i > T_i$. The completion-time theorem says that τ_i always meets its deadline iff $S_{k(i)}^i = S_{k(i)+1}^i \leq T_i$. In order to check whether a set of tasks is schedulable, we have to apply the completion-time test to each task individually.

HKL analysis for periodic tasks with sub-tasks of varying priority. We now return to the more general model which is applicable to our case. We first need some definitions.

The *canonical form* of a task τ_i is a new task τ'_i which consists of a sequence of sub-tasks $\tau'_{i,1}, \dots, \tau'_{i,m(i)}$, such that their priorities are strictly increasing, and $\tau'_{i,j}$ has been obtained by “compressing” two or more consecutive sub-tasks of τ_i with equal or decreasing priorities. The priority of $\tau'_{i,j}$ is the smallest among the priorities of the merged sub-sequence. For example, the canonical form of τ_1 in the platoon application (see Table 2) is a task with a single sub-task of priority 18. The canonical form of τ_2 is a task with two sub-tasks with priorities 19, 24.

The interest behind the canonical form of a task τ_i is as follows. Suppose sub-tasks $\tau_{i,j}$ and $\tau_{i,j+1}$ have priorities 10 and 9, respectively. The fact that $\tau_{i,j}$ has a higher priority than $\tau_{i,j+1}$ does not “help” task τ_i with respect to its total worst-case completion time (WCCT). Indeed, $\tau_{i,j+1}$ can be preempted or blocked by other sub-tasks of priority 9 or higher, and this will delay the entire task τ_i . Therefore, the WCCT of τ_i would be the same if the priority of $\tau_{i,j}$ was lowered to 9. This result (formally proven in [3, 4]) allows one, instead of checking whether τ_i is schedulable, to check whether its canonical form is schedulable.

Now define $Pmin_i$ to be $\min\{P_{i,j} \mid j = 1, \dots, m(i)\}$, that is, the minimum priority of all sub-tasks of τ_i . The next step is to classify all tasks τ_j , $j \neq i$, according to their *relative priority levels* with respect to $Pmin_i$. For example, τ_4 is classified with respect to τ_1 as follows. We have $Pmin_1 = 18$. The sequence of priorities of sub-tasks of τ_4 is 19, 10, 19, 10, 25, 10, or, relative to 18, H, L, H, L, H, L , where H stands for “higher or equal” and L for “strictly lower”. We say that τ_4 is a *type 2, or $(H^+, L^+)^+$* , task with respect to τ_1 . Performing the same classification for all tasks τ_2, \dots, τ_{11} with respect to τ_1 , we find three *types* of tasks, as identified in [3, 4]:

- Type 1, or H^+ , tasks. τ_2, τ_3, τ_9 belong in this class. For example, the sequence of priorities of τ_2 relative to 18 is H, H, H, H, H . These are tasks all sub-tasks of which have priority at least 18, and therefore can preempt τ_1 multiple times.
- Type 2, or $(H^+ L^+)^+$, tasks. $\tau_4, \tau_5, \tau_6, \tau_7, \tau_8$ belong in this class. Each of these tasks can preempt or block τ_1 at most once, since after it executes a sub-sequence of sub-tasks (or *segment*) of relatively higher priority H^+ , it must execute a strictly lower priority segment L^+ and will therefore be preempted by τ_1 .
- Type 4, or $(L^+ H^+)^+ L^+$, tasks. τ_{10}, τ_{11} belong in this class. For example, the sequence of priorities of τ_{10} relative to 18 is L, H, L, H, L, H, L . At most one of these tasks can preempt or block τ_1 , and at most once.

Now define $H_1(i), H_2(i), H_4(i)$ to be the indices of all tasks of type 1, 2, 4, respectively, relatively to τ_i . For example, $H_1(1) = \{2, 3, 9\}$, $H_2(1) = \{4, 5, 6, 7, 8\}$ and $H_4(1) = \{10, 11\}$.

For each $j \in H_2(i) \cup H_4(i)$, let $B(i, j)$ be the maximum total execution time of a segment of τ_j among all H^+ segments of τ_j with respect to τ_i . For example, there are three H^+ segments of τ_4 with respect to τ_1 , each consisting of one sub-task, namely, $\tau_{4,1}, \tau_{4,3}, \tau_{4,5}$.

Then, $B(1, 4) = \max\{C_{4,1}, C_{4,3}, C_{4,5}\} = \max\{h + p + c, p + c, w\} = w$. Now, define:

$$B_2(i) = \sum_{j \in H_2(i)} B(i, j), \quad B_4(i) = \max\{B(i, j) \mid j \in H_4(i)\}, \quad B(i) = B_2(i) + B_4(i).$$

$B(i)$ is called the *blocking time* of τ_i . For example, for τ_1 , we have:

$$\begin{aligned} B_2(1) &= B(1, 4) + B(1, 5) + B(1, 6) + B(1, 7) + B(1, 8) \\ &= 3 \cdot B(1, 4) + B(1, 7) + B(1, 8) \\ &= 3w + \max\{h + p + c, p + c, r, w + t + c\} + \max\{h + p + c + 3w + 2t + c, r + c\} \\ &= 3w + (w + t + c) + (h + p + c + 3w + 2t + c) \\ &= 7w + 3t + 3c + h + p = 1100 \\ B_4(1) &= \max\{B(1, 10), B(1, 11)\} = B(1, 10) = w = 120 \\ B(1) &= B_2(1) + B_4(1) = 1220 \end{aligned}$$

We are now ready to present the completion-time test in the general model [6]. Given a task τ_i , re-define $W_i(t) = \sum_{j \in H_1(i)} C_j \lceil \frac{t}{T_j} \rceil$. Also re-define the series $S_0^i = B(i) + \sum_{j \in H_1(i)} C_j$, and $S_{k+1}^i = B(i) + W_i(S_k^i)$. As in the simple model case, τ_i is schedulable iff there is a $k(i)$ such that $S_{k(i)}^i = S_{k(i)+1}^i \leq T_i$.

We now apply the completion-time test to τ_1 :

$$\begin{aligned} S_0^1 &= B(1) + \sum_{j \in H_1(1)} C_j = B(1) + C_2 + C_3 = 1720 \\ S_1^1 &= B(1) + W_1(S_0^1) = B(1) + C_2 \lceil \frac{1720}{4000} \rceil + C_3 \lceil \frac{1720}{8000} \rceil = 1720 \end{aligned}$$

We find that $S_0^1 = S_1^1 \leq T_1$, therefore, τ_1 meets its deadline.

6 Conclusions

We have described the embedded software architecture of a real automated vehicle control system. We have argued that the properties of the architecture make it particularly attractive for many other real-time control applications as well. We have also shown that the architecture is amenable to formal analysis, by casting it into a formal model, estimating the execution times of its various components, and performing a sophisticated schedulability analysis.

The idea of a publish/subscribe inter-process communication scheme is certainly not new. However, we believe that the scheme we present here contains the “right mix” of the important ingredients for real-time control applications. For instance, it is not a “pure” publish/subscribe scheme as the one proposed in [12], where there is only the notion of messages sent from publishers to subscribers. Instead, we maintain the notion of a shared memory, where readers can access variables at any time, while at the same time they have the possibility to subscribe by setting triggers.

The analysis techniques are not new either. However, we believe that it is useful to report their usage on a real-world case-study which put their applicability to the test. Performing the analysis manually has been both tedious and error-prone, the reason being the non-trivial size of our application. Therefore, it is important that such analysis techniques be automated.¹¹ It is also important to provide meaningful feedback to the user (e.g., in the form of a counter-example) in case some tasks miss their deadlines, as well as directives of which parameters to modify (e.g., priorities) in order for the tasks to become schedulable.

Our estimation of execution times has been admittedly crude. However, our purpose was not to estimate execution times accurately but to show the applicability of the analysis in our architecture. Estimating WCETs accurately and automatically is certainly one important problem as well as an active area of research (e.g., see [9, 16]).

Acknowledgments. I would like to thank Paul Kretz from PATH and Raj Rajkumar from CMU.

References

- [1] N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell, and A.J. Wellings. Fixed Priority Pre-Emptive Scheduling: An Historical Perspective. *Real Time Systems* 8, 2-3, March-May 1995.
- [2] A. Burns, K.W. Tindell and A.J. Wellings. Effective Analysis for Engineering Real-Time Fixed Priority Schedulers. *IEEE Trans. Software Engineering*, Vol 21, No 5, pp475-480, 1995.
- [3] M. Harbour, M.H. Klein, and J. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. *Proceedings of IEEE Real-Time Systems Symposium*, pp116-128, December 1991.
- [4] M.G. Harbour, M.H. Klein, and J.P. Lehoczky. Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems. *IEEE Transactions on Software Engineering*, vol. 20, no. 1, pp. 13-28, January 1994.
- [5] 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.
- [6] M.H. Klein, J. Lehoczky and R. Rajkumar. Rate Monotonic Analysis for Real-Time Industrial Computing. *IEEE Computer*, January 1994.
- [7] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. *IEEE Real-Time Systems Symposium*, pages 166-171, December 1989.

¹¹In fact, (generalized) rate-monotonic analysis is currently becoming available in commercial tools (e.g., see [17]).

- [8] C.L. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1) pp 46-61, January 1973.
- [9] P. Puschner and A. Burns. A review of WCET analysis. *Real Time Systems*, Special Issue on Worst-Case Execution-Time Analysis, Volume 18, Issue 2/3, May 2000.
- [10] http://www.qnx.com/literature/qnx_sysarch/index.html.
- [11] <http://www.qnx.com/literature/whitepapers/archoverview.html>.
- [12] R. Rajkumar, M. Gagliardi and L. Sha. "The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation". *IEEE Real-time Technology and Applications Symposium*, June 1995.
- [13] L. Sha, R. Rajkumar and S.S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *IEEE Proc.*, January 1994.
- [14] L. Sha, R. Rajkumar and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers*, September 1990.
- [15] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms. Kluwer Academic Publishers, Boston, 1998.
- [16] Swedish WCET Network home page: www.docs.uu.se/artes/wcet/.
- [17] White paper of Tri-Pacific Software Inc, by B. Watson: Using PERTS and PERTS*Sim to Analyze End-to-End Completion Times. Available at: www.tripac.com/html/whitepapers/end2end.pdf
- [18] 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.¹² 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.

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.

¹²Triggers 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.

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/sec²), 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/sec²), 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/sec²), 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.