CAN with eXtensible in-frame Reply: Protocol Definition and Prototype Implementation

Gianluca Cena, *Senior Member, IEEE,* Ivan Cibrario Bertolotti, *Member, IEEE,* Tingting Hu, *Member, IEEE,* and Adriano Valenzano, *Senior Member, IEEE*

***Abstract*—Controller Area Network (CAN) has been the de facto standard in the automotive industry for the past two decades. Recently, CAN with flexible data-rate (CAN FD) has been standardized, which achieves noticeably higher throughput. Further improvements are still possible for CAN, by exploiting its peculiar physical layer to carry out distributed operations among network nodes, implemented as atomic transactions mapped on quasi-conventional frame exchanges.**

In this paper, a proposal is made for an extension to the CAN protocol, termed CAN with eXtensible in-frame Reply (CAN XR), which enables upper protocol layers to define new custom services devoted to, e.g., network management, application-specific functions, and high-efficiency data transfer. The key point is that CAN XR retains full backward compatibility with CAN, therefore there is no need to change the protocol specification once again.

***Index Terms*—Controller area network (CAN), industrial control, real-time distributed systems.**

I. Introduction

CONTROLLER Area Network (CAN) was introduced by Bosch at the end of the 1980s for onboard vehicle use. CAN specifications remained mostly stable until 2012, when the CAN with flexible data-rate (CAN FD) protocol was presented [1]. CAN FD relies on the *overclocking* and *oversizing* techniques first appeared in 1999 [2]. Its standardization proceeded quickly and, recently, it has been included in ISO 11898-1 [3]. CAN FD provides a boost in network throughput by about one order of magnitude over *classical* CAN. Moreover, the related network controllers are not expected to be noticeably more expensive than their traditional counterparts. For these reasons, chances are that CAN FD will become the standard solution in the automotive industry in the next years.

Unfortunately, coexistence between classical and FD controllers may be a little tricky [4]–[6]. In fact, the former are unable to correctly decode the new FD data frame format and react by transmitting error frames. This drawback cannot be avoided and is tolerated only because of the performance advantages brought by CAN FD. In some respect, this resembles what happened 20 years ago, when the extended frame format (using 29-bit identifiers) was introduced. For the sake of backward compatibility, every FD controller can be configured so that it behaves exactly the same as a classical CAN device.

One of the most peculiar features of CAN (and CAN FD as well) is its physical layer, which performs a *wired* logical AND among the signals written on the bus by all the transmitting nodes. By design the network size is kept limited, so that the round-trip delay between any pair of nodes is strictly less than the nominal bit time (actually, it has to be shorter than the duration of the time segment that spans from the beginning of each bit to the sampling point). Hence, at any time all nodes in the network virtually see the same bus level (either *dominant* or *recessive*). This feature, sometimes called "*in-bit-time detection,*" is profitably exploited by the medium access control (MAC) layer of CAN to carry out bit-wise arbitration (hence avoiding destructive collisions), quickly detect bus errors (bit monitoring on the sender), perform network-wide error globalization (by means of error frames), adapt the transmitter speed to receivers (through overload frames), etc.

The particular behavior of the CAN bus can be leveraged to achieve additional benefits besides those listed above. For instance, a technique for quickly generating symmetric cryptographic keys was proposed in [7], which can help with the design of security countermeasures (i.e., to grant authentication, data integrity, privacy, and so on [8]). Basically, two nodes generate two random bit patterns, which are first suitably encoded by translating each original bit into a pair of consecutive bits at complementary levels, and then sent over the bus in the data field of a CAN FD frame (the FD format was chosen because of the larger payload), where they merge according to the wired-AND scheme. By analyzing the resulting bus levels, the involved senders are able to obtain information on the original random patterns, which are instead completely hidden to other nodes. This procedure operates correctly only if the two nodes start transmission exactly at the same time and the bit monitoring function is disabled for them. A custom solution was envisaged in [7] to accomplish this task.

Incorporating the ability to carry out such a kind of operations directly in the data-link layer of CAN is certainly advantageous, as it helps preventing additional changes to the protocol (and to the controllers as well) in the foreseeable future. From a practical viewpoint, leaving the frame format unchanged is a strict requirement, as failing to do so would unavoidably introduce both higher costs and serious compatibility issues, which could be hardly tolerated by manufacturers, especially those involved in the production of vehicles.

In this paper, an extensible mechanism is described to solicit a group of CAN nodes to reply in a coordinated way within

G. Cena, I. Cibrario Bertolotti, and A. Valenzano are with the National Research Council of Italy, Institute of Electronics, Computer and Telecommunication Engineering (CNR-IEIIT), I-10129 Turin, Italy.

T. Hu is with the University of Luxembourg, Faculty of Science, Technology and Communication (FSTC), L-4364 Esch-sur-Alzette, Luxembourg.

the same frame exchange. It is termed *CAN with eXtensible in-frame Reply* (CAN XR), and somehow it resembles the in-frame reply feature of Vehicle Area Network (VAN) [9]—a former competitor of CAN—although it allows multiple repliers. The main ideas behind CAN XR were first introduced, in a preliminary form, in [10]. To the best of our knowledge, no other approaches exist in literature aimed at similar purposes. The basic CAN XR mechanism is quite flexible and can be used by the upper protocol layers to "tailor" their own distributed atomic operations. For instance, the following application-level services can be envisaged: static and dynamic data slotting, fast minimum discovery, bit-wise data gathering, distributed consensus, and symmetric key generation according to [7]. Thanks to data slotting, a communication paradigm that resembles FlexRay [11] is supported. Therefore, XR can be leveraged to ease the transition of CAN from event-driven to time-triggered paradigms. Additional features can be conceived as well, which rely on the same mechanism.

The idea of applying data slotting to real-time communications is not new. The *summation frame* concept was introduced two decades ago in INTERBUS and, more recently, in EtherCAT [12] and PROFINET with dynamic frame packing [13]. These solutions are conceived to exploit either ring network topologies or full-duplex links (e.g., Fast Ethernet). Approaches like [14] try to bring slotting on legacy fieldbuses running on a shared bus, like MODBUS, which have no notion of time, so as to achieve temporal coherence in acquisition cycles. In [15] a solution aimed at increasing communication efficiency is described, which dynamically gathers multiple process data units from the same sender in a single CAN FD frame.

Much more interesting, from our point of view, is the time-triggered version of CAN, known as TTCAN [16]. In TTCAN, *basic cycles* are initiated by a specific node, known as the *time master*, which periodically transmits a *reference message* (RM) on the bus. On RM reception, every node resets its *cycle time* (a free-running counter), hence synchronizing operations of nodes through a common time base. Suitable *triggers* are then defined on nodes, each of which activates either frame transmission or reception whenever the cycle time matches the related time mark. Basically, this approach splits each basic cycle into a number of fixed *time windows*, within which frame exchanges take place in a disciplined way.

The main difference between TTCAN and CAN XR is that, the former simply superposes the time-triggered paradigm on CAN, whereas the latter permits multiple nodes to embed their data exchanges into the same CAN frame. This has two important consequences: First, a full-formed CAN message is fit in each time window in TTCAN, whilst protocol control information are not replicated for every piece of data in CAN XR. Second, safety margins between adjacent data exchanges, which have to be taken into account when computing time marks in TTCAN, become unnecessary (and forbidden) in CAN XR. These aspects make communication efficiency of CAN XR sensibly better that TTCAN.

While a new breed of controllers is required to support XR operations, the same frame format and protocol as CAN (or CAN FD) are adopted, so that complete backward compatibility is ensured with existing devices. Since multiple CAN XR nodes are allowed to take part in frame transmissions, overclocking can not be exploited because, in that case, the in-bit-time detection property might no longer hold during the data phase of the frame. In turn, this would make impossible to ensure that the resulting bit sequence on the bus always corresponds to a valid CAN frame (strict requirement for backward compatibility). Therefore, the same bit rate must be set in FD for both the arbitration and data phases. In this paper, classical CAN is taken as a basis for CAN XR, in order to swiftly prove its practical feasibility, although the maximum size of its message data field (8 B) is probably not large enough to offer the same level of benefits as CAN FD.

The paper is organized as follows: in Section II the basic principles behind CAN XR are introduced, while Section III describes the way data slotting is carried out. In Section IV some details are provided about the way XR services can be defined in CAN controllers and, in Section V, a prototype implementation is presented, based on real embedded devices communicating over the CAN bus, where XR protocol operations are emulated in real-time in software. In Section VI some light is shed concerning possible application-level services that rely on CAN XR and the advantages they bring on communication performance.

II. CAN XR

CAN XR is conceived as a proper extension of CAN. It is important to remark that XR extensions apply to both classical CAN and CAN FD, with either base or extended identifiers. Unless strictly necessary, in the following we will refrain from discriminating between the different flavors of the protocol, and will refer to all of them simply as CAN. Moreover, we will denote existing nodes (or controllers), which do not support XR extensions, as *non-XR*.

The most important requirement that drove CAN XR definition is that the sequence of bits sent on the bus by the related nodes shall comply completely to CAN frames, so that no error is raised by non-XR CAN controllers as a consequence of their inability to understand the format of the new frames.

A. Protocol Basics

During message transmission, CAN considers two kinds of nodes, namely the *producer* and the *consumers*. Each message must have exactly one producer, unless other countermeasures are taken to prevent different nodes from sending messages with the same identifier at the same time (which would prevent arbitration from operating correctly and cause unsolvable bus contentions). Frames with fixed data field are exceptions, but they are typically useless, except for frames with an empty data field (e.g., remote frames). Conversely, any number of consumers is allowed for the same message (including having no consumers at all).

In CAN XR the role of producer is played by two kinds of cooperating nodes, namely the *initiator* and the *responders*. Together, they carry out atomic "initiate-response" *XR transactions* over the bus. Basically, the initiator starts a transaction by sending the arbitration and control fields, which together make up the frame *header*. Reception of the header triggers a group of responders and consumers. The former reply by

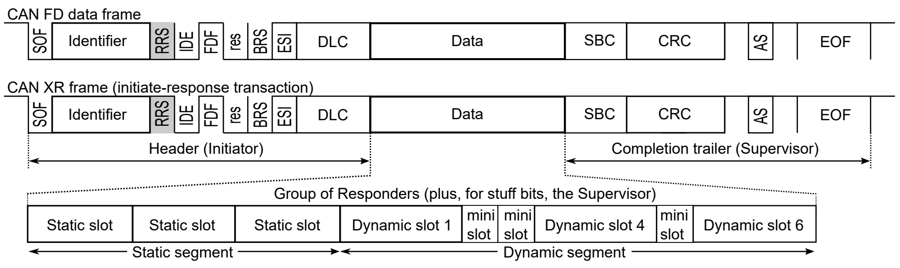

Fig. 1. Format of (ISO) CAN FD base frames with no bit-rate switch (above) and their usage to encode a sample XR transaction (below).

filling portions of the data field (*slots*) according to the rules given below, whereas the latter read in the relevant information in a similar way. Although initiators and responders are not conventional CAN nodes, to preserve full compatibility with CAN the bit sequence sent on the bus during a transaction has to be indistinguishable from CAN frames to all other nodes.

XR nodes discriminate between XR and non-XR frames by inspecting the identifier field: when it corresponds to specific, configurable patterns, atomic transactions take place. However, if XR extensions are applied to CAN FD, as in the case depicted in Fig. 1, protocol robustness can be increased if XR frames are additionally tagged corresponding to *slightly-off-specification* FD frames. In this way, misconfigured XR nodes are prevented from injecting on the bus undue responses and disrupting conventional data exchanges. As Fig. 1 shows, the Remote Request Substitution (RRS) bit can be used to distinguish between FD and XR frames. Only the base format is shown in the figure, but the same mechanism applies to the extended format as well. CAN FD does not define any specific "FD remote frame" and the RRS bit—located in the same position as the Remote Transmission Request (RTR) bit in classical CAN frames—must be sent dominant by transmitters. As happens to all reserved bits, its value is ignored by non-XR receivers.

In the following, RRS will be renamed Reply Request Select: it is set dominant in conventional FD frames, whereas a recessive value means that the frame is carrying an XR transaction. FD controllers lacking XR support see XR frames as FD data frames (off-specification is irrelevant for them in this case). Hence, they can decode the carried data in software and perform the role of consumers. Seemingly, using RRS to discriminate between FD and XR frames could help also in those cases where bus errors corrupt the identifier field and turn a conventional frame into an XR transaction (hence forcing responders to mistakenly reply, overwriting legal data). However, this kind of errors are not particularly worrying, as they are likely to trigger CAN error detection mechanisms.

To increase flexibility, tagging XR transactions based on the CAN FD format with the RRS bit should be optional and not mandatory. When RRS is exploited, using the same identifier for non-XR and XR messages is actually possible (though not advisable), provided that the Data Length Code (DLC) field—which specifies the size in bytes of the data field—is set to the same value. In case of arbitration clash, the FD frame will simply have precedence over XR because of the dominant RRS value. This is useful when the initiator (or any FD node) wants to mimic an XR transaction and enforce specific responses on its own. In fact, responders will not detect the transaction and refrain from replying. FD consumers, which are unable to distinguish between XR and FD frames, deal with both of them in the same way in software. Concerning XR consumers, they can be configured to decode specific FD frames in hardware according to XR rules.

B. Supervising Transactions

If more than one node is allowed to take part in the transmission of the frame part which follows the arbitration field, as in the case of XR transactions, some means must be defined to ensure that what is sent on the bus corresponds to a proper CAN frame. The node in charge of this task is referred to as the transaction *supervisor*. The supervisor is responsible for inserting stuff bits in the data field when required, so that CAN rules are never violated in the signal sent on the bus, not even in the case some responders do not reply. Moreover, the supervisor also finalizes the frame by dealing with the *completion trailer*, which includes the Cyclic Redundancy Check (CRC), Acknowledgment (ACK), and End of Frame (EOF) fields. In particular, it transmits the bit pattern corresponding to the CRC—that in CAN FD also encodes the stuff bit count (SBC)—immediately after the data field, then it checks the ACK slot (denoted as AS in Fig. 1) and EOF field, and deals with ACK and form errors, respectively. In computing the CRC value and determining values and positions of stuff bits, bit levels sensed on the bus have to be considered, since no node in the network can know the bit sequence corresponding to the entire transaction in advance. Carrying out operations in such a short time is not a problem. In fact, all existing CAN controllers are able to take decisions—including determining the value of the next bit to be sent—based on the bus level they sense at the previous sampling point.

In principle, any node could play the supervisor role, but reserving this to the initiator is by far the best choice. In fact, the transmission of the transaction's header implies that the related initiator is currently up and running. Relying on other nodes (e.g., a responder), which might be unavailable, could seriously undermine reliability. Concerning stuff bit insertion, responders that are in the process of transmitting their reply are also involved, besides the initiator.

Operations of the nodes involved in an XR transaction are sketched in the example in Fig. 2, which refers to a network that includes an initiator/supervisor and 4 responders (nodes A, B, C, and D). The reply of each responder is assumed to consist of exactly one byte. At any time during data field transmission, the active responder (if any) and the supervisor cooperate in inserting stuff bits (marked as "S" in the figure). When a responder does not reply (as in the case of node C), the supervisor prevents the bus from remaining stuck at recessive level for more than 5 bit times.

The bit monitoring mechanism must be enabled only when the node is actually transmitting. This happens in the header for the initiator, during the relevant reply for each responder, and when sending the completion trailer (ACK slot excluded) for the supervisor. In the latter, bit monitoring is also switched on whenever it is writing stuff bits in the data field.

In order to retain backward compatibility, error management for XR frames behaves exactly the same as CAN: as soon as any node (either XR or non-XR, and irrespective of its role) discovers an error, as per the CAN error detection mechanisms, it starts transmitting an error frame. This implies that all the responses included in the XR transaction are lost and have to be sent again. In theory, distinct responses could be checked and confirmed separately, but this would increase noticeably protocol complexity and overheads, lowering at the same time reliability since atomicity is lost. For this reason, we preferred to leave the basic XR protocol as simple as possible. It is worth pointing out that errors due to failures affecting the initiator/supervisor during a transaction are dealt with using the very same rules. By adopting the arrangements above, the original CAN robustness is not jeopardized, despite the producer role in CAN XR is distributed among a set of nodes.

C. Multiple and Implicit Initiators

The initiator of a transaction constitutes a single point of failure for CAN XR. To deal with this issue, the concept of "*multiple initiators*" can be exploited. Actually, a number of nodes can be configured as initiators for any given XR transaction. This is possible because the non-fixed part in the transaction header only includes the message identifier and DLC field. As long as all initiators select the same DLC value for XR messages with the same identifier, when two such nodes start transmitting at exactly the same time and their transmissions collide, the related bit streams will overlap and no error occurs. Of course, the cooperation of several nodes for triggering transactions improves communication reliability.

This approach resembles, in some way, backup time masters in TTCAN [16]. As in that case, initiators can be possibly configured so that the relevant identifiers differ in the least

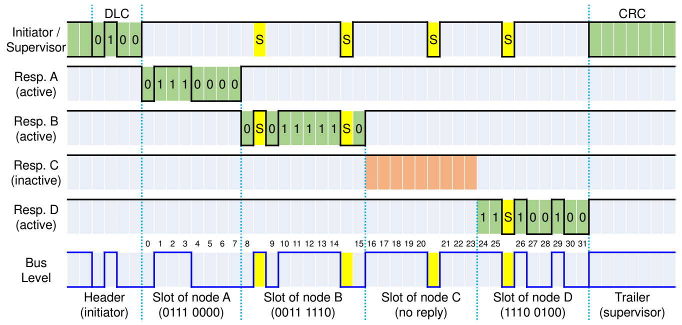

Fig. 2. Sample interaction among responders and supervisor.

significant bits. If so, suitable reception masks have to be set for message filtering on both responders and consumers.

Since the initiator can also act as a responder or a consumer in the same transaction it starts, the group of multiple initiators can be chosen as the set (or a subset) of the involved responders and consumers. In this case, they are referred to as "*implicit initiators*". As soon as any of such nodes starts a transaction (because, e.g., the value of its data has changed, an explicit request has been issued by an application program, or a timer has expired) the whole group of involved nodes is triggered and takes part to the exchange.

The set of data exchanged in this way is seen as a single entity on the network. Besides robustness, this improves data coherence across nodes. Moreover, there is no need to designate specific nodes to act as initiators only, so that costs can be reduced. When coupled with TTCAN, the overall behavior resembles FlexRay: XR transactions can be carried out inside exclusive windows, while non-XR frames are sent—possibly at higher bit rates in the FD case—in arbitrating windows.

III. Data Slotting

The data field in XR frames is given by a superposition of responses (sent by different nodes), which may possibly overlap. While disjoint replies are typically useful for gathering a number of (small-sized) data at once from responders, the aim of overlapping replies is to carry out special-purpose functions (such as distributed key generation). The latter feature can be suitably exploited by the upper protocol layers as a way to support extensibility. In theory, partially overlapping replies are also possible, but they are rather peculiar and their use will be investigated in future work.

Generally speaking, each response fits into a specific *slot* of the frame, and the data field can be seen as a sequence of slots. Several variants of the above approach can be devised, mainly depending on the amount of knowledge required by the initiator, responders, and consumers to carry out data exchanges. In CAN, the identifier field is enough to describe the payload format and meaning at both ends (producer and consumers) of any data transfer—besides the DLC field, which is required by the receiver MAC to determine the end of the frame. A similar approach can be adopted in CAN XR, but additional information is needed to identify specific slots in the frame.

To provide higher flexibility, two schemes are introduced in the following to deal with slotting, namely *static* and *dynamic*. In both cases, nodes not involved in a specific transaction do not need to be aware of slotting. They simply ignore the XR frame, but nevertheless take part in error detection and globalization, as in CAN.

A. Static Slotting

In this case, positions of responses are defined statically. Each responder must be configured separately, by specifying offset and size of its slot(s) in the data field of the relevant XR frame(s). The same holds for consumers. Importantly, responders and consumers are not required to know anything about slots they are not involved/interested in. No protocol control information is added at transmission time, as shown in Fig. 3a. This means that the *slot payload* takes the whole slot and there is no additional communication overhead. With this approach, other nodes can not determine whether or not any given responder is running and actually replying to the initiator, unless the slot pattern consisting of all recessive bits (not counting stuff bits) is reserved to this purpose to denote the lack of a response and not a proper value.

B. Dynamic Slotting

With dynamic slotting, responders and consumers are not required to be configured in advance with the starting position of their slots. Conversely, the *slot index* is used—together with the message identifier—to tag each slot uniquely and characterize the related information. Consumers are not even required (in theory) to know the slot size in advance. Instead, a *Slot Length Code* (SLC) field, encoded on 4 bits in a similar way as for DLC, is added to each reply by its sender in front of the payload, so that its boundaries can be discovered at transmission time by the other XR nodes (see Fig. 3b). SLC specifies the size (in bytes) of the slot payload. As for CAN, there is little point in having a bit granularity level for dynamic slots. This is not the case of static slots, where the slot size is only stored locally and not encoded in the frame.

Values of SLC in the range from 0000_2 to 1101_2 (0...13)—the upper bound is lowered to 0111_2 (7) for classical CAN frames—mean that the related responder is operating and actively taking part in the transaction. If so, the interested consumers read the slot into a local buffer, while responders and consumers involved in subsequent slots of the same transaction simply skip it. Thanks to SLC all of them can correctly advance to the beginning of the next slot.

Conversely, if SLC equals 1111_2 (15)—that is, the bus remains recessive for 4 bit times, excluding dominant stuff bits inserted by the supervisor (at most one, possibly at the end of SLC)—the responder is unavailable. This particular SLC pattern, which should not be used directly by responders, corresponds to a *minislot*. Its presence denotes that the slot is absent, and the next slot is expected to begin at the bit following SLC. Conceptually, a minislot is not the same as an *empty slot*, for which SLC is explicitly set to 0000_2 by the responder. Empty slots can be used when a responder purposely decides not to include anything in its reply (e.g., because no fresh data are available). Finally, the reserved value 1110_2 (14), termed *deferral notice*, is used to defer the actual response to the next relevant XR frame. As explained below, it is used when a responder is unable to include its reply in the current frame. As for minislots and empty slots, deferral notices do not have any associated payload.

Whether a regular/empty slot, a minislot, or a deferral notice is read (or written) on the bus, a suitable *slot counter* is increased by one in each CAN XR controller involved in the transaction. This counter identifies unambiguously the slot currently being received/sent in the XR frame, and is checked against the slot index assigned to the data to be exchanged. In the case they match, the related action (either transmission for responders or reception for consumers) is carried out. Such an approach resembles the Flexible Time Division Multiple Access (FTDMA) technique [17] (linear arbitration), adopted,

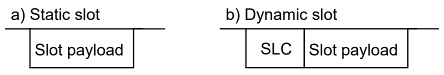

Fig. 3. Slot format (static and dynamic).

e.g., in the dynamic segment of FlexRay. Although dynamic slotting achieves higher flexibility, it is fairly more complex than static slotting. For this reason, its implementation should not be mandatory in CAN XR controllers.

Unlike static slots, where everything is decided at configuration time, a variable number of dynamic slots can fit in the data field of any given XR frame, mostly depending on how many responders are active and actually reply. Since the nominal size of each frame in CAN (as per the DLC value) is configured in advance, a responder is only allowed to reply if the room available in the remaining part of the data field (i.e., not used by the preceding slots) is large enough to contain its slot completely (both SLC and the whole payload)—it is worth noting that stuff bits have no effect on this check, as they are added after the frame has been assembled. On the contrary, a deferral notice is sent (when space permits) in its place. So that deferral notices can be included in the frame for all the dynamic slots envisaged by the transaction, their overall size (concerning the slots still to be sent) has to be accounted for when evaluating the remaining space.

Similarly to CAN, the lower the slot index, the higher is the chance that the slot will not be delayed. However, if a slot is too large to fit in the current frame, it may be overtaken by a (smaller) following one. By checking the presence of deferral notices, every other node (and, in particular, the initiator) can determine if there are responders that have data ready to send but have not been able to include them in the current frame. As for remote frames in CAN, it is completely up to initiators to decide if and when a further transaction has to be started for an XR frame that includes deferral notices.

Unlike static slotting, dynamic slotting does introduce communication overhead. In particular, 4 additional bits are required for every slot included in the frame (either regular ones, minislots, or deferral notices).

C. Hybrid Slotting

Mixing static and dynamic slotting in the same XR frame is possible, by configuring nodes so that they start decoding dynamic slots at a specific position of the data field. Conceptually, this means that the data field is thought of as split into a static and a dynamic segment, as in FlexRay. However, either segment is allowed to be empty in CAN XR.

In theory, multiple dynamic segments could be included in the same frame—possibly interleaved with static segments—but likely in this case the benefits would not outweigh the cost since this would make controller implementation fairly more complex.

IV. Service Definition

The internal architecture of CAN XR controllers, in terms of blocks devoted to response management, resembles TTCAN [16]. However, unlike TTCAN, where the elapsed time is used to determine transmission and reception windows inside basic cycles, CAN XR exchanges are driven by the detection of slots inside transactions. The frame filtering function—customarily implemented in hardware in the vast majority of CAN controllers to reduce the interrupt rate to the microcontroller—is mandatory in CAN XR, in order to quickly detect those frames which give rise to transactions. In fact, replies must occur in-frame, without disrupting the CAN frame format in any way.

A. Trigger-based Operations

As soon as a relevant XR message is detected by the frame filtering function of a CAN XR controller, data exchange is seamlessly started, which is controlled by specific triggers. In particular, *production triggers* (Prod_Trigger) and *consumption triggers* (Cons_Trigger) have to be defined on responders and consumers, respectively. Each trigger is characterized by the message identifier on which frame filtering is carried out (*message filter*). Only triggers whose filter matches the message currently being exchanged are *enabled*. As in conventional CAN controllers, a group of identifiers can be possibly specified by configuring a suitable register (*reception mask*). This could be useful in the case of multiple initiators, so as to provide receivers with an indication of the node who actually initiated the transaction.

The trigger is also linked to a *message object*, which provides the data structure for storing one slot (similar to what is required to hold a CAN frame). Message objects linked to production triggers contain data to be transmitted, whereas those used by consumption triggers are needed to get the content of the relevant slots.

More than one trigger (and hence, object) may exist in a CAN XR controller for the same message identifier, each one concerning a distinct slot in the frame. For this reason, besides the identifier of the related XR frame, each trigger is also characterized by either the absolute position of the slot in the frame (for static slots) or the slot index (for dynamic slots). To ease implementation, slots configured in the same controller are not allowed to overlap.

It is important to remark that only production triggers have to be implemented in hardware. Conversely, consumption triggers can be implemented in software on conventional CAN controllers. In this case, the entire data field (encoding the whole XR transaction) is read in, and it is up to the microcontroller singling out slots (both static and dynamic) and providing them separately to the application processes.

B. Triggers for Static Slots

Production and consumption triggers for static slots are defined by the following parameters:

- *Slot Offset* (SO): offset (in bits), from the beginning of the data field, where the slot is located. Each XR controller maintains a counter, known as *Bit Count* (BC), which is set to 0 at the beginning of the data field and is increased by one at every bit time (except for stuff bits). BC is checked against SO of every enabled trigger to determine the point in time when either a responder has to start sending the reply or a consumer has to start reading it. We refer to this condition by saying that the trigger has been *activated*. It is worth pointing out that several slots, belonging to distinct responders, are allowed to share the same starting position. This is required, e.g., by the key generation algorithm described in [7]. When XR frames are used to gather distinct data from different nodes, slots are not allowed to overlap and should be preferably placed one after the other, with no gaps in between.
- *Slot Size* (SS): nominal duration (in bits) of the slot payload. This enables a very fine granularity in allocating the space available in the data field—even smaller than one byte—and improves communication efficiency.
- *Slot Length Code* (SLC): payload size in the related message object, encoded according to the same rules used for DLC in CAN (FD). When stored in the controller's memory, the payload is aligned to byte boundaries.
- *Data Transmission Mode* (DTM): defines the way the slot payload is sent on the bus (either *exclusive*, *shared*, or *arbitrating*). For exclusive and shared slots, all payload bits have to be sent over the bus. Transmission in exclusive slots takes place according to the conventional rules used in CAN when dealing with the data field. They shall not overlap, otherwise bit monitoring errors may occur. Conversely, a dominant level sensed on the bus while a recessive bit is being sent in the payload of shared slots does not cause a bit monitoring error. In this way, a bit-wise AND function is carried out among overlapping responses. The third transmission mode is not dissimilar from shared slots, but resembles arbitration, i.e., a responder stops transmitting as soon as it senses a dominant level while sending a recessive bit. Arbitrating slots are useful to determine, within one transaction, the minimum among a set of values sent by different nodes.

C. Triggers for Dynamic Slots

To support dynamic slotting, one (or more) *dynamic segment triggers* (Dyn_Seg_Trigger) are required, which define where the dynamic segment is located in XR frames. Their operation is straightforward: a dynamic segment trigger is enabled when it matches the identifier of the frame being exchanged on the bus. To ensure correct operation, at most one of such triggers is allowed to be enabled in each node within the same transaction. Besides the message identifier, dynamic segment triggers are characterized by the following parameters:

- *Dynamic Segment Offset* (DSO): offset (in bits), from the beginning of the data field, of the first dynamic slot. It shall be set to the same value for all the dynamic segment triggers in the network associated to the same message identifier (or group of identifiers). The DSO parameter of an enabled trigger is checked against the bit count value BC to determine the beginning of the dynamic segment. When a match occurs, the trigger is activated and the controller starts scanning the incoming bit stream to single out dynamic slots.
- *Dynamic Segment Size* (DSS): nominal duration (in bits) of the dynamic segment. It shall not be larger than the part of data field located after DSO.

Production and consumption triggers for dynamic slots are not put into correspondence with message identifiers directly. Conversely, each of them is linked to a dynamic segment trigger and becomes enabled when the latter is activated. So that more than one dynamic slot in a given XR frame can be used by the same controller, several dynamic production or consumption triggers may refer to the same dynamic segment trigger. Such production/consumption triggers are defined by the following parameters:

- *Slot Index* (SI): relative position of the slot in the dynamic segment (1st, 2nd, 3rd, etc.). Each CAN XR controller maintains a counter, known as *Slot Count* (SC). Whenever a dynamic segment trigger is activated, SC is initialized to 0, and it is increased by one every time the beginning of a dynamic slot (either a regular one, a minislot, or a deferral notice) is subsequently discovered. Upon update, SC is checked against the SI entry in all the production and consumption triggers linked to the (unique) active dynamic segment trigger. If a match is found, the relevant operation (either write for responders or read for consumers) is carried out. Each controller must be configured in such a way that, at any time, no more than one of its dynamic production/consumption triggers can be active.
- *Slot Length Code* (SLC): size of the slot payload, encoded according to the CAN (FD) rules for DLC. Unlike static slots, patterns 1111_2 and 1110_2 are reserved for (received) minislots and deferral notices, respectively. Hence, the maximum payload size shrinks to 32 bytes. This is not a limiting choice, as dynamic slotting is mainly envisaged to collect small data packages.

D. Initiator and Supervisor Operations

In principle, no specific trigger has to be defined explicitly to support initiators' operation, unless they also perform roles of responders/consumers. However, some suitable way is needed for instructing the controller to start the transmission of an XR frame in place of a conventional CAN one, by either purposely specifying a new service primitive or extending an existing one. Upon invocation of the request primitive, the initiator controller starts sending the header and CAN arbitration is carried out. Since the initiator always acts as the supervisor, a confirmation primitive is issued on transaction completion. In all other involved nodes (responders and consumers), indication primitives are delivered to the upper layers.

To deal with XR frames that include a dynamic segment, initiators can optionally define specific objects that are linked to dynamic segment triggers: their aim is to detect dynamic slots and store the related transmission status. Information about every response type (regular slot, empty slot, minislot, or deferral notice) is captured and made available to the upper layers through a suitable data structure, organized as an array of status data. The presence of minislots denotes the unavailability of associated responders, whereas empty slots simply denote data unavailability. Instead, deferral notices mean that data are still stored in responders, and were not included in the dynamic segment due to lack of room. This condition can be possibly used to start a new transaction again.

V. Protocol Implementation and Assessment

In order to verify the practical feasibility of CAN XR and estimate its additional complexity with respect to a standard CAN controller, a proof-of-concept implementation was carried out and deployed as shown in the upper part of Fig. 4. The experimental setup consists of three embedded nodes based on an NXP LPC1768 [18], a popular low-cost microcontroller running at a core clock frequency of 100 MHz. Namely:

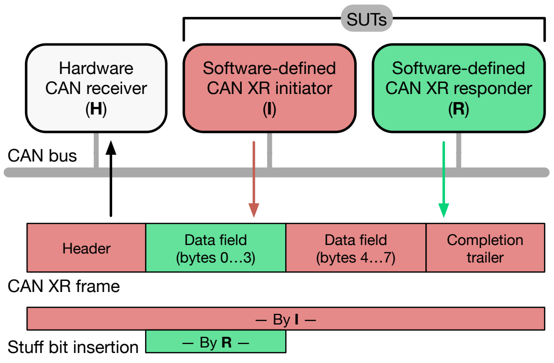

Fig. 4. Experimental setup for CAN XR implementation.

- Two nodes are the Systems Under Test (SUTs). They contain a *software-defined* CAN XR controller and play the role of initiator (I) and responder (R).
- A third node implements an ordinary *hardware-based* CAN receiver (H), based on one of the built-in CAN controllers available on the LPC1768.

All nodes are interconnected by means of a CAN bus that operates at 31.25 kb/s, the maximum speed software-defined controllers can operate at, due to processing power limitations better described in Section V-C.

In order to leverage readily-available hardware controllers for backward compatibility assessment, the implementation is based on classical CAN. However, this approach is more than adequate to prove the practical feasibility of the proposed method, especially for what concerns its most critical part, that is, the ability of supporting in-frame replies at the bus level. In fact, differences between CAN and CAN FD only concern the payload size, the format of the control and CRC fields, and CRC computation, none of which affects XR operations.

A manifest exception is that bit-rate switching and overclocking cannot be explored by means of classical CAN. However, this does not bring any limitations, as this feature is not compatible with XR. In principle, extending the implementation to CAN FD would not pose significant obstacles, as long as overclocking is not used, because frame timings are analogous. On the other hand, a proper support for overclocking must contemplate possible synchronization and timing tolerance issues during the data phase. A thorough investigation of these aspects, at both the protocol definition and implementation levels, has been foreseen as a future work.

A. Software-Defined CAN XR Controller

The software-defined CAN controller (SDCC) consists of several layered modules, organized as depicted in the upper right part of Fig. 5. Their structure and relationship are derived from the CAN specification [3] and closely reproduce it.

Unlike its hardware-based counterpart, which implements the CAN protocol as a whole in hardware and interfaces

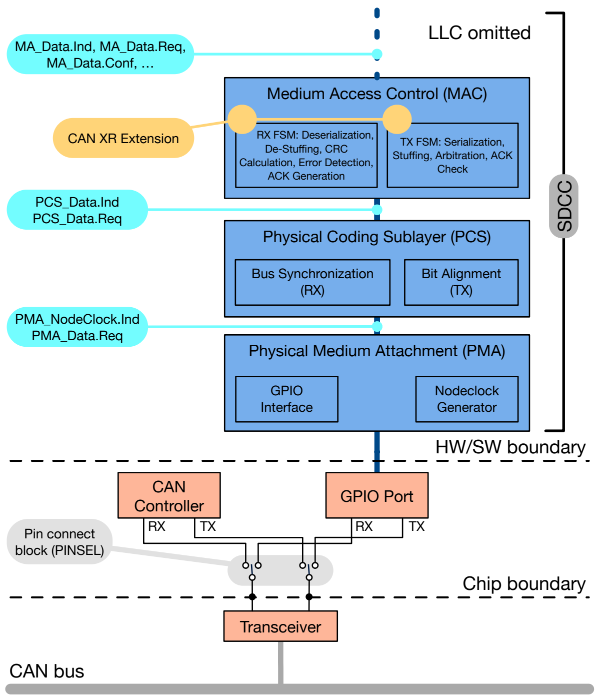
MA_Data.Ind, MA_Data.Req, MA_Data.Conf, ...
LLC omitted
CAN XR Extension
Medium Access Control (MAC)
RX FSM: Deserialization, De-Stuffing, CRC Calculation, Error Detection, ACK Generation
TX FSM: Serialization, Stuffing, Arbitration, ACK Check
PCS_Data.Ind
PCS_Data.Req
Physical Coding Sublayer (PCS)
Bus Synchronization (RX)
Bit Alignment (TX)
SDCC
PMA_NodeClock.Ind
PMA_Data.Req
Physical Medium Attachment (PMA)
GPIO Interface
Nodeclock Generator
HW/SW boundary
CAN Controller
GPIO Port
RX
TX
RX
TX
Pin connect block (PINSEL)
Chip boundary
Transceiver
CAN bus

Fig. 5. Architecture of a software-defined CAN XR controller.

directly with application-layer software, the only hardware components needed by SDCC are a General-Purpose Input-Output (GPIO) port and a transceiver. The first transforms the numeric information generated by SDCC into electrical signals that are then brought off chip and vice versa, while the second takes care of the low-level electrical interface to the CAN bus.

Integrating SDCC within a typical CAN-enabled microcontroller is generally easy, provided it makes use of an external transceiver. This is because (as shown in the lower part of Fig. 5) most microcontrollers are capable of routing different sets of internal signals to the same physical chip input and output pins. In this way the default connections to the on-chip, hardware-based CAN controller can conveniently be replaced by connections to one of the GPIO ports. On the LPC1768, this function is performed by the Pin Connect Block (PINSEL).

The SDCC layer closest to the hardware is the Physical Medium Attachment (PMA). Its two main purposes are to interface the software with the GPIO port registers—to allow it to interact with the transceiver—and generate a free-running node clock, which is used to retrieve the CAN bus level at every quantum and provide a timing reference to SDCC as a whole. This information is conveyed to the upper layer by means of a **PMA_NodeClock** indication. On the transmitting side, the **PMA_Data** request allows the upper layer to set the CAN bus to the level specified as argument.

Proceeding further up, we find the Physical Coding Sublayer (PCS). Its main responsibilities are to implement CAN bus synchronization through edge detection, sample the bus accordingly, and convey the sampled bit stream to the upper layer by means of **PCS_Data** indications. For what concerns the transmitting side, the PCS handles bit transmission requests coming from the upper layer through the **PCS_Data** request and ensures that bit transmission is properly aligned with respect to bus bit boundaries.

The layers presented up to this point are the same as in classical CAN. The CAN XR extension mainly affects the Medium Access Control (MAC) layer, to be discussed next. The software-defined implementation of the MAC layer revolves around two Finite State Machines (FSM) or automata.

The *receive* FSM is driven by **PCS_Data** indications and is divided into two nested sub-automata. The first one implements bus integration, SOF detection, and bit de-stuffing. Moreover, it performs bus monitoring (while the controller is transmitting), besides detecting stuff, bit, and ACK errors. The second sub-automaton operates on the de-stuffed data stream coming from the first. It implements de-serialization and recompiles the frame structure, while checking CRC and form errors. In addition, it transmits an ACK bit when appropriate. Provided a complete frame has been received successfully, it also generates a **MA_Data** indication.

The *transmit* FSM has the same internal structure as the receive FSM and is clocked by **PCS_Data** indications, too. The first sub-automaton coordinates with the receive FSM to honor **MA_Data** requests coming from the upper layer and start transmission on the bus when it is idle, besides performing bit stuffing on the data stream provided by the second sub-automaton. The second sub-automaton performs frame serialization and, in concert with the receive FSM, detects arbitration losses. At the end of a frame, it relies on the receive FSM to calculate the CRC to be transmitted and subsequently confirm that an ACK has been received, flagging an ACK error if this is not the case. Finally, it generates a **MA_Data** confirmation after any frame transmission.

For the sake of completeness, we must also mention that a full-fledged SDCC shall also include a Logical Link Control (LLC) layer, which implements programmable frame acceptance filtering, bus overload notification, and error recovery by means of automatic frame retransmission. However, these functions were not deemed necessary for the proof-of-concept implementation being described, and have been omitted.

B. *Frame Exchange Configuration*

Referring back to Fig. 4, SUTs I and R have been configured for a static slotting frame exchange pattern, as described in Section III-A. More specifically, I has been programmed to periodically initiate a CAN XR frame with a data field of 8 bytes. It is also responsible for sending the last four bytes of payload, as well as transmitting the frame header and completion trailer.

On the other hand, R responds to the frame sent by I with an in-frame reply that fills the first four bytes of payload and acknowledges it upon successful reception. Node I also plays the role of supervisor and inserts stuff bits where required within the whole frame. As shown in the lower part of Fig. 4, R inserts stuff bits only while it is transmitting on the bus.

Node H only receives frames, checks their correctness by means of the rules built in the hardware-based CAN controller, and acknowledges them as required by the CAN specification. Its purpose is to verify that CAN XR frames are completely backward compatible with CAN and that connecting CAN and CAN XR nodes to the same bus is not a cause of concern.

C. Experimental Results

Experiments performed on about 10000 frame exchanges with random payloads showed that all nodes (including H) are able to receive CAN XR frames correctly, thus confirming CAN XR's backward compatibility. In addition, disconnecting either H or R from the bus (but not both) does not hinder frame exchanges, thus showing that both CAN and CAN XR nodes are able to acknowledge a CAN XR frame correctly, and independently from each other. On the other hand, the disconnection of both H and R leads I to report acknowledgment errors, as expected.

Additional tests showed that stuff bit insertion operates correctly even when a stuff bit is required at the boundary between parts of frame I and R are responsible of, for instance, between the header and the first data byte, or between the fourth and fifth data bytes. Further insights on the correct behavior of CAN XR were gained by disconnecting R from the bus. In this case, the first four data bytes received by I and H consist of all recessive bits, because R no longer drives the bus within that portion of the frame. However, frames are still correct because I inserts stuff bits within them, as required, and H acknowledges them upon successful reception.

From the SDCC performance point of view, the current software version is able to reliably process up to 780,000 quanta per second, even though the CAN bus bit rate has been conservatively limited to 31.25 kb/s in the experiments just described. This is a remarkable result, considering that SDCC consists of about 2100 lines of C code and, at a core clock frequency of 100 MHz, this figure corresponds to about 128 clock cycles to process a quantum.

Another interesting information that can be derived from analyzing the code is that only about 150 out of 2100 lines of code were needed to extend SDCC and implement CAN XR. Even though additional effort is needed to support dynamic slotting, this confirms that existing CAN (and CAN FD) controllers can be extended to implement XR functionality without disrupting their structure significantly.

VI. Applications of CAN XR

CAN XR provides additional, very generic communication primitives to CAN, which can be exploited by the upper protocol layers to define new distributed services. For example:

- *Combined message*: XR transactions can be used to deal with master-slave distributed systems, and achieve increased communication efficiency when small-sized data packets (even less than one byte) are exchanged among devices. The initiator acts as the master, whereas responders and consumers carry out the roles of input and output slave devices, respectively. Non-overlapping *exclusive* static slots resemble logical addressing in EtherCAT (or the static segment in FlexRay). Dynamic slots, instead, mimic the dynamic segment of FlexRay.
- *Distributed key generation*: By using overlapping *shared* static slots, the distributed mechanism for generating symmetric keys in [7] can easily be implemented. The main advantage is that no custom solution has to be purposely defined in order to have the two nodes transmitting on the bus at the same time.
- *Min-Max discovery*: By using overlapping *arbitrating* static slots, the minimum among a set of values provided by responders can be quickly found. By logically complementing the involved values, the maximum can be found. It is worth reminding that what consumers see on the bus is a conventional CAN frame whose data field (or part of it) carries the minimum value, and no awareness of the XR Min-Max operation is needed for them.
- *Event notification*: By using static slots, mapped on single bits on responders, a multitude of devices are allowed to efficiently notify events. The wired-AND behavior of CAN could be possibly exploited to deal with events produced by multiple sources. The presence of a dominant value in any of these bits means that the related node (or at least one of them, in case of shared slots) has raised the corresponding event. To enable asynchronous event notifications, devices can be set as implicit initiators. Stuff bit insertion is carried out correctly, irrespective of the values enforced by devices, since it is based on bus levels and, besides responders, is backed by the supervisor.
- *Distributed consensus*: By using non-overlapping *exclusive* static slots and specifically exploiting atomicity of XR transactions, distributed consensus (e.g., majority voting) can be easily achieved among processes running on separate networked nodes. Each such node is configured as a responder for a specific slot and as a consumer for all the other slots. Because of the robust CAN error detection and globalization mechanisms, processes are ensured that they are agreeing and taking decisions on the same pattern of values.

By suitably combining the different options foreseen by CAN XR, other distributed services may be conceived as well.

A. Performance Comparison

A very interesting use of the CAN XR data slotting is collecting a number of process data, produced by distinct devices, into the same frame. In some circumstances this approach can achieve higher throughput than simply enabling bit-rate switching in CAN FD transmission. In Fig. 6, the overall time T (in bit times) taken to exchange a set of process data is shown for classical CAN, CAN FD, TTCAN, and CAN XR, by varying the number N of involved devices.

Each device is assumed to produce a single process datum, whose size D is set equal to 1, 2, and 4 bytes in the upper, middle, and lower plot, respectively. In the case of CAN FD, four sub-cases are taken into account, where the overclocking factor α (ratio between the bit rates in the data and arbitration phases) is set to $\times 1$, $\times 2$, $\times 4$, and $\times 8$, respectively. For TTCAN, all process data are assumed to fit exactly in the basic cycle and, for simplicity's sake, no safety margins are included in time windows. Thus, the duration of a reference message (Level 1, including only one data byte) was simply added to T. For CAN XR, a single combined message with static slotting is considered and the frame format (classical vs. FD) is chosen so as to minimize the wasted space. Not all sizes are allowed for the data field when DLC exceeds 8, which explains piecewise linear plots.

When process data are small ($D \leq 2$ bytes), CAN XR is always advantageous, provided that at least 3 producers are

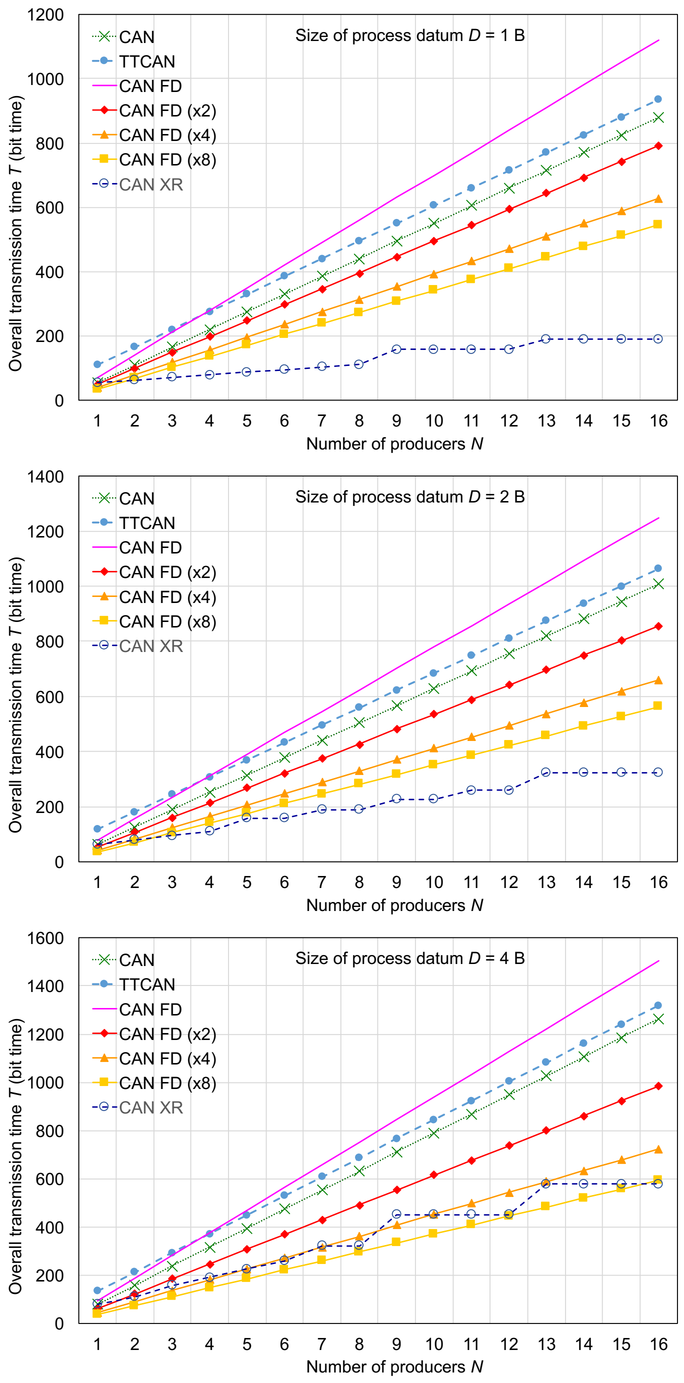

Fig. 6. Minimum transmission time T taken for exchanging all process data vs. number N of producing nodes and size of data $D = 1, 2,$ and 4 bytes.

involved in data exchanges. In the case of larger process data ($D \geq 4$ bytes), CAN FD overtakes CAN XR, but only when the bit rate in the data phase is increased tangibly ($\alpha > 4$).

As TTCAN is meant to improve determinism, its performance in our sample system is always slightly below CAN. Consequently, its throughput is not as good as CAN XR, unless overclocking is exploited and process data are large enough.

VII. Conclusion

In this paper an extension of CAN has been presented, called CAN XR, which augments the original protocol with *in-frame replies* in such a way that multiple nodes can include their data in the same frame. By exploiting the physical layer of CAN, which grants every node to observe the same level on the bus at any given time, complete backward compatibility is retained with existing CAN devices.

CAN XR can be used by applications and upper protocol layers to define a variety of new distributed services. Therefore, it has to be regarded as an *extensible* mechanism, and provides users with noticeably higher flexibility with respect to basic CAN transmission services. Several use cases have been described in the paper, which show how CAN XR can be profitably exploited in some practical situations.

This paper mainly focuses on the CAN XR protocol definition and provides some guidelines on the implementation of its services in real devices. Protocol feasibility and coexistence with existing CAN controllers have also been assessed, by using a purposely-developed experimental setup consisting of three nodes, two of which equipped with an XR-enabled software-defined CAN controller.

A thorough performance analysis, when data slotting is used to carry out efficient data transfers, as well as additional details on its use in real-world distributed applications, is planned for future works. Moreover, the option of bringing in some way overclocking in XR—overcoming the lack of the in-bit-time detection property during the data phase and the consequent synchronization and timing tolerance issues—will also be carefully investigated.

References

[1] F. Hartwich, "CAN with flexible data-rate," in *Proc. Intl. CAN Conference (iCC)*, Mar. 2012, pp. 14-1–14-9.

[2] G. Cena and A. Valenzano, "Overclocking of Controller Area Networks," *Electronics Letters*, vol. 35, no. 22, pp. 1923–1925, Oct. 1999.

[3] ISO, *ISO 11898-1:2015 – Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*, International Organization for Standardization, Dec. 2015.

[4] G. Cena, I. Cibrario Bertolotti, T. Hu, and A. Valenzano, "Improving compatibility between CAN FD and legacy CAN devices," in *Proc. of the 1st IEEE Intl. Forum on Research and Technologies for Society and Industry (RTSI)*, Sep. 2015, pp. 419–426.

[5] K. Lennartsson, "CAN FD filter for classical CAN controllers," in *Proc. of the Intl. CAN Conference (iCC)*, Oct. 2015, pp. 03-13–03-20.

[6] T. Adamson, "Hybridization of CAN and CAN FD networks," in *Proc. of the Intl. CAN Conference (iCC)*, Oct. 2015, pp. 03-9–03-12.

[7] A. Mueller and T. Lothspeich, "Plug-and-secure communication for CAN," in *Proc. of the Intl. CAN Conference (iCC)*, Oct. 2015, pp. 06-6–06-14.

[8] S. Woo, H. J. Jo, and D. H. Lee, "A practical wireless attack on the connected car and security protocol for in-vehicle CAN," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 993–1006, Apr. 2015.

[9] ISO, *ISO 11519-3:1994 – Road vehicles – Low-speed serial data communication – Part 3: Vehicle area network (VAN)*, International Organization for Standardization, Jun. 1994.

[10] G. Cena, I. Cibrario Bertolotti, T. Hu, and A. Valenzano, "CAN XR: CAN with eXtensible in-frame Reply," in *Proc. 14th IEEE Intl. Conference on Industrial Informatics (INDIN)*, Jul. 2016, pp. 1198–1201.

[11] ISO, *ISO 17458-1 – Road vehicles – FlexRay communications system – Part 1: General information and use case definition*, International Organization for Standardization, Jan. 2013.

[12] IEC, *Industrial Communication Networks – Fieldbus specifications – Part 3-12: Data-Link Layer Service Definition – Part 4-12: Data-link layer protocol specification – Type 12 elements*, Aug. 2014, ed. 2.0, IEC 61158-3/4-12.

[13] R. Schlesinger, A. Springer, and T. Sauter, "Automatic Packing Mechanism for Simplification of the Scheduling in Profinet IRT," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, pp. 1822–1831, Oct 2016.

[14] V.-G. Gaitan, N.-C. Gaitan, and I. Ungurean, “A flexible acquisition cycle for incompletely defined fieldbus protocols,” *ISA Transactions*, vol. 53, no. 3, pp. 776–786, May 2014.
[15] O. Garnatz and P. Decker, “CAN FD with dynamic multi-PDU-to-frame mapping,” in *Proc. of the Intl. CAN Conference (iCC)*, Oct. 2015, pp. 05-8–05-13.
[16] ISO, *ISO 11898-4 – Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication*, International Organization for Standardization, Aug. 2004.
[17] G. Cena and A. Valenzano, “On the properties of the flexible time division multiple access technique,” *IEEE Transactions on Industrial Informatics*, vol. 2, no. 2, pp. 86–94, May 2006.
[18] *LPC17XX User manual, UM10360 rev. 2*, NXP B.V., Aug. 2010.

Gianluca Cena (SM’09) received the Laurea degree in electronic engineering and the Ph.D. degree in information and system engineering from the Politecnico di Torino, Turin, Italy, in 1991 and 1996, respectively.

In 1995 he became an Assistant Professor with the Department of Computer Engineering, Politecnico di Torino. Since 2005 he has been a Director of Research with the Institute of Electronics, Computer and Telecommunication Engineering of the National Research Council of Italy (CNR–IEIIT), Turin. His research interests include wired and wireless industrial communication systems, real-time protocols, and automotive networks. In these areas he has coauthored more than 130 technical papers and one international patent.

Dr. Cena served as a Program Co-Chairman for the 2006 and 2008 editions of the IEEE International Workshop on Factory Communication Systems, and as a Track Co-Chairman in six editions of the IEEE International Conference on Emerging Technologies and Factory Automation. Since 2009 he has been an Associate Editor of the IEEE Transactions on Industrial Informatics.

Ivan Cibrario Bertolotti (M’06) received the Laurea degree *(summa cum laude)* in computer science from the University of Torino, Turin, Italy, in 1996.

Since then, he has been a Researcher with the National Research Council of Italy (CNR), Rome, Italy. Currently, he is with the Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Turin. He has taught several courses on real-time operating systems at Politecnico di Torino, Turin; has co-authored two books on the same topics; and serves as a Technical Referee for primary international journals and conferences. His research interests include real-time operating system design and implementation, industrial communication systems and protocols, as well as modeling languages and runtime support for cyber-physical systems. He received, as a coauthor, the Best Paper Award presented at the 8th IEEE Workshops on Factory Communication Systems (WFCS 2010).

Tingting Hu (M’11) received her master degree in Computer Engineering in 2010 and PhD degree with the best dissertation award in Computer and Control Engineering in 2015 both from Politecnico di Torino, Turin, Italy. Between 2010 and 2016, she also worked as a research fellow with the National Research Council of Italy (CNR), Turin, Italy. Since December 2016, she works as a post-doc researcher in the University of Luxembourg with the Faculty of Science, Technology and Communication.

Since the beginning of her research activities, her primary research interest concerns embedded systems design and implementation, spanning through topics such as real-time operating systems, communication protocols, formal verification of software modules and communication protocols, as well as security, with a special focus on the practical application of these concepts. Currently, she is devoting herself to the research on model driven engineering for safety-critical embedded systems. In the meantime, she serves as program committee member and technical referee for several primary conferences in her research area.

Adriano Valenzano (SM’09) received the Laurea degree in electronic engineering from Politecnico di Torino, Torino, Italy, in 1980.

He is Director of Research with the National Research Council of Italy (CNR). He is currently with the Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Torino, Italy, where he is responsible for research concerning distributed computer systems, local area networks, and communication protocols. He has coauthored approximately 200 refereed journal and conference papers in the area of computer engineering.

Dr. Valenzano is the recipient of the 2013 IEEE IES and ABB Lifetime Contribution to Factory Automation Award. He also received, as a coauthor, the Best Paper Award presented at the 5th, 8th and 13th IEEE Workshops on Factory Communication Systems (WFCS 2004, WFCS 2010 and WFCS 2017). He has served as a technical referee for several international journals and conferences, also taking part in the program committees of international events of primary importance. Since 2007, he has been serving as an Associate Editor for the IEEE Transactions on Industrial Informatics.