Chapter 2
Reference Architecture of a CAN-Based System

A CAN communication system requires the implementation of a complex protocol stack, from the application-level programming interface (API), down to the hardware implementation of the Medium Access Control (MAC) and Logical Link Control (LLC) layers inside the CAN adapter. Several properties of the communication, including latencies of messages, with possible priority inversions and loss of any type of time predictability, depend on the design choices at all levels including the hardware architecture of the peripheral adapter (e.g., available number of message buffers and their management), the structure of the operating system and middleware layers (e.g., the model of the operations for enqueuing and dequeuing messages in the device driver, OS, and Interaction layer), and the I/O management scheme (e.g., interrupt-based or polling-based) inside the device driver.

Figure 2.1 is a representation of the CAN communication architecture, as prescribed by the OSEK COM automotive standard [10] and appears, with little or no substantial change, in several commercial products. The main components of the architecture are:

- The *CAN Controller* is the (hardware) component that is responsible for the physical access to the transmission medium. It provides registers for the configuration of the connection to a bus with given characteristics, including the selection of the bit rate, the bit sample time, and the length of the interframe field. The controller also provides the functionality for managing all the aspects of the CAN protocol, including the management of the transmission modes and the handling of the bus off state. The controller offers a number of data registers for holding messages for outgoing transmissions and incoming data, and provides support for message masking and filtering on reception.
- The *CAN Device Driver* is the (software) component responsible for several tasks related to the low-level transmission of messages. The device driver initializes the CAN controller, performs the transmission and reception of individual messages, handles the bus-off state, and (when available) handles the sleep mode and the

M. Di Natale et al., *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice*, DOI 10.1007/978-1-4614-0314-2_2,

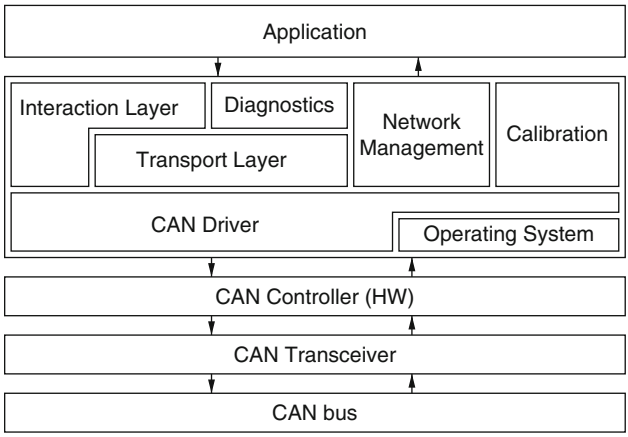

Fig. 2.1 Typical architecture of a CAN-based system. Actual systems may have different dependencies among layers

transitions in and out of it. Most CAN drivers also extend the controller capability with respect to the number of available transmission buffers, by providing software queue(s) for the temporary storage of messages.

- The *Transport* layer handles message segmentation and recombination whenever the upper layers, including the application-level functions, require the transmission of data with length exceeding 8 bytes. The transport layer provides mechanisms for sending acknowledgements and controlling the flow. The layer may also assist in establishing virtual channels and providing an extended addressing scheme.
- The *Interaction* layer offers an API to the application for reading and writing data according to application-defined types. In addition, it typically provides services for flow control, acknowledgement, and periodic transmission of messages. The interaction layer provides buffering and queuing of messages with several options. This layer also performs byte order conversion and message interpretation.
- The *Diagnostic and Network Management* layers determine and monitor the network configuration and provide information about the network nodes that are active. These layers also manage synchronized transition of all network nodes to the bus sleep mode when requested, and the management of virtual (sub)networks or network working modes.
- The *Calibration* layer supports the CAN Calibration Protocol (CCP). The CCP protocol was defined as a standard by the European ASAP automotive manufacturers working group (ASAP is an acronym from the German words for Working Group for the Standardization of Calibration Systems), and later handed over to the ASAM (Association for the Standardization of Automation and Measurement Systems). CCP is a high-speed interface based on CAN for measurement and calibration systems. The CCP protocol provides features for the development and testing phase as well as for end-of-line (flash) programming,

leveraging the general capability to read from and write into ECU memory locations for the purpose of parameter calibration and measurement.

The following sections provide additional details on the functionality, the design, the implementation options, and the management policies of the above hardware controllers and software layers.

2.1 CAN Controller and Bus Adapter

There are several CAN controllers available on the market. Some are available as stand-alone chips; however the most common option is that of a CAN controller integrated in a microcontroller unit, with a processing core, memory, and other devices. CAN controllers are of two types: *basic* and *full*.

In basic CAN controllers, hardware support is limited to the minimum logic necessary to generate and verify the bitstream according to the protocol. Only one transmission data register (or hardware buffer) is available for messages; if the node needs to transmit more than one message, then the upper layers need to buffer the messages in software queues. Two data registers are dedicated to the reception of messages. Hardware support for acceptance filtering is drastically limited in these controllers. Eight-bit code and mask registers are made available for the selection of the messages to be received. This means that filtering is limited to the 8 most significant bits (MSB) of the identifier. This might be barely sufficient in systems with 11-bit (standard) identifiers. When 29-bit identifiers are used, however, the hardware filtering must be supplemented by software. The two receiving buffers are usually accessed in FIFO (First-In-First-Out, the message that is extracted is the oldest among the two) order, so that a message can be received into one buffer while the microcontroller is reading the information from the other buffer. If the software does not respond in time to the reception interrupt signal and one more message is received while both receiving buffers are full, then the latest received message is overwritten and lost. Basic CAN controllers used to be popular because of their lower cost. Today, however, they are less common because the higher integration of electronic components allows the fabrication of full controllers at a very competitive price.

Full CAN controllers offer several *Objects* or data registers, typically configurable to act on the transmission or reception side. These controllers are capable of performing masking and filtering at the level of the individual reception registers (referred as *RxObjects*), and are also capable of arranging the transmission of messages in the (multiple) available transmission buffers (referred as *TxObjects*) according to a given policy. *RxObjects* and *TxObjects* are typically mapped in the RAM addressing space and made accessible to the controller DMA to free the processor from the burden of data transfer. Full CAN controllers are capable of performing full acceptance filtering for each reception object. In addition, if a remote (request) message is received, the controller is capable of performing an

Table 2.1 CAN controllers

Model	Type	Buffer Type	Priority and abort
Microchip MCP2515	Standalone controller	2 RX–3 TX	Lowest message ID, abort signal
ATMEL AT89C51CC03 AT90CAN32/64	8 bit MCU w. CAN controller	15 TX/RX msg. objects	Lowest message ID, abort signal
FUJITSU MB90385/90387 90V495	16 bit MCU w. CAN controller	8 TX/RX msg. objects	Lowest buffer num. abort signal
FUJITSU 90390	16 bit micro w. CAN controller	16 TX/RX msg. objects	Lowest buffer num. abort signal
Intel 87C196 (82527)	16 bit MCU w. CAN controller	14 TX/RX+1 RX msg. objects	Lowest buffer num. abort possible (?)
INFINEON XC161CJ/167 (82C900)	16 bit MCU w. CAN controller	32 TX/RX msg. objects (2 buses)	Lowest buffer num., abort signal
PHILIPS 8xC592 (SJA1000)	8 bit MCU w. CAN controller	One TX buffer	Abort signal

automatic comparison between the identifier indicated in the remote request and the identifiers of the messages in the *TxObjects*. If a match is found, the controller automatically sends the message. Of course, the software layers above the device driver are still responsible for ensuring that the data content of the message to be transmitted is up-to-date.

The configuration and management of the peripheral buffers is of utmost importance in the evaluation of the priority inversion at the adapter and of the (worst case) blocking times for real-time messages. For example, in [60], message latency is evaluated with respect to the availability of TxObjects for the messages, and the possibility of aborting a transmission request and changing the content of a TxObject.

A large number of CAN controllers is available in the market. Table 2.1 summarizes information on seven controllers from major chip manufacturers. The chips listed in the table are Micro Controller Units (MCUs) with integrated controllers or simple bus controllers. In case both options are available, the product codes of the integrated controller are shown between parenthesis. All controllers allow both polling-based and interrupt-based management of both the transmission and the reception of messages.

For the purpose of time predictability, controllers from Microchip [47] and ATMEL [14] exhibit the most desirable behavior (see the following chapter on worst-case timing analysis). These chips provide at least three transmission buffers, and the peripheral hardware selects the buffer containing the message with the lowest ID (the highest priority message) for attempting a transmission

whenever multiple messages are ready. Furthermore, in those controllers, message preemption is always possible with the following behavior: the pending transmission communications are immediately aborted but the on-going communication will be terminated normally, setting the appropriate status flags.

Other chips, from Fujitsu [28], Intel [33], and Infineon [32], provide multiple message buffers, but the chip selects the lowest buffer index for transmission (not necessarily the lowest message ID) whenever multiple buffers are available. In [60] it is argued that, since the total number of available buffers for transmission and reception of messages is 14 (for the Intel 82527), the device driver can assign a buffer to each outgoing stream and preserve the ID order in the assignment of buffer numbers. Unfortunately, this is not always possible in several applications (including automotive) or when the node is acting as a gateway, because of the large number of outgoing streams. Furthermore, in several systems, message input may be polling-based rather than interrupt-based and a relatively large number of buffers must be reserved to input streams in order to avoid message loss due to overwrite.

Finally, the Philips SJA1000 chip [55], an evolution of the 82C200 controller discussed in [60], did not improve much on the shortcomings of its 82C200 predecessor, that is, a single output buffer, although with the possibility of aborting the transmission and performing preemption.

2.2 CAN Device Drivers

The CAN driver includes several functions that can be roughly classified into

- *Initialization and restart* for the initialization of the software data structures and the controller (hardware) registers both after power-on and after bus-off.
- *Transmission of messages*, providing for the transmission of a single message, and the handling of errors and exceptions (the Data Link Layer functionality).
- *Reception of messages*, both by interrupt or polling, with filtering and several queuing options on the receiving side.
- *Other functions*, including mode management, sleep states and wakeups.

The following sections provide details about the functionality and the data structures for each set of services.

2.2.1 Transmission

On the transmit side, the device driver software provides a function for requesting the transmission of a single CAN message. When the function is called, the driver first checks whether the transmitter is in the off-line state or not. Then, the driver checks the availability of a TxObject in the controller. If all the TxObjects are occupied by messages or not usable by the message to be transmitted (the allocation

of TxObjects to messages is one of the policies under the control of the driver), the request can be stored temporarily in a software transmit queue. Some drivers may offer the option of evicting one of the messages from the adapter TxObjects to make it available for the newly arrived transmission request. Although this option makes sense if the transmission request is for a high-priority message (at the very minimum higher than one of the messages in the TxObjects), few drivers make it available. The number of available TxObjects depends on the specific CAN controller, but their assignment to messages is under the control of the CAN driver. TxObjects may be dedicated to messages in exclusive mode (avoiding the need to copy the DLC and ID fields) or associated with a set of messages to be put in a software queue. In any case, upon a request for transmission, if one TxObject is available, the message data is copied into it. Otherwise, the message is enqueued, and the transmit function typically returns with an acknowledgement code signalling that the transmit request was accepted by the driver. The message copy from the data buffer associated to the message to the CAN controller hardware register may be performed by a user notification function or by the CAN driver itself. If the transmission is rejected for any reason (error, absence of available TxObjects and/or no available software queue), then the application is notified and it is typically responsible for re-trying the transmit request.

After the message content has been copied into the adapter transmit buffer, transmission is enabled on the CAN controller. In most cases, completion of the transmission will be signalled by the arrival of an interrupt signal (if the driver enables the interrupt management of the empty transmit object event). A polling-based management is also possible; however this will be highly undesirable for time predictability. The interrupt service routine activated by the interrupt signal right after the successful transmission of a message performs a set of actions. First, the user has the option of setting up a confirmation mechanism, in the form of a confirmation flag (to be set by the driver upon reception), or a confirmation function (a notification function defined by the user to be called), or both. If they are used, then the confirmation flag is set and the confirmation function is called.

Next, if the CAN driver is configured to use a transmit queue, then it checks whether the queue is empty. If so, the transmit interrupt routine terminates. If there are messages waiting in the queue, one message is removed, copied into the available TxObject and the transmission of the message is enabled. The selection of the message to be extracted from the queue should be performed by priority (the message with the lowest identifier). Unfortunately, several drivers use a FIFO queue for the temporary storage of messages. The price paid for the apparent benefit of a simpler and faster management of the message queue is the possibility of multiple priority inversions and a very difficult or impossible time predictability, as explained in the next chapter.

In most drivers, the transmit queue holds only the transmission request of a CAN message but not the data content. The message data is stored in a dedicated buffer. On a transmit interrupt, when the driver extracts the message to be transmitted from the queue, it typically retrieves the DLC and Identifier (ID) fields of the message from the descriptor in the queue, and then copies the message data from its

dedicated buffer area to the TxObject. The message copy is performed by the driver in the context of an interrupt service routine, but the same message buffer is also made available to the upper layers for updating the data content. It is typically the user's responsibility to guarantee the data consistency, because a write access by the upper layers (the application) may be interrupted by a transmit interrupt. The client software may ensure consistency by disabling interrupts while writing to the global buffer, or by using some other synchronization mechanism.

2.2.2 Reception

Controller Area Network messages are received asynchronously. For the upper software layers, message reception happens without any explicit service function call, either by using a notification function or asynchronously, through a mailbox. The CAN driver may perform the actual reception of the message (copying the contents from the RxObject buffer register and placing it in a suitable memory area) when it has been informed of the reception by the CAN controller. The controller can either send an interrupt signal, or the driver can be programmed to periodically check the content of the RxObjects using a polling strategy. In the rest of the section, we will assume an interrupt-based management of message reception. The polling option is available in several driver architectures, and the respective effects on the timing performance of messages will be discussed in later chapters.

When the receive interrupt signal arrives and the handler routine is executed, a CAN message has passed the masking and filtering acceptance scheme defined in the adapter hardware, and its contents are stored in a receive register. Unfortunately, in the case of a basic CAN controller, this is not sufficient to guarantee that the message is actually meant to be processed by the node. Also, in all controllers, there is the possibility that the node is acting as a gateway for the message, and needs to process it for forwarding in the shortest possible time. To this purpose, many drivers offer the opportunity of defining a receiver callback routine to be called after the hardware acceptance stage. If the user defines such a function, it may perform application-specific filtering and message processing. In addition, the driver typically offers a default service for software acceptance filtering using linear search, hash-table search or index search. If a CAN message has passed the hardware filter but is rejected by the software acceptance functions (in the case of a *basic CAN* receiving object), a special callback function may be called (if configured). If the received CAN identifier matches the identifiers in the table, the driver continues with the processing of the message received by copying it into ring buffers, FIFOs, and/or a global data buffer. Several CAN drivers optimize the memory required by copying only those bytes used by the application (up to the last byte within the message which contains data of interest for the application). In addition, indication actions may be defined for each received message. If this is the case, the indication flag is set and/or the indication function is called. The application has the responsibility of clearing the indication flag.

Regardless of the number of objects (transmit or receive registers) that are available on the CAN chip, in several current architectures (specifically in the automotive domain) message input is typically polling-based rather than interrupt-based, and all peripheral objects or buffers, except one, are reserved to input streams in order to avoid message loss by overwriting. In these systems, a single transmit object is typically available for all the output messages from a node.

2.2.3 Bus-Off and Sleep Modes

The CAN driver also has the responsibility to notify a detected bus-off state to the application by calling a special callback function. When a node goes in bus-off state, the application may attempt to restore its functionality by re-initializing the CAN controller using a suitable driver function.

Some CAN controller supports a sleep mode with reduced power consumption. In these cases, the driver provides service functions to enter and leave the sleep mode on the request of the application software. If the CAN controller can be automatically awakened by the CAN bus, then a callback is typically made available to the application to inform it of the node wakeup and to make the necessary arrangements.

2.3 Interaction Layer

The description of the typical Interaction Layer (IL) services and structure given in this section follows to a large extent the specification provided for this layer in the OSEK COM standard [10]. On the transmission side, the interaction layer provides:

- signal-oriented and application data-oriented functions to send data over a bus to a given application-level object destination;
- several transmission modes to the application for sending periodic streams or performing transmission on request (with the possibility of specifying a minimum time interval between any two transmissions);
- a notification to the application when a signal is transmitted, which acts as confirmation.

On the reception side, the layer

- provides the application with functions for the reception of application-level data arriving over the CAN bus;
- notifies the application when a new value for a given signal arrives as part of a message payload (indication);

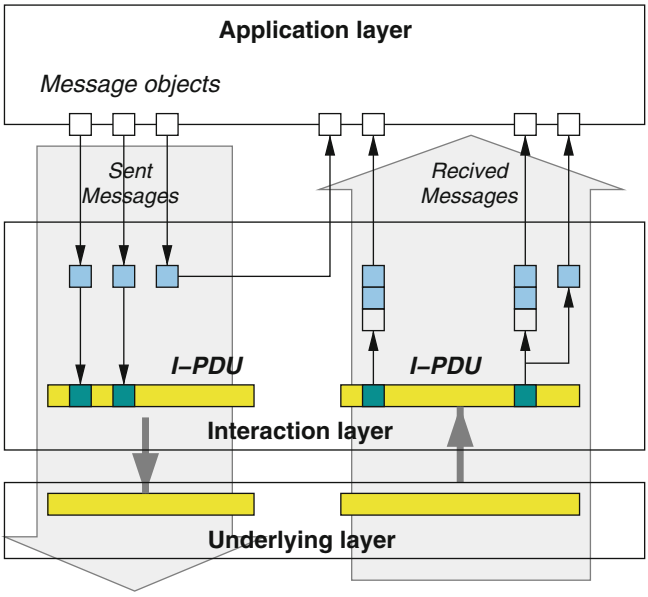

Fig. 2.2 Interaction layer architecture

- allows to control and monitor the reception of periodic data streams (or time-sensitive signals) by allowing the definition of timeouts on reception and callbacks on timeout expiration;
- provides (in case of an error) notification to the application and allows the definition of default values to be set in the reception mailboxes.

The IL also provides support for multiple channels in transmission and reception, and multiplexing (demultiplexing) of information. In OSEK COM, the IL realizes the abstraction of communication from the physical placement of senders of receivers, offering the same API for local as well as remote communication.

Administration of messages is done in the IL based on the concept of *message objects* (in the OSEK COM terminology, also defined as *application signals* in some commercial products). Message objects are associated with application types on both the sending and receiving sides, and have an associated identifier. OSEK COM supports only static object addressing. A statically addressed message object can have multiple receivers (or even no receiver at all!) defined at system generation time. A message object can have a static length or its length may be variable with a maximum defined at design-time (in this case, it is a *dynamic-length* message object). The IL packs one or more message objects into assigned *Interaction Layer Protocol Data Units* (I-PDU), and passes them to the underlying layer (Fig. 2.2). Within an I-PDU, message objects are bit-aligned. The size of an object is specified in bits, and the object is allocated to contiguous bits within its I-PDU. Objects cannot be split across I-PDUs.

The byte order (endianess) in a CPU can differ from other CPUs on the network. Hence, to ensure correct interpretation of message data and to provide interoperability across the network, the IL defines a network representation and provides for the conversion from the network representation to the local CPU representations and vice versa. The conversion is typically statically configured on a per-message basis.

The IL offers a set of functions as part of an application programming interface (API) to handle messages. The API provides services for initialization, data transfer, communication management, notification, and error management. The functions for the transmission of message objects over the network are usually non-blocking. This implies that a function for message transmission does not return a value indicating the transmission status/result because the network transfer is still in progress when the function returns. To allow the application to determine the status of a transmission or reception, the IL provides a notification mechanism.

OSEK COM supports communication from multiple sender objects to multiple receiver objects (m-to-n). Receivers may be on the same node as the source object, or they may be remote. In the case of internal (local) communication, the Interaction Layer (IL) makes the data from source object(s) immediately available to the destination objects (zero, one, or multiple) by direct copy. In the case of remote communication, the data from sender object(s) are copied into zero or one I-PDUs.

I-PDUs may contain data from one or more sending objects and can be received by zero or more CPUs. A receiving CPU reads the I-PDU data content and forwards the data to the destination (zero or more) receiving objects, where the data values become available to the application software. A receiving message object may receive the message from a local sending object or an I-PDU (in case it receives data from the network).

A receiving message object can be defined as either *queued* or *unqueued*. In the case of queued objects, the queue is of FIFO type. The queue size needs to be specified individually for each object. When the queue is empty, the IL does not provide any message data to the application. An object can only be read once, i.e., the read operation removes the oldest object from the queue. If the queue of a receiving message object is full and new object data arrive, the new data is lost and the next request for reception on the IL object returns the (oldest) data values and also the information that a message has been lost. The unqueued message objects can be read multiple times and are not consumed. An arbitrary number of receivers may receive the message object and each of them will get the last received value. This means that unqueued objects are overwritten by newly arrived messages. If no message has been received since the system startup time (the initialization of the IL layer), then the application receives the message value set at initialization. Given that the object values are shared, the IL needs to guarantee that the data in the application's message copies are consistent using a suitable locking or synchronization mechanism.

On the transmission side, an object to be transmitted on the network can have one of the following two transfer properties:

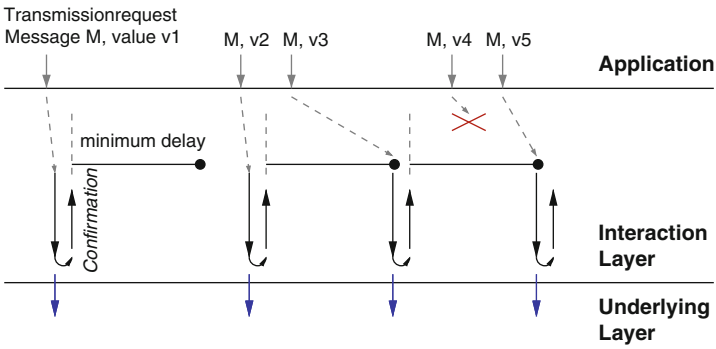

Fig. 2.3 Timing for the transmission of direct mode message objects

- *Triggered Transfer*: the data field in the assigned I-PDU is updated with the object contents, and the IL issues a request for the I-PDU transmission.
- *Pending Transfer*: the object value in the I-PDU is updated, but the I-PDU is not transmitted immediately.

These reflect into three transmission modes for I-PDUs, which are:

- *Direct Transmission* mode: the transmission of the I-PDU is initiated by the request for the transmission of a triggered transfer object.
- *Periodic Transmission* mode: the I-PDU is transmitted periodically by the IL without the need of an explicit request (for pending objects).
- *Mixed Transmission* mode: the I-PDU is transmitted using a combination of both the direct and the periodic transmission modes.

2.3.1 Direct Transmission Mode

The transmission of an I-PDU with direct transmission mode originates from the request for transmission of any object with triggered transfer property mapped into the I-PDU. The request immediately results in a transmission request for the I-PDU from the IL to the underlying layer. Each I-PDU with a direct transmission mode is also associated with a *minimum delay time* between transmissions. The minimum delay time starts at the time the transmission is confirmed from the lower layers. Any request for transmission arriving before the expiry of the minimum delay time is postponed and only considered at the end of the delay interval. As shown in the Fig. 2.3, the value `v1` is assigned to message object `M`, which is transmitted in direct mode. The message is copied in an I-PDU, which is immediately transmitted using the lower level API. Upon reception of the confirmation, the minimum delay timer is started. The following request for transmission for the same object with value `v2` comes after the minimum delay, and is therefore immediately processed.

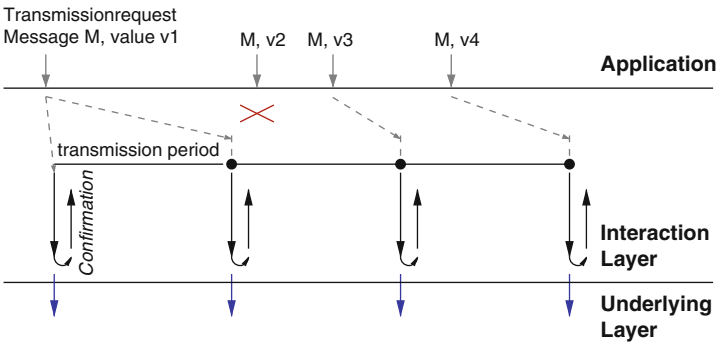

Fig. 2.4 Timing for the transmission of periodic mode messages objects

This is not true for the request associated with value `v3`. In this case, the minimum delay has not expired yet, and the I-PDU transmission will wait until the timer expires. Similarly, the request for `v4` comes before the minimum delay has passed from the previous transmission. In this case, however, before the transmission of the I-PDU containing the value `v4` can be issued, a new value `v5` is written into the object, and the value `v4` is overwritten and lost. The minimum delay time does not apply to the case of transmissions resulting in an error, e.g., expiration of the transmission deadline. In this case, a retransmission attempt can start immediately.

2.3.2 Periodic Transmission Mode

In the periodic transmission mode, the IL issues periodic transmission requests for an I-PDU to the underlying layer. On the application side, the API calls for the transmission of message objects (`SendMessage` and `SendDynamicMessage` in OSEK COM) to update the message object fields of the I-PDU with the latest values to be transmitted, but do not generate any transmission request to the underlying layer. The periodic transmission mode ignores the transfer property of the objects contained in the I-PDU, although clearly only pending transfer objects should be associated with a periodic transmission mode. The transmission is performed by repeatedly calling the appropriate service in the underlying layer with a period equal to the one defined for the periodic transmission mode. Figure 2.4 shows an example of periodic transmission modes. Regardless of the time at which the contents of the sending objects are updated, the I-PDU is transmitted periodically to the lower layers. This implies that some values may be transmitted twice (as is the case for `v1` in the figure), or may be overwritten (as happens to `v2`) and never transmitted over the network.

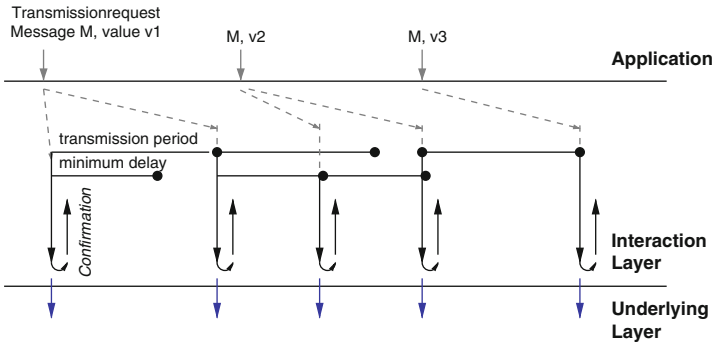

Fig. 2.5 Timing for the transmission of mixed mode messages objects

2.3.3 Mixed Transmission Mode

The mixed transmission mode is a combination of the direct and the periodic modes. Similar to the periodic mode, transmissions are performed with a given period by calling the appropriate service in the underlying layer. However, I-PDUs may also be transmitted upon request in between the periodic transmissions. These requests occur on-demand with the same modality of the direct transmission mode, that is, whenever the application requests the transmission of an object with triggered transfer property mapped into the I-PDU. Intermediate transmissions do not modify the base cycle, but must still comply with the minimum delay time defined between any two transmissions. If a transmission is requested within a shorter time, the I-PDU transmission is postponed until the delay time expires. The minimum delay interval also applies to periodic transmissions. Therefore, a periodic transmission request that occurs before the minimum delay time from any direct transmission will be delayed to ensure that no two transmissions are separated by less than the minimum delay. The following periodic transmission instances will occur after a full period has elapsed from the delayed instance (the entire periodic sequence is shifted in time). An example is shown in Fig. 2.5. The value `v1` is transmitted twice with the regular period of the periodic mode. Then, the value `v2` is transmitted, but not immediately. The I-PDU transmission is postponed to allow a minimum delay from the last periodic instance. Also, the next periodic transmission (again with value `v2`) is deferred to allow for a minimum delay with respect to the last direct transmission.

The periodic transmission sequences of the periodic and the mixed transmission modes are activated by a call to a suitable API service. This call also typically allows to specify the offset for the transmission requests (the time interval to the first transmission instance). Correspondingly, the periodic transmission mechanism is stopped by means of another API function.

2.3.4 Deadline Monitoring

OSEK COM provides a mechanism for monitoring the transmission and reception times of message objects and taking appropriate actions in the case of timing faults. This mechanism is called *Deadline Monitoring*. On the sender side, deadline monitoring verifies that the underlying layer confirms the request for a transmission within a given time period. On the receiver side, the IL checks that periodic messages are received within the time period associated with them. Deadline monitoring applies to external communication only and the monitoring is performed at the level of the I-PDU, that is, controlling the reception of the I-PDU containing the object. Deadlines and actions can be configured individually for each object. The use of this mechanism is not restricted to monitoring the reception of messages (I-PDUs) transmitted using the periodic transmission mode, but also applies to I-PDUs sent using the direct and the mixed modes.

2.3.5 Message Filtering

The IL also provides filtering functions that extend and complement the base filtering mechanisms offered by the adapter and driver layers, based on the identifier of the received message. IL filtering is performed for each object on both the sending and the receiving side. Filtering provides means to discard the message objects received or to be sent when a set of conditions are not met by the message object based on its value content or other attributes. Each message filter is a configurable function constructed by composing a set of predefined algorithms. On the sender side, the filter algorithm decides whether to actually update the I-PDU and send it according to the message contents (for example, avoiding the transmission if the value of the object has not changed). There is no filtering on the sender side for internal transmissions. On the receiver side, a filter mechanism may be used for both internal and external transmission. Filtering is only used for fixed-size messages that can be interpreted as C language unsigned integer types (characters, unsigned integers, and enumerations). In OSEK COM, the filter algorithm can use a set of parameters and values, including the current and last values of the message object, a bitmap used as a mask, a minimum and maximum value, the object period and offset, and the number of occurrences of the message.

2.4 Implementation Issues

In order to complete the discussion about the typical structure and features of a CAN architecture, it is useful to discuss some of the implementation options in more details. As specified in the previous sections, the CAN driver and interaction

layers contain a number of options, protocol specifications (including management of time for timeouts and periodic transmission) and data structures (for queues and buffer storage) for which design, algorithm and code solutions may differ, and the implementation choices may impact the timing performance and the predictability of the message transmission times.

2.4.1 Driver Layer

On the transmission side, the driver puts the messages that need to be sent in a priority based software queue. The queue, sorted by message identifier (priority), is used to assign the transmit object inside the adapter. As described in the previous section, the driver can be configured to provide access to the transmission objects in exclusive mode, or it may associate a software queue to them. In most cases, the output of a software queue is not associated to a set of TxObjects but only to one, with the possibility of setting up multiple queues, one for each TxObject.

As for the implementation, the message queue should be a priority queue, in such a way that whenever a TxObject becomes available, the highest priority (lowest identifier) message is extracted from the queue and copied into it. FIFO queues can easily result in large priority inversions and undesirable message delays in the worst case. For the implementation of a priority queue, different options exist. The simplest and usually inefficient way is a simple sorted list. If the list is kept sorted, the time to add an element (message) to the list (insertion time) is in the order of $O(n_m)$ (the insertion time depends on the number n_m of messages in the queue) while extraction is done in $O(1)$ (constant) time, given that the highest priority message is always the first in the queue. To get better performance, priority queues may be realized as heaps, giving $O(\log(n_m))$ performance for insertions and extractions of messages. There are several heap structures and management algorithms available, including self-balancing binary search trees, binomial heaps, and Fibonacci heaps. The heap structure can be further optimized based on the knowledge of the set of messages that are transmitted by the node. To this purpose, it is worth mentioning that in most cases, the configuration of the CAN driver data structures or even procedures is performed automatically by code generation and configuration tools. Given that in the case of a CAN message queue the values to be sorted in the priority list are known, faster implementations are possible. In general, time efficiency can be traded for memory space. If the largest priority value to be managed is P (integer), a van Emde Boas tree [54] can be used to perform insertion and extraction operations in $O(\log(\log(P)))$ time, at an additional memory cost in the order of $O(\log_2(P))$. Constant time insertions and extractions are also possible at a memory cost $O(P)$. Both methods use bitmaps to store information about the priority values that are currently enqueued.

Other driver implementation issues that are worth discussing are the difference between implementations supporting the *interrupt* based (which is the most common method) and the *polling* based management of transmissions. Messages are

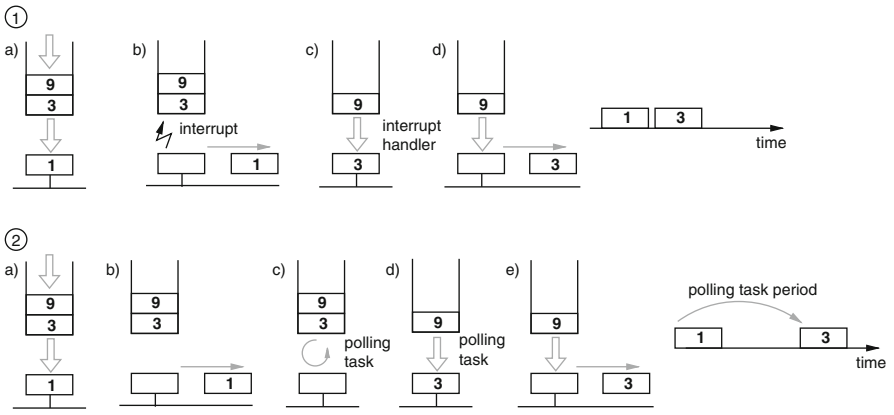

Fig. 2.6 Interrupt- or polling-based management of the TxObject at the CAN peripheral

dequeued by an interrupt handler, triggered by the interrupt signal that informs the completion of the transmission of the message occupying the TxObject at the adapter (shown in sequence 1 of Fig. 2.6), or by a polling task (shown in sequence 2 in Fig. 2.6).

- *Interrupt based*: The interaction layer (or the application if there is no interaction layer) use the driver API function to send messages. Inside the driver, if a TxObject that is a valid destination for the message is empty, it is filled with the message data, otherwise the message is queued. An example is shown in step 1a of the figure, where messages with identifiers 1, 3, and 9 are queued. When a message wins the contention (possibly after a *contention delay*) and leaves the TxObject, the buffer becomes empty and an interrupt is triggered (step 1b). Messages are dequeued by the interrupt handler, executed in response to the interrupt signal. The interrupt handler selects the message on top of the queue (the highest priority message) as the one that has to be placed into the buffer (step 1c). After the transmission of this message, the situation on the bus is the one on the right hand side of the sequence (1): the transmission of the messages that are placed in the queue is typically almost back-to-back, with a possible small separation time that is caused by the response time of the interrupt handler.
- *Polling based*: Messages are queued by the IL as in the previous case. In addition, an I/O driver-level task is activated periodically. When a message is transmitted, the next message in the queue is not immediately fetched to replace it. Only when the period of the polling task expires, it checks if the peripheral buffer is empty (step 2c of the second sequence). If so, it fetches the highest priority message from the queue and transfers it into the available buffer (step 2d). In this case, the typical behavior is that the transmission of messages from the same node is separated by a time interval roughly equal to the period of the polling task (possibly with network idle time in between).

On the receive side, the reception of a message causes a call of the Rx-Interrupt. The interrupt sub-routine compares the incoming message with the existing one. Depending on the result of the comparison, a special flag associated with the message and indicating the status of Changed Data may be set (to enable conditional application-level processing when new values are available). The interrupt sub-routine will copy the message to the buffer of the Interaction Layer.

2.4.2 Interaction Layer

Most of the functionalities required from the interaction layer requires the execution of periodic or time-driven computations. Some of the functionalities check the requests for data transmission, realize the periodic communication for the periodic mode I-PDUs and enforce the minimum delay for all messages. In several implementations these functions are realized by code called periodically at a fixed time interval and executed as part of a task in a multitasking operating system. We call this task *TxTask*, in accordance with the name it is assigned in several commercial solutions [4]. In most cases, the TxTask not only processes and performs the periodic transmissions of I-PDUs by calling the lower layer transmission API functions, but also synchronizes the transmission of all messages. In this case, transmissions on request will just store a transmit request. The actual transmission will be processed by the TxTask and therefore deferred until the next call of TxTask. If the TxTask period equals the minimum delay time, this ensures compliance with the IL protocol requirements for a minimum inter-transmission time. Other tasks to be performed periodically include timeout monitoring. These tasks are performed by a different function, typically called with a fixed period and implemented as part of the TxTask (if the periods are the same) or another task.

Application tasks copy the data values for all the signals that need to be transmitted in variables shared with the TxTask (see Fig. 2.7). Inside the IL, the TxTask is activated periodically and, at some point (typically at the end) of its execution, calls the driver- (or transport-) level function for the transmission of the I-PDU. In many cases, the I-PDUs are limited to be less than 8 bytes, therefore avoiding the need for a transport layer implementation and mapping one-to-one to CAN messages at the driver level.

In addition, each message M_i is defined to be transmitted only in some *virtual network* configurations, corresponding to operating modes of the CAN bus, and has an associated period T_i, expressed as an integer multiple of the TxTask period T_{TX}($T_i = k_i T_{\mathrm{TX}}$). Virtual networks may be defined for the system as the specification of a working mode, in which a subset of the system messages are actually transmitted, with their rates. The set of messages to be transmitted and their periods may be different from one virtual network configuration to another. Virtual networks management is typically part of the network management layer.

Inside the Interaction Layer, a descriptor consisting of a binary moding flag and a counter is associated to each message. The flag indicates whether the message

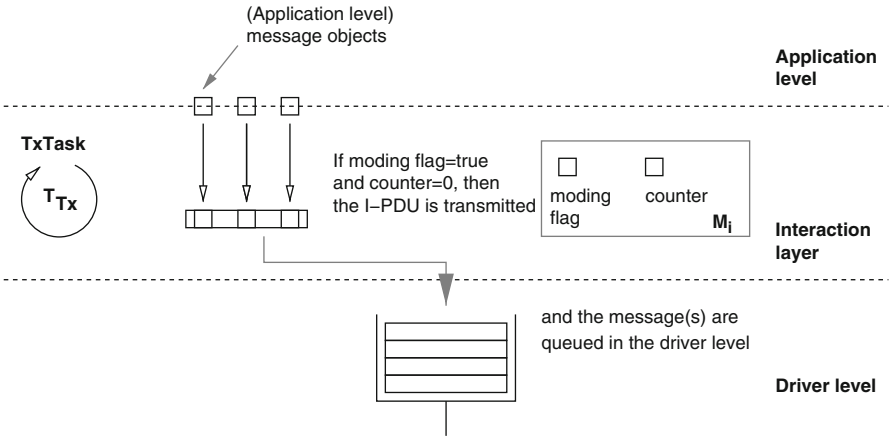

Fig. 2.7 The middleware task TxTask executes with period T_{TX}, reads signal variables are enqueues messages

should be sent for the currently active mode or not. When the TxTask is activated, it scans all the local message descriptors. For each of them, if the moding flag is set, it decrements the counter. If the counter is zero, it reads the values of all the signals that are mapped into the message, assembles the message data packet, and enqueues the packet in the software priority queue.

Some implementations also enforce a periodic processing for the received messages. On the receive side, the interaction layer buffers are used by the driver (often the interrupt handler inside it) to copy the content of the messages that passed the receive filtering at the driver level. The interrupt routine may also signal to the interaction layer the occurrence of a new value as compared to the latest reception (using a flag), and use an indication flag for signalling the occurrence of a newly received message. Inside the IL, a periodic task (called *RxTask*, clearly because of its association with the reception process) will check the occurrence of indication flags on the IL buffers. If the flag is set, the task resets the timeout timer for the message objects and calls the object-related indication function at the application level.

http://www.springer.com/978-1-4614-0313-5

Understanding and Using the Controller Area Network Communication Protocol
Theory and Practice
Di Natale, M.; Zeng, H.; Giusto, P.; Ghosal, A.
2012, XVIII, 226 p., Hardcover
ISBN: 978-1-4614-0313-5