07AE-290

HIS/VectorCAN Driver API on top of a Time-Triggered Communication Protocol

R. Obermaisser

Vienna University of Technology, Austria

D. Riezler

TTTech, Austria

ABSTRACT

The HIS/VectorCAN driver provides an Application Programming Interface (API) that is used in many present day cars and makes CAN-based applications independent from the implementation details of specific communication controllers. This paper describes a solution for establishing this API in a time-triggered computer system. We construct integrated node computers, which provide both an execution environment for time-triggered applications and a CAN execution environment. The CAN execution environment offers to the application software the API according to the HIS/VectorCAN driver specification. Thereby, we allow the reuse of existing software, which has been developed for the API of the HIS/VectorCAN driver, as part of future time-triggered in-vehicle electronic systems. For this purpose, this paper introduces middleware services that operate between a time-triggered operating system and the CAN-based applications. In a first step, the middleware establishes an event channel on top of the time-triggered communication protocol in order to support CAN transmission requests at a priori unknown points in time. By using this event channel, the middleware then realizes the services comprising the API of the HIS/VectorCAN driver, including send and receive operations, message filtering, and callbacks. In a prototype setup with a TTP cluster, we show how the API can be used by periodic and sporadic CAN application tasks.

INTRODUCTION

At present, two different paradigms are prevalent in the design of real-time architectures. In event-triggered architectures the system activities, such as sending a message or starting computational activities, are triggered by the occurrence of events in the environment or the computer system. In time-triggered architectures (e.g., Time-Triggered Architecture (TTA) [1], FlexRay [2]), activities are triggered by the progression of global time. The major contrast between event-triggered and time-triggered approaches lies in the location of control. Time-triggered systems exhibit autonomous control and interact with the environment according to an internal predefined schedule, whereas event-triggered systems are under the control of the environment and respond to stimuli as they occur.

The time-triggered approach is generally preferred for safety-critical systems [3, 4]. For example, in the automotive industry a time-triggered architecture will provide the ability to handle the communication needs of by-wire cars [5]. In addition to hard real-time performance, time-triggered architectures help in managing the complexity of fault-tolerance and corresponding formal dependability models, as required for the establishment of ultra-high reliability (failure rates in the order of 10^{-9} failures/hour).

Today, the event-triggered CAN (Controller Area Network) protocol [6] is by far the most widely used automotive protocol with a 100% market penetration. Major advantages of CAN include its high flexibility (e.g., strong migration transparency, no need to change a communication schedule when adding messages), resource efficiency through the sharing of bandwidth between Electronic Control Units (ECUs), low cost of CAN hardware, and high availability of CAN-based tools and engineers with CAN know-how.

Despite the use of time-triggered architectures in future by-wire cars, CAN is likely to remain as a communication protocol for non safety-critical application subsystems due to the higher flexibility and average performance. Even for safety-related subsystems, CAN-based legacy applications will not be replaced instantly.

In this context, event channels for the dissemination of CAN messages in a time-triggered architecture are a solution for layering event-triggered CAN communication on top of a time-triggered communication protocol [7].

An event channel is a generic architectural service that accepts as its input transmission requests from a CAN-based application at arbitrary instants, thus improving the flexibility of a time-triggered architecture. In addition to periodic time-triggered message exchanges, this service permits to communicate without having to statically predefine the instants of all message exchanges. Liberating non safety-critical application subsystems from the need to specify a static communication schedule at design time simplifies future modifications of the application software. If additional or different messages are sent via an event channel, the underlying time-triggered communication schedule can remain unchanged. Furthermore, multiple messages sent by an ECU can share bandwidth on the network in order to improve resource efficiency.

Event channels enable the integration of applications for which a CAN communication service is preferable, as well as the integration of CAN-based legacy applications into a time-triggered computing platform. However, for the reuse of CAN-based legacy applications with event channels, it is also important to establish the higher protocol layers and the APIs legacy applications have been developed for. The separation of the hardware of an ECU from its embedded software has been identified as a key requirement for the reuse of embedded software [8]. For this purpose generic APIs have been specified, which abstract from any particular CAN controller chip. Communication drivers, such as the HIS/VectorCAN driver [9], CANopen [10], SDS [11], and DeviceNet [12] provide to application software a generic API that abstracts from low-level interfaces (e.g., register set of a particular CAN controller, message buffer configurations like Full-CAN and Basic-CAN).

Based on event channels, this paper presents a solution for establishing the services of the HIS/VectorCAN driver [9] in a time-triggered computer system. We have chosen the services of the HIS/VectorCAN driver [9] over other higher protocols and APIs, because it is used and supported in the automotive domain by several vehicle manufacturers[1]. Nevertheless, the introduced model of an integrated ECU and the middleware implementation has been designed in a flexible manner to be able to accommodate different higher protocols. A modular design with middleware layers separated by well-specified interfaces ensures that only a single layer – the front-end – needs to be replaced for establishing different APIs.

The availability of the API of the HIS/VectorCAN driver within the ECUs of a time-triggered system provides a basis for designers who intend to reuse CAN-based applications despite the migration to a time-triggered platform. In addition to the ability to retain investments in existing software, such a migration permits the replacement of physical CAN networks through event channels. As overlay networks on the time-triggered communication service, the resulting virtual CAN networks enable significant cost and reliability benefits through the reduction of connectors and wiring. Wiring and connectors are currently a prevalent source of faults in the automotive area [14].

[1] www.automotive-his.de (OEM Initiative Software)

CAN-BASED HIGHER-LAYER PROTOCOLS AND APIS

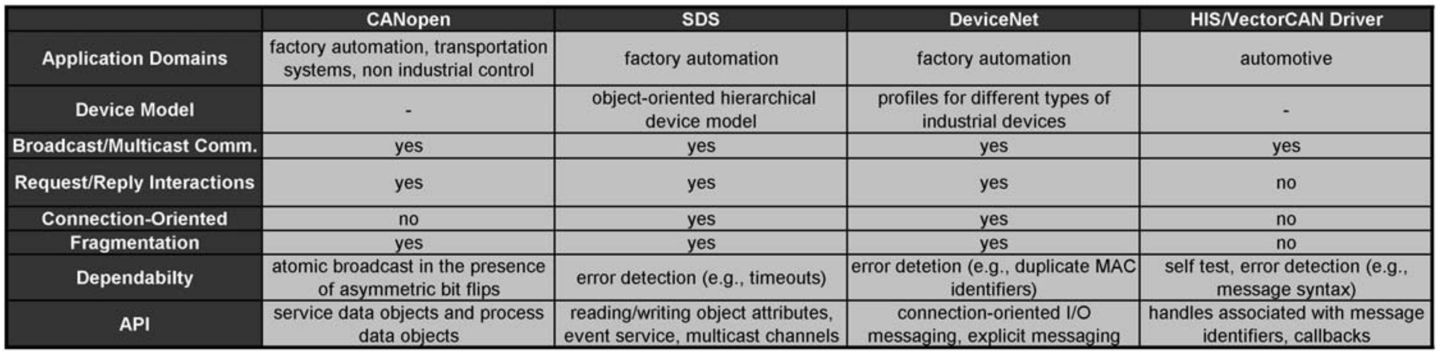
The CAN standard [15] specifies the physical layer and the data link layer. The higher protocol layers described in this section provide additional functionality (e.g., message fragmentation) and provide uniform APIs. By abstracting from the details of any specific CAN controller, a uniform API facilitates the reuse of application software on top of different hardware platforms. In particular, we can support the reuse of CAN-based application software in a time-triggered system by establishing the respective API. Hence, the fact that there is no physical CAN network, but a virtual CAN network on top of a time-triggered network is hidden from the application software.

Figure 1 contains a summary of the key properties of prevalent CAN-based higher protocols. We list the respective application domain and give information whether a device model for exchanged messages is defined. In addition, for each protocol the API provided to the application software is summarized. At the operational level, we look at the supported types of interactions between nodes (i.e., multicast, request/reply, connection-oriented) and the ability for message fragmentation. Those protocols that support fragmentation, permit the exchange of application data with more than 8 bytes through the concatenation of multiple CAN frames. Finally, fault-tolerance mechanisms that aim at improving the dependability of a CAN-based system are identified in the different protocols.

CANOPEN

CANopen [10] aims at providing a uniform application layer for distributed embedded systems. Initially, the design of CANopen has occurred within the Esprit project. Today, the international users' and manufacturers' group of CAN in Automation (CiA) continues the development of CANopen. The major application fields include low- and mid-volume embedded systems, such as off-road vehicles, trains, factory and building automation, and industrial and non-industrial control.

CANopen introduces the concept of device profiles, which define both mandatory functionality ensuring basic interoperability and additional manufacturer-specific functions. Each device profile contains an object dictionary, which captures both application data and configuration parameters (e.g., manufacturer name, communication parameters).



	CANopen	SDS	DeviceNet	HIS/VectorCAN Driver
Application Domains	factory automation, transportation systems, non industrial control	factory automation	factory automation	automotive
Device Model	-	object-oriented hierarchical device model	profiles for different types of industrial devices	-
Broadcast/Multicast Comm.	yes	yes	yes	yes
Request/Reply Interactions	yes	yes	yes	no
Connection-Oriented	no	yes	yes	no
Fragmentation	yes	yes	yes	no
Dependabilty	atomic broadcast in the presence of asymmetric bit flips	error detection (e.g., timeouts)	error detetion (e.g., duplicate MAC identifiers)	self test, error detection (e.g., message syntax)
API	service data objects and process data objects	reading/writing object attributes, event service, multicast channels	connection-oriented I/O messaging, explicit messaging	handles associated with message identifiers, callbacks

Figure 1: CAN-based higher protocols and APIs

Process Data Objects (PDOs) and Service Data Objects (SDOs) access the application data of the object dictionary. SDOs employ a client-server mechanism allowing a server to read or write entries in the client's object dictionary. Messages contain up to 7 data bytes and additional control fields that are used by an atomic broadcast protocol that prevents inconsistent message disseminations. As identified in [16], CAN [6] can exhibit duplicate message failures and duplicate message omissions in case of asymmetric bit flips and crash failures.

CANopen also performs fragmentation of data that is too large to fit into a single CAN message. In case no fragmentation is required, PDOs enable the transfer of up to 8 data bytes within a single CAN frame. A PDO exhibits a fixed identifier and is transmitted by a single node only.

CANopen supports different types of triggers for initiating communication activities. In event- or timer-driven communication, the device profile defines an event or a local timer that triggers PDO communication. In a remotely requested communication, a device may request a message transmission from another device. A synchronous interaction comprises the transmission of a so-called sync frame by a master and the slave response after every *n*th frame or on a sync frame following an application-specific event.

SDS

Smart Distributed System (SDS) [11] has been developed by Honeywell's Micro Switch Division as an advanced bus system for intelligent sensors and actuators. Application fields range from packaging/food processing equipment over automotive plant floors, material handling and conveyor systems to automated storage retrieval systems.

Each SDS node contains one or more logical devices and provides connection to the CAN bus. A logical device is a bus addressable entity that is further structured into up to 32 SDS objects. The two most important attributes of an SDS object are its object type and network data descriptor. The object type refers to the location of the object in a hierarchy of SDS objects, which describes the object behavior with respect to actions and events. The network data descriptor defines the size, granularity, and data type of the network data which is managed by SDS interfaces and is accessible to other SDS devices. Communication in SDS is established through four primitive functions: request, indication, response and confirm. After a request at the application layer, an Application Layer Protocol Data Unit (APDU) is transmitted and results in an indication of the received message at another device. The receiving device can invoke a response function, which is subsequently acknowledged at the initiating device trough the confirm primitive. Based on these four primitives, SDS provides services for reading/writing object attributes, indicating event occurrences, and communicating via multicast- and peer-to-peer channels.

DEVICENET

DeviceNet [17] is a CAN-based higher protocol developed by the Open DeviceNet Vendor Association (ODVA) for industrial automation applications. DeviceNet is an implementation of the Control and Information Protocol (CIP) on top of CAN and interconnects CAN nodes representing industrial devices. DeviceNet introduces an object model for industrial devices, such as sensors, actuators, and controllers, and employs a producer-consumer messaging model.

In addition to protocol-specific objects, each node possesses application objects that contain the input or output data of the industrial device at this node. In order to provide vendor-independence, electronic data sheets and predefined device types are used to specify mandatory attributes of application objects and supported interaction types (e.g., polling, cyclic).

Each DeviceNet node is either a master or a slave. Master nodes establish connections, acquire input data (e.g., sensory information) from slaves, and send output data (e.g., set point for an actuator) to slaves. While a master can interact with multiple slaves, a slave node belongs to at most one master.

DeviceNet is a connection-oriented protocol. During the creation of a connection, a master establishes the ownership of a slave node. The master defines the type of the connection (e.g., event-triggered interactions called change-of-state messaging), sets the message rate, and configures the produced and consumed message paths.

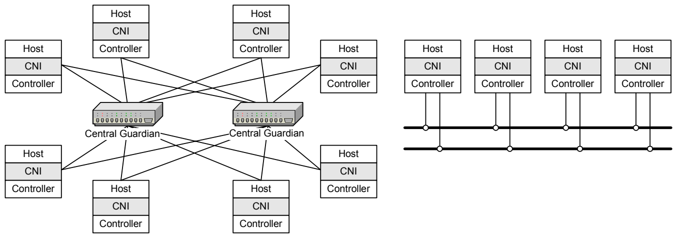


Figure 2: TTP network with star and bus topology

HIS/VECTORCAN DRIVER

The HIS/VectorCAN driver [9] has been developed by Vector Informatik GmbH and standardized by the OEM Initiative Software, an interest group in standardizing software founded by leading german car manufacturers. The aim of the HIS/VectorCAN driver is to provide uniform services to transmit and receive messages over the CAN bus as well as uniform services to control hardware-specific features (e.g., power management) by the application. The HIS/VectorCAN driver has been developed for the use in automotive systems, but it also meets the requirements of other application fields.

To offer a uniform application interface for different CAN controllers and to hide different message buffer configurations, the HIS/VectorCAN driver offers a handle-based application interface. A handle is either a transmit object or a receive object, which incorporates a CAN identifier, a data length code, a pointer to the CAN frame data, and pointers to callbacks. An example for a callback is a pretransmit routine, which enables the application to update the CAN frame data prior to the transmission on the bus. Another callback is the confirmation routine, which informs the application about the successful transmission of a message after the acknowledgment has been received from the communication controller. In analogy, the reception of a CAN message which has passed the hardware and software acceptance filtering is reported to the application by several configurable callback functions.

INTEGRATED ARCHITECTURE FOR TIME-TRIGGERED AND CAN-BASED APPLICATIONS

The TTA provides a computing infrastructure for the design and implementation of dependable distributed real-time systems [1]. The basic building block of the TTA is a node computer (denoted node for short), which is a self-contained composite hardware/software subsystem [18]. A cluster is a set of nodes that are interconnected by a network running the Time-Triggered Protocol (TTP) [19], which provides clock synchronization, predictable exchange of state messages, and a membership service. For the implementation of the network, the TTA distinguishes between two physical interconnection topologies (see Figure 2), both employing two redundant communication channels.

In [7], a solution for the layering of a CAN communication service on top of TTP has been proposed. By providing such an emulated CAN communication service

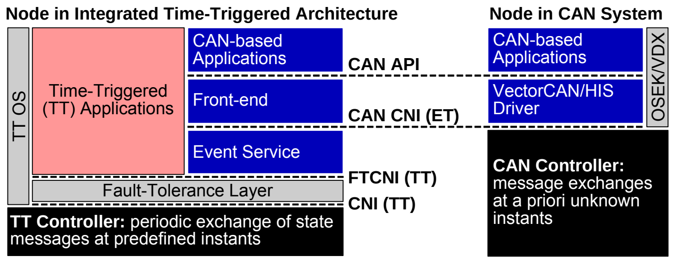

Node in Integrated Time-Triggered Architecture
Node in CAN System
TT OS
Time-Triggered (TT) Applications
CAN-based Applications
Front-end
Event Service
Fault-Tolerance Layer
TT Controller: periodic exchange of state messages at predefined instants
CAN API
CAN CNI (ET)
FTCNI (TT)
CNI (TT)
CAN-based Applications
VectorCAN/HIS Driver
OSEK/VDX
CAN Controller: message exchanges at a priori unknown instants

Figure 3: HIS/CAN API in both a node of the integrated time-triggered architecture (left) and in a conventional CAN node (right)

in the TTA, we establish an integrated architecture for event-triggered and time-triggered control [20]. This architecture supports the coexistence of applications that require an event-triggered communication service (i.e., CAN) with applications that require a time-triggered communication service. Safety-critical functionality is always realized as a time-triggered application, while the CAN communication service offers to non safety-critical applications an execution environment with high flexibility and average performance. The underlying design decision of using the time-triggered control paradigm for safety-critical functionality conforms to widely accepted requirements for the infrastructure of safety-critical real-time systems [5, 3].

Figure 3 depicts both a node in the integrated time-triggered architecture with support for event-triggered CAN communication, as well as a node in a conventional CAN-based system. In order to interact with the CAN communication system, both nodes provide to CAN-based applications a generic CAN API. In the conventional CAN node, the VectorCAN/HIS driver provides this API, which supports message transmissions, message receptions, message filtering, and callbacks via message handles. In the following, we call this API, which is established by the VectorCAN/HIS driver, the HIS/CAN API for short.

The node in the integrated time-triggered architecture uses middleware services (fault-tolerance layer, event service, and front-end) for the step-wise transformation of the Communication Network Interface (CNI) of the underlying time-triggered network onto the HIS/CAN API. The middleware services realize the communication infrastructure for the CAN subsystem as a Virtual CAN Network (VCN), which is an overlay network on top of the time-triggered physical network. For the realization of the VCN, we perform a temporal subdivision of the communication resources provided by the Time Division Multiple Access (TDMA) scheme of the time-triggered physical network. TDMA statically divides the channel capacity into a number of slots and assigns to each node a unique slot that periodically reoccurs at a priori specified global points in time. We subdivide each node's slot into two subslots, namely a slot for time-triggered communication and a slot for the event-triggered dissemination of CAN messages (see Figure 4).

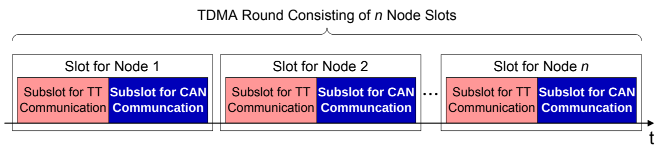

TDMA Round Consisting of n Node Slots
Slot for Node 1
Slot for Node 2
Slot for Node n
Subslot for TT Communication
Subslot for CAN Communcation
t

Figure 4: Temporal subdivision of TDMA slots

FAULT-TOLERANCE LAYER

The fault-tolerance layer [21] hides replication by voting on the data from redundant communication channels and redundant nodes. It maps the CNI of the time-triggered communication controller into a memory element called the Fault-Tolerant CNI (FTCNI), which contains the voted state messages. The fault-tolerance layer supports fault-tolerance through active redundancy transparently to the application software and higher protocol layers, such as the event service and front-end. The fault-tolerance layer fuses the redundant messages received via the two redundant TTP channels and tolerates a single detectably faulty channel (i.e., a message omission failures or a syntactically incorrect message as indicated by a CRC check). The fault-tolerance layer also supports N-modular redundancy (NMR) [22] for masking node failures.

EVENT SERVICE

The event service is a middleware service that exploits the subslot for CAN communication in order to establish an event channel for the on-demand transmission of CAN messages. The event service maps an event-triggered communication service (denoted as CAN CNI in Figure 3) to the sparse time base [23] of a time-triggered communication service. Although message transmission requests can occur at arbitrary instants, the dissemination of the messages on the underlying time-triggered network is always performed at the predefined global points in time of the CAN subslots. Outgoing messages are buffered in message queues until the respective node's subslot for CAN communication occurs in the TDMA scheme. Also, the queuing of messages permits bursts during which the bandwidth consumption of outgoing messages exceeds the bandwidth that is available via CAN subslots. In every node, the event service performs a fragmentation of outgoing messages into packets that can be placed in the node's CAN subslot. In addition, the event service reassembles messages out of received packets.

FRONT-END

This middleware service establishes a CAN-based higher protocol (e.g., one of the protocols described in the previous section) and provides an API by which application software can access the VCN. The front-end provides the operations for the transmission and reception of messages and the reconfiguration of the CAN communication system (e.g., setting of message filters). The front-end also implements interrupt mechanisms via callbacks that enable the application to react to significant events, such as the reception of a message or the occurrence of an error. In addition, the front-end also provides functionality that goes beyond the services defined in the CAN specification [6], e.g., message fragmentation.

IMPLEMENTATION

This section presents the implementation of the HIS/CAN API in the Time-Triggered Architecture (TTA) [1]. We start with an overview of the platform employed for the implementation, namely a cluster with four nodes interconnected by a TTP network. Middleware services map the time-triggered state message interface of the TTP network to the HIS/CAN API required by the CAN-based application software.

HARDWARE AND SOFTWARE PLATFORM

The platform used for the implementation comprises a TTP development cluster with four TTP power nodes [24]. A TTP power node contains a Freescale MPC555 PowerPC host computer with floating point unit running at 40 MHz. Each node is equipped with 1 MByte of RAM, 4 MBytes of flash memory and provides physical interfaces to two 5 Mbps TTP channels and to fieldbus networks (e.g., LIN [25]). For accessing the TTP channels each node is equipped with a C2 communication controller (AS8202) [26].

The communication controller performs periodic exchanges of state messages at a priori defined global points in time. It provides access to the state messages for the host computer via the CNI, which is implemented by a Dual Ported RAM (DPRAM). The DPRAM contains state messages that are either written by the communication controller with data read from the TTP network or written by the host computer and sent on the TTP network. In addition, the DPRAM contains control and status information (e.g., global time, membership information).

The operating system of the host computer is the time-triggered operating system TTPos [27]. The scheduling decisions in TTPos are controlled by a static schedule table, which contains information about the global points in time for the activation and preemption of tasks within the cluster cycle. The cluster cycle is the least common multiple of all task periods, thus denoting the periodicity of the scheduling decisions. The schedule table is constructed off-line by design tools and ensures that all mutual exclusion and precedence constraints of application tasks are met. TTPos also exploits the knowledge about the global points in time of the communication activities for implicit synchronization between the host computer and the communication controller. Tasks are scheduled in such a way that no concurrent accesses of the CNI through the application software in the host computer and the communication controller can occur.

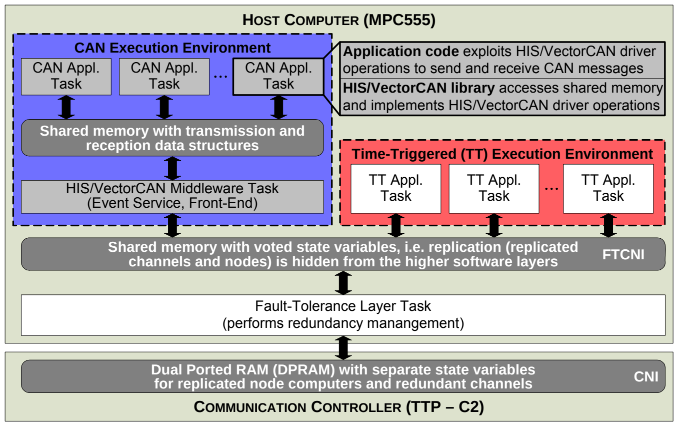

Host Computer (MPC555)
CAN Execution Environment
CAN Appl. Task
CAN Appl. Task
CAN Appl. Task
Application code exploits HIS/VectorCAN driver operations to send and receive CAN messages
HIS/VectorCAN library accesses shared memory and implements HIS/VectorCAN driver operations
Shared memory with transmission and reception data structures
Time-Triggered (TT) Execution Environment
TT Appl. Task
TT Appl. Task
TT Appl. Task
HIS/VectorCAN Middleware Task (Event Service, Front-End)
Shared memory with voted state variables, i.e. replication (replicated channels and nodes) is hidden from the higher software layers
FTCNI
Fault-Tolerance Layer Task (performs redundancy manangement)
Dual Ported RAM (DPRAM) with separate state variables for replicated node computers and redundant channels
CNI
Communication Controller (TTP – C2)

Figure 5: Node with CAN and TT execution environments

NODES WITH SUPPORT FOR TIME-TRIGGERED AND CAN-BASED APPLICATIONS

A node as depicted in Figure 5 supports the coexistence of time-triggered and CAN-based application software. The host computer of the node provides two execution environments, namely a CAN execution environment and an execution environment for time-triggered applications. The latter execution environment supports one or more application tasks that use a time-triggered communication service. The CAN execution environment contains periodic and sporadic application tasks, which require a CAN communication service. Both execution environments built on top of the fault-tolerance layer, which is implemented as a middleware task. In the implementation, the fault-tolerance layer provides the FTCNI to the higher layers using a shared memory containing the voted state messages. The tasks of the time-triggered execution environment access this memory region directly, while the tasks of the CAN execution environment use a second middleware service: the HIS/VectorCAN middleware. The HIS/VectorCAN middleware realizes the VCN with the event service and the front-end. The HIS/VectorCAN middleware provides another shared memory with transmission and reception data structures containing handles representing transmit or receive objects. For offering to the CAN application tasks the API of the HIS/VectorCAN driver, we use in conjunction with the middleware task a HIS/VectorCAN library. This library is linked to the application code in order to offer the operations of the HIS/CAN API, which access the shared memory with the transmission and reception data structures.

TASK STRUCTURE OF THE IMPLEMENTATION

We can distinguish five types of TTPos tasks in the implementation (see Figure 6). Except for the background task, all tasks are time-triggered and scheduled at a priori specified global points in time within each cluster cycle. The fault-tolerance layer task is employed for both the time-triggered execution environment and the CAN execution environment. The CAN application tasks and the HIS/VectorCAN middleware task are specific to the CAN execution environment. In addition, time-triggered application tasks form the time-triggered execution environment.

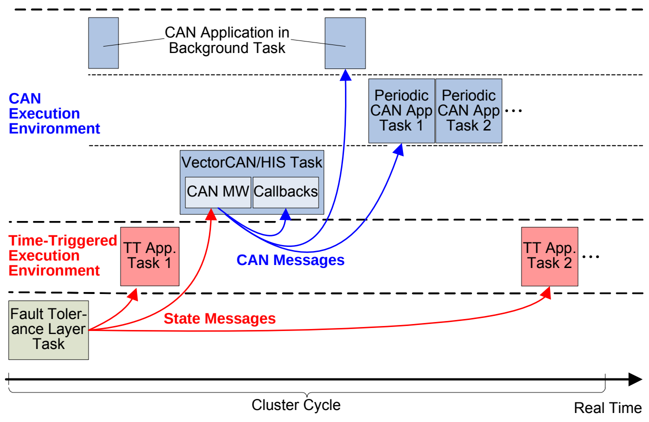


Figure 6: Task schedule with five types of tasks: *fault-tolerance layer task, time-triggered application task, periodic CAN application task, HIS/VectorCAN middleware task, background task*

1. Fault-Tolerance Layer Task: This task implements the fault-tolerance layer by mapping the CNI onto the FTCNI. Thereby, the fault-tolerance layer also hides the implementation details of the time-triggered communication controller (i.e., TTP–C2 [28] in the implementation). In addition, the fault-tolerance layer task maps multiple state variables to a single TTP frame. For this purpose, the fault-tolerance layer packs all outgoing state variables of a node into a TTP frame for transmission on the network. Adversely, each received TTP frame is used to update a corresponding set of incoming state variables in the FTCNI. In the implementation of the HIS/CAN API, each node reserves in the FTCNI one outgoing state variable for transmitting CAN messages and one incoming state variable for each other node in the system deployed with a CAN execution environment. The fault-tolerance layer packs these messages into TTP frames together with the state variables exchanged by the time-triggered applications.

2. Time-Triggered Application Tasks: The time-triggered application tasks are designed to access the FTCNI, which offers a temporal firewall interface [29]. The obligations of a time-triggered application task consist of the timely update (producer obligation) and use (consumer obligation) of the state variables. It is the obligation of a time-triggered application task to update its output variables to maintain temporal accuracy [30] of the real-time images contained in the state variables. The consumer obligation of a time-triggered application task relates to the use of input variables containing real-time images. Based on the a priori knowledge about the temporal accuracy of the real-time images in the FTCNI, the consumer must sample the information in the FTCNI with a sampling rate that ensures that the accessed real-time image is temporally accurate at its time of use.

3. Periodic CAN Application Task: Periodic CAN application tasks incorporate periodic application activities that access a VCN via the API of the HIS/VectorCAN driver. In contrast to the time-triggered application tasks, the CAN application tasks employ event-triggered communication at a priori unknown points in time. During each activation cycle, a CAN application task can send multiple times or not at all depending on the occurrence of significant events. CAN application tasks exploit the HIS/VectorCAN middleware to invoke the operations of the HIS/CAN API for sending, receiving, configuration, error handling, and callback registration. In particular, CAN application tasks can configure the handles of the HIS/VectorCAN driver, e.g., by setting identifiers, filter masks, and data length codes. The interface towards the HIS/VectorCAN middleware is a shared memory, which contains transmit and receive handles. Access to this shared memory is implicitly synchronized by scheduling the periodic CAN application tasks and the time-triggered application tasks in such a way that concurrent access is prevented.

4. HIS/VectorCAN Middleware Task: Based on the FTCNI of the fault-tolerance layer task, the HIS/VectorCAN middleware task realizes the VCN and establishes the HIS/CAN API. The HIS/VectorCAN middleware task is executed once per cluster cycle. It copies CAN messages between the FTCNI and the transmit/receive data structure in the shared memory towards the CAN-based application tasks. Upon each invocation, the HIS/VectorCAN middleware task determines whether new CAN messages are present in the FTCNI and copies these messages into the receive data structures. Adversely, the HIS/VectorCAN middleware task copies CAN messages from the transmit data structures into the FTCNI. The size of the state variable reserved for CAN messages in the FTCNI determines the maximum number of CAN messages that can be copied at one invocation of the HIS/VectorCAN middleware task.

The HIS/VectorCAN middleware task also functions as a server task for sporadic application code. Periodic server tasks are a solution for mapping sporadic tasks onto periodic ones [31]. If a sporadic computational activity needs to be executed, it is executed within the server task. The sporadic computational activities are callbacks, which are registered at the HIS/VectorCAN middleware. We support the following types of callbacks:

• **Error handling callbacks.** In these callbacks, the application can react to value and timing message failures. For example, a transmission failure (as indicated by the TTP membership) results in the invocation of an error handling callback. Another example for a failure resulting in such a callback is a syntactically incorrect CAN messages (e.g., invalid data length code).

• **Transmission callbacks.** The transmission callbacks include a so-called pretransmit function, which is used for setting the data bytes of an outgoing CAN message prior to the start of the transmission. Another example for a transmission callback is the confirmation function that is called after the completion of a CAN message transmission.

• **Reception callbacks.** The reception callbacks comprise reception notification functions for user-defined identifier ranges, as well as precopy routines that enable the application to bypass the copying of a message into the reception data structures.

5. Background Task: The background task is an alternative solution for executing sporadic application code. The background task of TTPos is executed in the idle time between the time-triggered tasks, i.e., whenever no time-triggered task requires execution. In analogy to the CAN-based application code in the time-triggered server tasks, a HIS/VectorCAN library is linked to the application code and offers the operations of the HIS/CAN API. The HIS/VectorCAN library employs the shared memory, which contains transmission and reception data structures, for communicating with the HIS/VectorCAN middleware.

However, since the background task is not synchronized with the periodic tasks of the CAN execution environment (i.e., HIS/VectorCAN middleware task and the periodic CAN application tasks), explicit synchronization is required in order to ensure consistency of the shared data structures. Since the background task is only activated when no periodic task requires execution, access operations to the shared memory by periodic tasks are never interrupted by the background task. However, an access operation to the shared memory by the background task could be interrupted by a periodic tasks. For this reason, the operations of the HIS/CAN API, such as those for the transmission of a message, employ short critical sections with a duration of less than 10 μs in the implementation.

In the background task, each critical section is protected by disabling the periodic tasks of the CAN execution environment before accessing the shared memory. After the reading or writing of the shared memory within the operation of the HIS/CAN API is completed, the HIS/VectorCAN middleware task and the periodic CAN application tasks are reenabled. Hence, in the worst case, an operation of the HIS/CAN API delays the activation of the HIS/VectorCAN middleware task or a periodic CAN application task by 10 μs.

While this solution prevents concurrent access to the transmit and receive data structures, a variable execution time for the HIS/VectorCAN middleware task and the periodic CAN application tasks can occur. The maximum duration of a critical section needs to be used for extending the execution slots for the periodic tasks in the CAN execution environment. Of course, the implementation supports the complete disabling of application code within the background task in case the extension of execution slots is regarded as too costly by the designer.

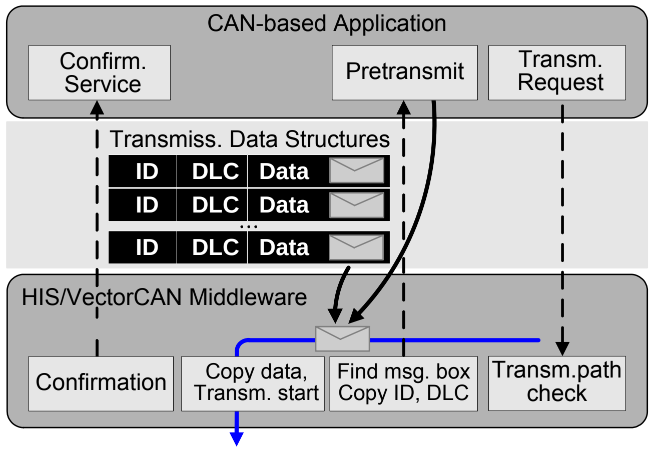


Figure 7: Message transmission

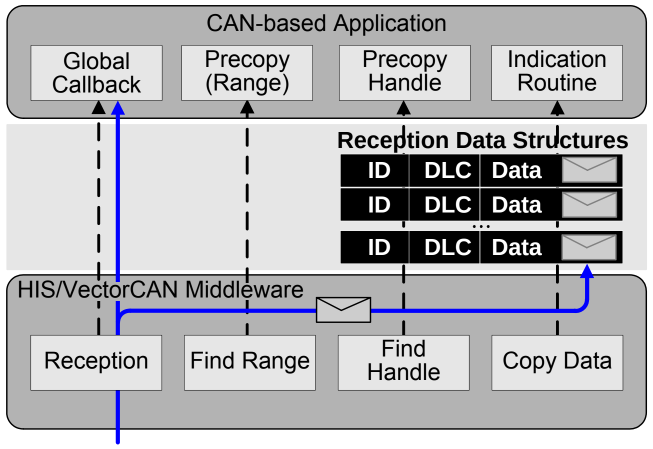


Figure 8: Message reception

MESSAGE TRANSMISSION PROCESS

The CAN-based application initiates the transmission process by invoking a transmit operation at the API provided by the HIS/VectorCAN middleware. The invocation of the transmit operation represents a transmission request from the application, which triggers the activities comprising the transmission process depicted in Figure 7. The application identifies the CAN message that shall be sent via a handle.

After being invoked by the transmit operation, the HIS/VectorCAN middleware searches for the entry in the transmit data structure that matches the handle specified by the application. The identifier, the data length code, and the data bytes are copied into the transmit data structure.

If a pretransmit function is registered as a callback for the handle, it is now called in order to give the CAN-based application the chance to execute application-specific code prior to the message transmission on the network. In particular, the CAN-based application is passed a reference to the memory holding the message, thus allowing it to modify the contents of the CAN message (identifier, data length code, data bytes).

The HIS/VectorCAN middleware, then, passes the message to the event channel where the message is broadcast via the VCN. For this purpose, the event channel buffers the message in a queue until the next occurrence of the node's slot in the underlying TDMA scheme [7, 20]. After the message has been transmitted via the event channel, a callback called the confirmation function is invoked. This function contains application-specific code that is executed following a message transmission.

MESSAGE RECEPTION PROCESS

The time-triggered HIS/VectorCAN middleware task checks upon each invocation whether CAN messages have arrived from the network. If a message is present in the FTCNI, the HIS/VectorCAN middleware invokes the global callback function (see Figure 8), which is a callback of the CAN-based application that is passed the complete message as a parameter.

The HIS/VectorCAN middleware, then, determines if the message identifier matches one or more of the four identifier ranges. The identifier ranges are statically set via ternary strings: 0, 1, and X, where 0 defines an identifier bit which must be 0, 1 defines an identifier bit which must be 0 and X denotes a bit which does not care in the matching process. Each of the four ranges possess a range-specific callback function, which is called in case of an identifier match.

In the next step, the HIS/VectorCAN middleware determines whether the receive data structure contains a handle matching the identifier of the received message. Thereby, the middleware implements message filtering, because the message is discarded when there is no match with a handle in the receive data structure. If a handle has been found, the handle is passed as a parameter to a precopy function in the CAN-based application. After the execution of the precopy function, the HIS/VectorCAN middleware writes the contents of the message (identifier, data length code and data bytes) into the handle's entry in the receive data structure. Finally, an indication routine in the CAN-based application is called in order to allow the CAN-based application to read and process the new message in the receive data structure.

RESULTS AND DISCUSSION

This section discusses the ability of the implementation of the HIS/CAN API to enable the reuse of CAN-based legacy applications in an integrated time-triggered architecture. We have executed test application programs for validating the HIS/VectorCAN driver transmission and reception behavior with different types of callbacks (e.g., pretransmit, precopy, notifications). In addition, we have looked at the ability of the HIS/VectorCAN middleware to operate in different driver states (e.g., online, offline, partially offline) and handle message overloads.

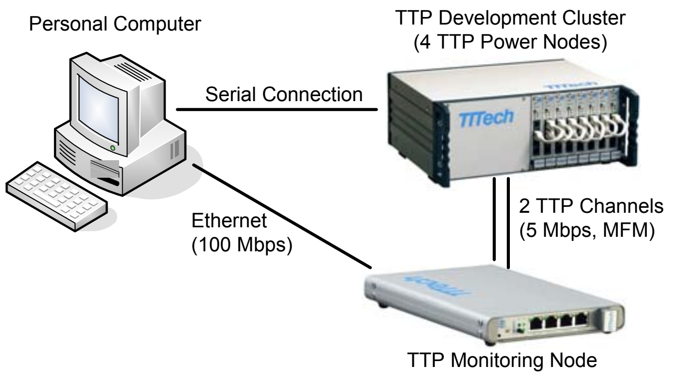


Figure 9: Test setup including a PC and a TTP development cluster with a monitoring node

The test application programs have been executed on a cluster with four TTP power nodes. Test records with information about message transmissions, message receptions, and invocations of callbacks have been sent to a PC via a serial connection. In addition, a TTP monitoring node in conjunction with the TTP View tool [32] running on the PC has been employed to monitor the message exchanges while the TTP network is operating. The complete setup is depicted in Figure 9.

The TTP network has been configured with a message descriptor list providing a cluster cycle with 2 TDMA rounds. Each TDMA round has a duration of 5 ms and consists of four slots, allowing each node computer to send once per TDMA round.

In TTPos, we have scheduled the CAN applications both as time-triggered tasks and within the background task. The time-triggered tasks encompass either periodic computational activities or operate as a server task in order to support sporadic computational activities. The background task serves applications that do not fit into a time-triggered task schedule.

HIS/VECTOR CAN DRIVER FUNCTIONALITY

As part of the validation, we have evaluated the ability of the HIS/VectorCAN middleware and library for providing the functionality defined by the HIS/VectorCAN driver specification [9]. We have performed tests for validating the behavior of the transmission and receptions paths. The testing of the reception path has focussed on the message reception callbacks and filtering functionality. We have tested the receive path including the generic precopy function, specific precopy functions for each receive handle and the indication service.

The software acceptance filter mechanism has been tested by sending dynamic messages on one node with identifiers only partly included in the receive structures of all the other nodes. The identifier range filter mechanism has been tested by defining four different range-specific precopy functions, with each range accepting different identifiers. One node has been sending messages with the corresponding identifiers and the execution of the correct range-specific precopy function has been checked. The validation of the transmission path has included the testing of the pretransmit functions and the internal data copy mechanism, as well as the testing of the confirmation service (confirmation flags and functions) that is invoked after a successful transmission. Furthermore, we have looked at overload scenarios. For example, we have validated that a CAN application that requests the transmission of more messages then fit into the transmission data structures is informed by an error code about the buffer overflow.

In addition, we have checked the behavior of the CAN execution environment in different HIS/VectorCAN driver states. By switching the CAN API on the sending node in regular intervals between active and passive state, online and offline state, and online and partial offline state.

TEMPORAL PERFORMANCE

The message throughput of the VCN is determined by the frequency and size of the TTP slots dedicated to CAN message transmissions. The test implementation of the VCN allows the maximum transmission of 10 CAN messages during a cluster cycle of 10 ms, which is 1 CAN message per millisecond. Execution time measurements of the test program have shown that one execution of the transmit operation of the HIS/CAN API including a simple pretransmit function takes 0.020 ms. The execution of the HIS/VectorCAN middleware task without confirmation and indication services needs a time budget of 0.552 ms if 10 CAN messages are sent and 30 CAN messages are received. Adding the time needed to send 10 CAN messages (0.2 ms) as well as the time budgets of the fault-tolerance layer (0.8 ms) results in a CPU usage of 15.5% in a cluster cycle of 10 ms.

DEPENDABILITY

In the presented implementation, the CAN communication service is established on top of the fault-tolerant time-triggered communication protocol TTP. The HIS/VectorCAN middleware and thus all CAN-based applications build on top of the fault-tolerance layer, which supports fault-tolerance through active redundancy (e.g., two redundant communication channels) transparently to the application software. Consequently, the resulting system shows no change in the timing behavior for both time-triggered and CAN-based applications for faults that are covered by the fault hypothesis of TTP [33].

CONCLUSION

In the automotive domain, reuse of existing software has become not only a concern for suppliers, but also for the car manufacturer. Important aspects are the reuse over different car generations, as well as the reuse over different type series. The difference between suppliers and manufacturers is that the manufacturers have to deal with the complete automotive system and perform the integration of automotive functions via a network of distributed computer systems. A major requirement for software reuse is the separation of the hardware of an ECU from its embedded software. The API of the HIS/VectorCAN driver aims at providing such a high-level interface for CAN-based ECUs that allows to abstract from the underlying implementation details (e.g., a specific CAN controller).

In this paper, we have shown a solution for establishing the API of the HIS/VectorCAN driver for a virtual CAN network on top of an underlying time-triggered communication protocol. The developed solution enables car manufacturers to reuse existing HIS/VectorCAN driver-based applications in the time-triggered computer systems of future car generations. Due to the benefits with respect to dependability, predictability, and available bandwidth, time-triggered networks will be used as the communication infrastructure of upcoming advanced driver assistance systems and future by-wire cars. In this context, the API of the HIS/VectorCAN driver on top of a time-triggered communication protocol enables car manufacturers to reduce the number of CAN ECUs by moving the software into the nodes of the time-triggered system, while retaining the investments in existing CAN-based application software. This strategy promises lower cost due to a significant reduction of wiring and the overall number of ECUs.

In addition, we have presented a three layer model for the middleware services that provide the CAN communication service on top of the time-triggered protocol. Consequently, the middleware can easily accommodate different underlying time-triggered protocols and different APIs. In particular, this modular concept makes the presented solution also useful for applications building on different APIs and higher protocols (e.g., CANopen, DeviceNet).

ACKNOWLEDGMENTS

This work has been supported in part by the European IST project ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

REFERENCES

[1] H. Kopetz and G. Bauer. The time-triggered architecture. IEEE Special Issue on Modeling and Design of Embedded Software, January 2003.

[2] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. FlexRay Requirements Specification Version 2.1, December 2005.

[3] J. Rushby. Bus architectures for safety-critical embedded systems. In Proc. of the 1st Workshop on Embedded Software, October 2001.

[4] H. Kopetz. Why time-triggered architectures will succeed in large hard real-time systems. In Proc. of the 5th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, Cheju Island, Korea, August 1995.

[5] E. Bretz. By-wire cars turn the corner. IEEE Spectrum, 38(4):68–73, April 2001.

[6] Robert Bosch Gmbh, Stuttgart, Germany. CAN Specification, Version 2.0, 1991.

[7] R. Obermaisser. CAN Emulation in a Time-Triggered Environment. In Proc. of the 2002 IEEE Int. Symposium on Industrial Electronics (ISIE), volume 1, pages 270–275, 2002.

[8] B. Hardung, T. Kolzow, and A. Kruger. Reuse of software in distributed embedded automotive systems. In EMSOFT'04: Proc. of the fourth ACM Int. conference on Embedded software, pages 203–210, New York, NY, USA, 2004. ACM Press.

[9] Volkswagen AG. HIS/VectorCAN driver specification 1.0. Technical report, Berliner Ring 2, 38440 Wolfsburg, August 2003.

[10] CANopen application layer and communication profile v4.0.2. Technical report, CiA DS 301, March 2005.

[11] Honeywell Inc., Micro Switch Division. Smart distributed system: Application layer protocol v2.0. Technical report, Freeport, IL, USA, April 1999.

[12] D. Noonen, S. Siegel, and P. Maloney. DeviceNet application protocol. In Proc. of the 1st Int. CAN Conference, Mainz, Germany, 1994.

[13] Analysis of the european automotive in-vehicle network architecture markets. Technical report, Frost & Sullivan, October 2004.

[14] J. Swingler and J.W. McBride. The degradation of road tested automotive connectors. In Proc. of the 45th IEEE Holm Conference on Electrical Contacts, pages 146–152, October 1999.

[15] Int. Standardization Organisation, ISO 11898. Road vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication, 1993.

[16] J. Rufino, P. Ver´ssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In Proc. of the 28th Int. Symposium on Fault-Tolerant Computing Systems, pages 150–159, Munich, Germany, June 1998.

[17] EN 50325-2: Industrial communications subsystem based on ISO 11898 (CAN) for controller-device interfaces – part 2: DeviceNet. Technical report,

CENELEC – European Committee for Electrotechnical Standardization, 2000.

[18] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In Proc. of Sixth IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing, pages 51–60, May 2003.

[19] H. Kopetz. Specification of the TTP/C Protocol. TTTech, Schonbrunner Straße 7, A-1040 Vienna, July 1999. Available at http://www.ttpforum.org.

[20] R. Obermaisser. Event-Triggered and Time-Triggered Control Paradigms – An Integrated Architecture. Real-Time Systems Series. Kluwer Academic Publishers, November 2004.

[21] C. Tanzer, S. Poledna, E. Dilger, and T. Fuhrer. A fault-tolerance layer for distributed fault-tolerant hard real-time systems. In Proc. of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, San Juan, Puerto Rico, USA, April 1998.

[22] A. Avizienis. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. In Proc. of the Int. conference on Reliable software, pages 458–464, 1975.

[23] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In Proc. of 12th Int. Conference on Distributed Computing Systems, Japan, June 1992.

[24] TTTech Computertechnik AG, Schonbrunner Strasse 7, A-1040 Vienna, Austria. TTP Powernode – The TTP Development Board, 2004.

[25] LIN Consortium. LIN Specification Package Revision 2.0, September 2003.

[26] TTTech Computertechnik AG, Schonbrunner Strasse 7, A-1040 Vienna, Austria. Time-Triggered Protocol TTP/C – High Level Specification Document, July 2002.

[27] TTPos - the time-triggered and fault-tolerant RTOS. In Real-Time Magazine 99-4, 1999.

[28] TTTech Computertechnik AG, Schonbrunner Strasse 7, A-1040 Vienna, Austria. TTP/C Controller C2 Controller-Host Interface Description Document, Protocol Version 2.1, November 2002.

[29] H. Kopetz and R. Nossal. Temporal firewalls in large distributed realtime systems. In Proc. of IEEE Workshop on Future Trends in Distributed Computing, Tunis, Tunesia, 1997. IEEE Press.

[30] H. Kopetz. Real-Time Systems, Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, 1997.

[31] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. Journal of Real-Time Systems, (1):27–60, 1989.

[32] I. Smaili. Real-Time Monitoring for the Time-Triggered Architecture. PhD thesis, Technische Universitat Wien, Institut fur Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2004.

[33] H. Kopetz. The fault hypothesis for the time-triggered architecture. In Proc. of the IFIP World Computer Congress, 2004.