# On a Software-Defined CAN Controller for Embedded Systems

Gianluca Cena[a], Ivan Cibrario Bertolotti[a,*], Tingting Hu[b], Adriano Valenzano[a]

[a] *CNR – National Research Council of Italy, IEIIT, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy*
[b] *University of Luxembourg – Faculty of Science, Technology and Communication, 6 Avenue de la Fonte, L-4364 Esch-sur-Alzette, Luxembourg*

**Abstract**

Controller Area Network (CAN) technology is nowadays ubiquitous in vehicular applications and is also gaining popularity in other contexts, for instance, embedded and industrial automation systems. The recent standardization of CAN with flexible data rate (CAN FD), as well as other academic proposals, have highlighted the usefulness of enhancing the CAN physical and data link layers to attain better performance and other features. This paper describes a portable software-defined CAN controller called SDCC. Besides being handy as a research tool for experimenting with novel protocol concepts at the data link layer, SDCC is also fully capable of real-time execution. Hence, it can interact with real-world CAN devices through a physical bus interface.

*Keywords:* Controller area network (CAN), Industrial control, Real-time distributed systems

## 1. Introduction and Related Work

Since shortly after its conception in 1986 [1] and its first ISO standardization [2], Controller Area Network (CAN) has enjoyed an ever increasing popularity, not only within its intended application scenario—in-vehicle communication—but also elsewhere, for instance, in Cyber Physical Systems (CPS) such as real-time distributed industrial control systems [3]. After its initial evolution, culminating in 1995 with the introduction of CAN version 2.0B (supporting 29-bit extended identifiers) the CAN data link layer remained unchanged for years. One notable exception is time-triggered CAN (TTCAN) [4] that, as its name suggests, incorporates the time-triggered paradigm into CAN communication, although it did not enjoy widespread adoption.

Much more recently, the standardization of CAN with flexible data rate (CAN FD) [5] at the end of 2015, as well as other academic proposals, like CAN+ [6] and CAN with extensible in-frame reply (CAN XR) [7], have revamped the CAN protocol by improving its data transfer rate, maximum frame length, and flexibility, while also enabling it to support new services. At the same time, the latter research has shown the usefulness of working at the CAN data link layer to experiment with novel, advanced protocol features. However, experimental work on the data link layer of CAN (as for virtually any other protocol stack) has customarily been confined to the hardware domain, because the high performance required ruled out a software implementation.

An FPGA-based implementation approach is feasible in many cases, but requires specialized programming tools and a thorough knowledge of hardware description languages, such as VHDL [8], which CAN protocol designers and practitioners may or may not possess. Moreover, any prototype design can be deployed only on systems equipped with an on-board FPGA. Due to the relatively high cost of an FPGA with respect to a typical embedded microcontroller, this may severely limit the extent and complexity of the experimental test bed—for instance, in terms of the number of nodes.

Therefore, this paper presents a completely software-defined CAN controller called SDCC. Its design and implementation was inspired by the concept of software-defined radio (SDR) [9], in which software modules take the place of radio communication components traditionally realized in hardware. SDRs are nowadays popular both in literature and in practical applications for exactly the same reasons just mentioned. SDCC has been completely implemented in the C programming language [10], which most programmers are likely proficient with. Although it can be used in simulation mode and operate on a virtual CAN bus, unlike other software simulators formerly presented in literature [11], SDCC is also fully capable of real-time execution and can be connected to a real CAN bus. In this way, SDCC combines the flexibility and ease of use of a simulator with the ability to interact with other off-the-shelf or custom CAN devices through a physical bus, typical of the hardware-based approach. Since exactly the same software runs in both cases (with the exception of a thin and easy to check hardware adaptation layer), there is no validation gap between the two scenarios.

The SDCC approach also differs radically from other software products, like the Vehicle Network Toolbox (VNT) of MATLAB/Simulink [12]. In fact, VNT provides an extensive set of primitives to send, receive, log, and analyze CAN messages, but still relies on a standard, hardware CAN controller for bus communication. As a consequence, its users cannot deviate from the

*Corresponding author. CNR-IEIIT c/o Politecnico di Torino, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy. Tel.: +39 011 0905426, Fax: +39 011 0905429.

*Email addresses:* gianluca.cena@ieiit.cnr.it (Gianluca Cena), ivan.cibrario@ieiit.cnr.it (Ivan Cibrario Bertolotti), tingting.hu@uni.lu (Tingting Hu), adriano.valenzano@ieiit.cnr.it (Adriano Valenzano)

ISO-standardized data link layer protocol in any way. Similarly, but at a lower level of abstraction, SocketCAN [13] adds a user-accessible, socket-based CAN communication API to the Linux kernel. However, it does not support any simulation modes and, as VNT, relies on a hardware CAN controller for bus communication.

The viability of the SDCC concept has been confirmed by using it to evaluate enhancements to the CAN data link layer [7]. At the application layer, SDCC has been employed to implement existing proposals [14] and explore novel ways of supporting security protocols on CAN-based networks [15]. Besides research purposes, SDCC may prove to be advantageous also for deployment in real environments. For instance, the Local Interconnect Network (LIN), standardized as ISO 17987 [16], is customarily employed for in-vehicle body electronics at a maximum communication bit rate of 20 kb/s, due to its lower cost with respect to CAN. This is mainly due to the slower, simpler 2-wire connection and the software protocol implementation. If LIN and CAN protocol overheads are considered to be comparable, coupling SDCC with, for instance, SAE J2411 single-wire CAN (SWCAN) [17] at the same bit rate would probably lead to similar performance and saving as LIN, but with the remarkable advantage of being fully compliant to the CAN paradigm.

Industrial Internet of Things (IIoT) [18] is another scenario where SDCC could likely provide tangible benefits. As sketched in [19], it is possible to adopt CAN as the underlying transmission technology for wired sensor networks, which ensures industrial applications uninterruptible operation because of the external power supply. Moreover, several techniques already exist that map the Internet Protocol (IP) atop CAN [20, 21]. This means that most existing embedded devices (including those lacking an internal CAN controller) can be potentially turned into wired IIoT nodes easily and inexpensively, thanks to SDCC.

The paper is organized as follows: After a short introduction to the CAN protocol, given in Section 2, the paper describes the general principles behind the SDCC design in Section 3. Then, Section 4 provides more information on its implementation and the optimization techniques needed to achieve good real-time performance, even within the typical computing resource constraints of embedded systems. Section 5 focuses on SDCC real-time execution mode, that is, how its core supports a physical CAN bus interface. A different, simulation-oriented operating mode using mostly the same code base is presented in Section 6, while Section 7 is centered on SDCC experimental evaluation. Section 8 concludes the paper and hints at future work.

## 2. CAN Protocol Overview

The CAN protocol is specified by the ISO 11898 standard [5]. Its physical-layer medium is a broadcast bus that can assume two possible levels, *recessive* (the idle level, usually denoted as 1) and *dominant* (denoted as 0). Simultaneous transmission by multiple nodes is allowed. In this case the dominant level prevails on recessive, thus implementing a distributed wired-and among all transmitted values. The standard defines four possible frame formats at the data link layer. Two of them

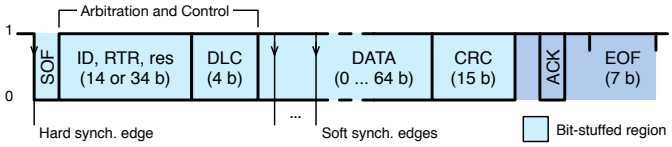



Figure 1: Classical CAN frame formats (stuff bits not shown).

(FBFF and FEFF formats) pertain to CAN FD and will not be described further for conciseness. The other two (CBFF and CEFF) are also called *classical* formats and are summarized in Figure 1. A classical frame consists of:

- The *start of frame* (SOF) dominant bit that delimits the beginning of the frame.
- The *arbitration* and *control* fields, their most important components being the message identifier *ID* and the data length code *DLC*.
- The *data* field, whose length is determined by the DLC value and can be between 0 and 8 bytes.
- The *cyclic redundancy check* (CRC) field, whose coverage spans from the SOF bit to the end of the data field and helps to assess frame integrity.
- The *acknowledgment* (ACK) bit, surrounded by two recessive delimiter bits, which is driven to dominant by receivers to confirm they successfully received the frame.
- The *end of frame* (EOF), a sequence of recessive bits that marks the end of the frame.

It must also be noted that the arbitration and control fields also include a remote transmission request (RTR) bit and a couple of reserved (res) bits, which are unimportant for this brief overview. The only differences between the two frame formats are the length of the message ID, 11 bits for the CBFF and 29 bits for the CEFF format, as well as the number and placement of reserved bits.

Although CAN nodes must confirm the bus is idle before attempting a transmission, it is still possible that multiple nodes start transmitting a frame simultaneously. In this case, the bus access conflict is resolved by means of a content-based distributed arbitration algorithm involving the arbitration field. This algorithm, leveraging the intrinsic bit-by-bit wired-and operation performed by the bus, ensures that the node with the lowest ID keeps transmitting, whereas all the others are informed they lost the conflict, are still able to receive the current frame, and will retry their transmission at a later time.

Since the bus does not have any dedicated clock line, CAN nodes have to recover the clock autonomously in order to receive individual bits correctly. They do so by means of a free-running local clock with a period $q = t/n$ called *quantum*, where $t$ is the bit time and $n$ is an integer that represents the number of quanta per bit. A quantum counter $c \in [0, n-1]$ is incremented according to the local clock and indicates the

current position within a bit. CAN nodes synchronize their notion of bit boundary by looking at the edges of the incoming bit stream and adjusting $c$ as needed. Due to the electrical characteristics of the bus, recessive → dominant edges are sharper than dominant → recessive edges, and hence, only the former are considered for synchronization.

As illustrated in Figure 1, the leading edge of the SOF bit marks the beginning of the CAN frame and is considered a *hard* synchronization point, that is, it unconditionally aligns $c$ to the bit boundary. Instead, all the other edges are *re-synchronization* points. In a re-synchronization, a configuration parameter called synchronization jump width (SJW), expressed in quanta, caps the maximum adjustment to $c$ to prevent wayward synchronizations due to noise spikes.

To ensure that the edges suitable for synchronization indeed occur frequently enough in the bit stream, the whole CAN frame except the part that follows the CRC field is subject to *bit stuffing*. According to this encoding technique, CAN nodes must append one extra stuff bit whenever they transmit 5 consecutive bits at the same level on the bus. Its level must be the complement of the preceding bits, thus enforcing the presence of (at least) one edge in the bit stream. Symmetrically, CAN nodes must also transparently remove stuff bits from the incoming bit stream while decoding the frame, a process known as *de-stuffing*.

Besides bit-level synchronization, CAN nodes also need to achieve proper frame-level synchronization. This cannot be done by looking at the SOF bit alone because, as outlined above, it is an ordinary dominant bit and cannot be distinguished from other dominant bits that appear elsewhere in the frame. For this reason, CAN nodes must in some cases wait until they detect a stream of at least 11 consecutive recessive bits, going through a process known as *bus integration*, before they start looking for SOF bits and start receiving/transmitting frames. One of those cases is initial startup, while others are related to error recovery. The value 11 corresponds to the recessive delimiter bit that follows the ACK, plus 7 recessive EOF bits, plus 3 recessive bits of intermission that separate CAN frames, not shown in Figure 1. This is also the minimum number of bits the bus must stay at the recessive level between consecutive frames according to the CAN standard, except when signaling an overload condition.

The CAN standard [5] specifies five different error detection measures for classical frames. More specifically, transmitters compare the value they are transmitting with the actual bus level, a mechanism known as bit monitoring. Any discrepancy leads to a bit error. They also detect an ACK error if the frame they transmitted has not been acknowledged properly by any receiver. Receivers check the incoming CRC and flag a CRC error if they find it incorrect. They also verify that the bit stuffing rules are fulfilled, with any violations leading to a stuff error. Finally, they confirm that the frame is well-formed, that is, some specific bits within the frame, like the delimiters, have the intended value. If this is not the case, a form error occurs. After detecting an error, CAN nodes are expected to globalize it, that is, make all the other nodes on the bus aware that an error occurred by transmitting an error flag. Then, transmitters should also re-transmit the frame. Further discussion of these mechanisms is however beyond the scope of this overview and readers are referred to [5] for further details.

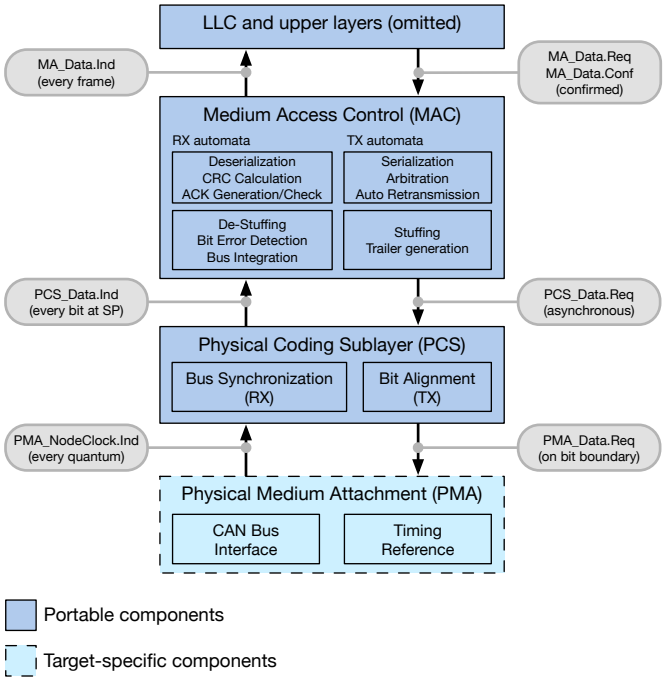



Figure 2: SDCC layers and their general structure.

## 3. General Design

As shown in Figure 2, SDCC has been designed according to the well-known layered approach. The internal structure of each layer, as well as their relationships, closely reproduce what is specified in the CAN standard [5]. More specifically, SDCC consists of three layers. Proceeding in a a top-down fashion, we find:

1. The medium access control (MAC) layer, which is responsible of frame-level synchronization and serializes/deserializes CAN frames according to Figure 1. Moreover, it implements the arbitration algorithm, takes care of bit stuffing and de-stuffing, and is in charge of error detection, globalization, and recovery.
2. The physical coding sub-layer (PCS), which implements bit-level synchronization and aligns transmitted bits with respect to the locally reconstructed bus clock.
3. The physical medium attachment (PMA), which connects SDCC to the physical or virtual CAN bus hardware and interfaces it with the local clock.

### *3.1. Medium Access Control (MAC)*

The MAC layer is the most complex of the three and contains 1185 lines of C code. It implements two pairs of nested finite state machines (FSM), or automata, one pair for the receive path, and the other pair for the transmit path.

The lower-level receive FSM is activated whenever the physical coding sub-layer (PCS) receives a bit from the bus, by means of the PCS_Data.Ind indication. It implements bus integration, outlined in Section 2, to ensure that SDCC synchronization at the CAN frame level is correct and performs start-of-frame (SOF) detection when bus integration is complete. In addition, it performs bus monitoring (while the controller itself is transmitting) and detects stuff and bit errors according to the definition given in Section 2. Most importantly, it also carries out bit de-stuffing. The higher-level receive FSM takes as input the bit stream coming from the first. Its main purpose is to de-serialize the incoming frame, then reconstruct and analyze its structure while detecting CRC and form errors. Upon the successful reception of a whole frame, this FSM is also responsible for acknowledging it, by transmitting an ACK bit, and forwarding the frame to the upper layers by triggering a MA_Data.Ind indication.

The two transmit FSMs are also clocked by PCS_Data.Ind indications, the sole timing reference of the MAC layer. The higher-level FSM serializes the frame to be transmitted, passed to the MAC layer by means of a MA_Data.Req request. Cooperating with the higher-level receive FSM, it also implements the CAN arbitration algorithm and stops transmitting when it detects an arbitration loss. Towards the end of frame transmission, it collaborates with the high-level receive FSM again to calculate the CRC to be transmitted and subsequently confirm that an ACK has been received correctly. If this is not the case, it flags an ACK error. If any error occurs during transmission, it applies the CAN automatic frame retransmission rules. Finally, it generates a MA_Data.Conf confirmation after any frame transmission. The lower-level transmit FSM coordinates with the lower-level receive FSM to determine when frame transmission can start, according to CAN rules. Moreover, it performs bit stuffing on the serialized data stream coming from the higher-level transmit FSM. As described in Section 2, bit stuffing is crucial to ensure that PCS bit-level bus synchronization, to be discussed in Section 3.2, can operate correctly.

### *3.2. Physical Coding Sub-layer (PCS)*

The PCS consists of 441 lines of C code. Along the receive path, its main role is to keep SDCC synchronized with the incoming CAN bus bit stream, by means of edge detection. The full details of the mechanism are specified in the CAN standard [5], but the general idea resembles a digital phase-locked loop. In summary:

- The PMA activates the PCS every quantum period $q$, by means of the PMA_NodeClock.Ind indication, providing the sampled bus level.
- The PCS performs edge detection on the incoming stream, by comparing adjacent samples. Barring noise spikes, these edges shall nominally occur on bit boundaries and serve as the base for synchronization. As explained in Section 2, only recessive → dominant edges are considered to this purpose.
- Various edge rejection techniques are applied to enhance robustness against noise. For instance, after detecting an edge, subsequent ones are neglected until a recessive bit is received, in order to avoid multiple synchronizations very close in time, which would likely be due to noise spikes.
- If an edge passes all rejection checks, the PCS determines the *phase error* $e$, defined as the number of quanta between the actual edge arrival and the nominal position of the bit boundary according to prior PCS knowledge.
- Last, the PCS adjusts its notion of bit boundary according to $e$. As recalled in Section 2, the SJW may cap the maximum adjustment depending on whether it is a hard synchronization or a re-synchronization.

Based on the knowledge of bit boundary position (derived from synchronization) and configuration information, the PCS is also responsible for sampling the bit value at the appropriate sampling point (SP) and forwarding it to the MAC by means of the PCS_Data.Ind indication. Being invoked once per bit, the same indication also provides a periodic timing reference to the MAC.

For the transmit path, the PCS handles bit transmission requests, submitted asynchronously by the MAC by means of the PCS_Data.Req request. Using again the synchronization information described previously, it ensures that bit transmission is properly aligned with respect to bus bit boundaries, by buffering PCS_Data.Req requests and issuing PMA_Data.Req requests at the appropriate time. It is worth noting that, although all SDCC layers depend on a common timing reference that originates in the PMA, PCS_Data.Req requests must still be considered asynchronous because the MAC invokes this primitive immediately after ascertaining what should be transmitted as the next bus bit. This typically happens shortly after the sampling point (SP) of the previous bit, with a variable delay that depends on the internal MAC processing time.

### *3.3. Physical Medium Attachment (PMA)*

The PMA is the layer closest to the CAN bus hardware, either real or simulated, and its implementation is therefore target-specific for the most part. The only portable PMA module consists of 93 lines of C code. It exports the target-independent layer application programming interface (API) and acts as an interface towards the other modules. The main PMA functions are:

1. Propagate the PCS layer data output, as indicated by means of the PMA_Data.Req primitive, through a bus transceiver and onto the CAN bus.
2. Sample the CAN bus every quantum period $q$ and provide its state to the PCS layer by means of the PMA_NodeClock.Ind primitive. Being strictly periodic, this primitive also furnishes a timing reference to the other SDCC layers.

In order to realize the second function, the PMA must also be connected to a suitable timing reference, which could be

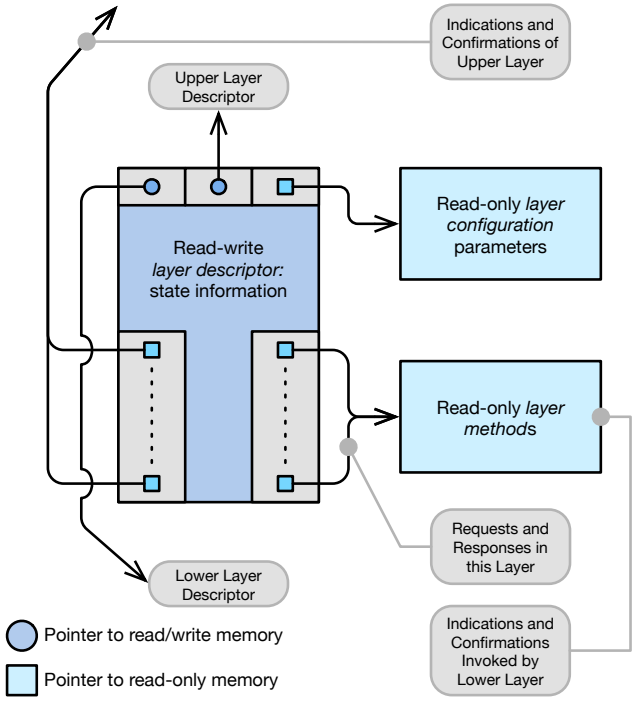



Figure 3: Internal structure of an SDCC layer and links between them.

a free-running hardware timer on a real-time target, or a periodic event source in an event-driven simulator. Further details about PMA implementation on a real-time target are given in Section 5, while Section 6 describes the role of the PMA in a simulation context.

## 4. Implementation Technique and Optimization

As stated in Section 1, SDCC has been implemented in C rather than in another, higher-level programming language to improve performance and reduce memory footprint. Both are especially important features when a piece of code is intended for real-time execution on an embedded platform. However, object-oriented (OO) design concepts have been used to preserve modularity, as well as ease of code reuse and enhancement, bearing in mind that one of the main SDCC purposes is to encourage experimentation with the CAN data link layer. Figure 3 provides more details about the internal structure of individual SDCC layers, which have been introduced and summarized in Figure 2. The centerpiece of each layer is a data structure called layer descriptor, which contains all the basic state variables the corresponding layer needs.

Table 1 describes the contents of the layer descriptors of the portable SDCC layers and is meant to complement the discussion of the same layers given in Section 3. Adjacent layer descriptors are bound together by a pair of pointers, to form a double-linked list (these pointers are shown at the top left of the descriptor in the figure). The layer descriptor contains a vector of pointers to methods, or functions, accessible from adjacent layers (sometimes called public methods). They are shown at the bottom right of the descriptor in Figure 3. These methods are responsible for implementing all requests and responses directed to the layer.

Moreover, other methods handle indications and confirmations originated by the lower layer. Methods of this kind are made known to the lower layer during protocol stack initialization through a registration procedure that establishes further links among adjacent layers. According to the OO programming paradigm, both kinds of method receive a pointer to the descriptor of the layer they belong to as first argument upon invocation. The number and type of the next arguments are method-specific instead. More specifically, the registration procedure fills a vector of methods, which is part of the descriptor of layer $x$, to point to the methods of layer $x + 1$ that are responsible for handling indications and confirmations originated by, and coming from, layer $x$ (this vector is shown at the bottom left of the descriptor in Figure 3). It is worth remarking that this implementation technique directly relates to the well-known upcall mechanism specified by the ISO Open Systems Interconnection (OSI) model [22].

The use of a vector of pointers to refer to methods indirectly, instead of using hardwired names, enhances layer modularity at a negligible performance penalty because, in this way, the public interface of a layer stays the same regardless of how its internal implementation may evolve with time. A further pointer within the layer descriptor (at the top right of the descriptor in the figure) leads to a data structure that contains (often constant) layer configuration parameters. Examples of parameters of the PCS layer, discussed in Section 3.2, are the SJW and the SP position.

Another advantage of using pointers to link together information pertaining to a certain layer instead of using a monolithic data structure is that, in this way, it becomes possible to mark part of the data as read-only, rather than read-write. Marking can be accomplished easily and without breaking compatibility with the C language standard [10] by using the `const` keyword sensibly. First of all, this enhances compiler optimizations, by improving its chance of correctly identifying immutable data. Secondly, when marking is combined with appropriate directives to the linker—usually embedded in a linker script [23]—it becomes possible to place code and the read-only part of these data structures in non-volatile (Flash) memory rather than RAM. This is especially important on embedded systems, in which RAM is often a precious resource.

On the other hand, it is worth remarking that, at the cost of a modest RAM footprint overhead, inter-layer pointers have been kept in RAM. In this way, SDCC supports the dynamic configuration of the protocol stack, allowing users not only to statically build the chain of layers of their choice, but also to change it at runtime.

Optimizations have been performed exclusively at the C language level, building on past experience with other performance-critical software [24] and prior knowledge of how compilers work internally. One of the goals was to show that it is possible to achieve good optimization results even across different processor architectures by working at this level, without sacrificing portability and conformance to language standards, and without using handwritten assembly code.

Table 1: SDCC Layer State Information Items (Portable Layers)

| *Layer/Item* | *Description* |
|---|---|
| *MAC* | |
| rx_fsm_state | Receive automata state |
| bus_integration_counter | Bit counter to measure the bus integration interval (11 consecutive recessive bits) |
| nc_bits, nc_pol | Number of consecutive bits at the same polarity, and their polarity, for bit (de)stuffing |
| crc | Cyclic Redundancy Check of the frame being received or transmitted |
| field_bits | Bit counter used to determine the end of multi-bit fields being received |
| bus_bits, de_stuffed_bits | Number of bits received from the bus and number of de-stuffed bits (for statistics and diagnostics) |
| rx_identifier | Identifier of the incoming frame |
| rx_rtr | Remote Transmission Request bit of the incoming frame |
| rx_ide | Identifier Extension bit, discriminates between classical CAN base (CBFF) and extended (CEFF) frames |
| rx_fdf | CAN FD Format indicator of the incoming frame (currently, SDCC does not handle FD frames) |
| rx_dlc | Incoming Data Length Code, used to determine the length of the data field and de-serialize accordingly |
| rx_byte | Temporary buffer to hold each data field byte as it is being de-serialized |
| rx_byte_index | Index of the current byte (held in rx_byte) within the data field |
| rx_data | Data field of the incoming frame |
| tx_fsm_state | Transmit automata state |
| data_req_pending | Indicates whether a frame is waiting to be transmitted or not |
| tx_identifier | Identifier of the frame to be transmitted |
| tx_format | Format of the frame to be transmitted (CBFF or CEFF) |
| tx_dlc | Data Length Code of the frame to be transmitted |
| tx_data | Data field of the frame to be transmitted |
| tx_byte_index | Index of the byte in tx_data currently being serialized |
| tx_bit_count | Bit counter used while transmitting multi-bit fields |
| tx_shift_reg | Shift register to serialize multi-bit fields while transmitting |
| *PCS* | |
| nodeclock_ts | Node clock timestamp counter for debugging and data logging |
| prescaler_m_cnt | Counter for the node clock prescaler |
| quantum_m_cnt | Quantum counter $c$ within a bit, subject to hard synchronization and re-synchronization |
| quanta_per_bit | Total number of quanta per bit $n$, derived from configuration parameters |
| prev_bus_level | Bus value sampled at the previous quantum, for edge detection |
| prev_sample | Bit value at the previous sampling point, for dominant → recessive edge rejection |
| sending_level | Level currently being sent on the bus, for positive-$e$ edge rejection while sending recessive |
| sync_inhibit | Synchronization inhibit flag, to avoid multiple synchronizations until sampling a recessive bus |
| hard_sync_allowed | Flag to allow or forbid hard synchronization, set by MAC based on the current position within the frame |
| output_unit_buf | Buffer to store the value to be transmitted while aligning it with bit boundaries |

## 5. Real-Time, Physical Bus Interface

As discussed in Section 3, only a small part of the PMA layer is portable because it is responsible for interfacing SDCC with real or simulated hardware components. In this section, we describe a PMA that enables SDCC to operate in real-time on a physical CAN bus. A different PMA, which supports the SDCC simulation mode instead, will be presented in Section 6.

The connection between the PMA and the physical CAN bus typically takes place through a general-purpose I/O (GPIO) port connected to a CAN transceiver external to the microcontroller chip. These connections are shown on the left of Figure 4. The role of the GPIO port is to convert the numeric bus level generated along the SDCC transmit path into an electrical signal that is then brought off chip, and vice versa along the receive path. On the other hand, the transceiver takes care of all the details concerning the electrical interface to the CAN bus.

Connecting the GPIO port to the transceiver is generally easy on a CAN-enabled microcontroller because, as shown in the figure, most of them are capable of routing multiple sets of internal signals to the same input and output pins. For instance, the NXP LPC17xx family of microcontrollers [25] embeds a pin connect block (PINSEL) to this purpose. As shown in Figure 4, the pins to which the transceiver is electrically connected are ordinarily routed to the hardware CAN controller but, by re-programming the PINSEL logic, it is possible to route the same pins to the GPIO port.

The other important component the PMA must interact with is a free-running timer, shown on the right of the figure. It is used to periodically sample the bus level and generate the node clock, that is, the stream of PMA_NodeClock.Ind indications that makes up the timing reference for SDCC as a whole, as described in Section 3.3. On the LPC17xx, any of the four 32-bit, programmable timers (TIMER0...3) has an adequate operating frequency and resolution, and is suited for use.

As an example, Figure 5 shows how this core PMA func-

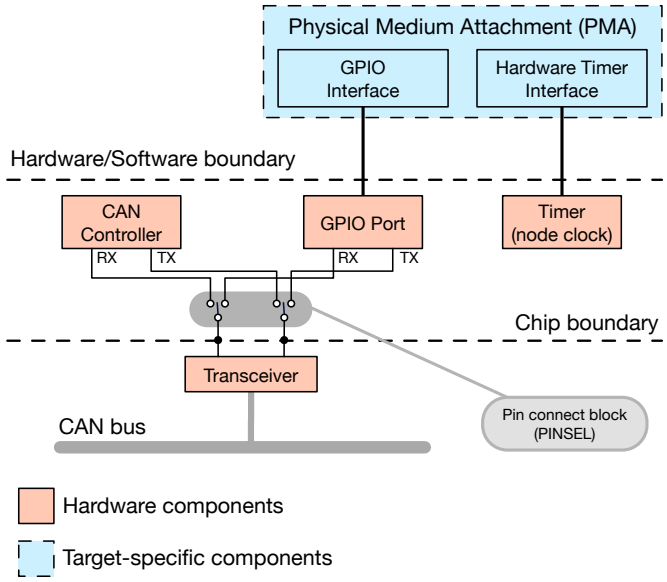



Figure 4: SDCC real-time interface towards a physical CAN bus.

```
x = read_ts();
while(1)
{
  /* Synchronize with TIMER0, the node clock */
  while(x == read_ts());

  /* Sample bus level, generate a node clock ind. */
  if(pma->primitives.nodeclock_ind)
    pma->primitives.nodeclock_ind(pma->pcs,
                                  gpio_rx_pin());

  /* Prepare to wait for next quantum */
  x++;

  /* Simple cycle overflow check */
  if(x == read_ts())
    LED_ON(GREEN);
  else
    LED_OFF(GREEN);
}
```

Figure 5: Core of the PMA towards a hardware CAN transceiver and timer.

tion has been implemented. For simplicity, the implementation follows a polling-based approach, but interrupt-driven methods are also feasible. The outer, infinite loop starts executing after SDCC initialization. Immediately before entering the outer loop, variable `x` is set to the current value of timer TIMER0, returned by the function `read_ts`. By means of initialization code not shown in the figure, this timer has been programmed to run at the desired node clock frequency. The inner loop traps the processor in an active wait until the current value of TIMER0 changes. In other words, this loop synchronizes the PMA with timer value transitions that, as stated previously, occur once every node clock period. Upon exiting from the inner loop, the code samples the current bus level provided by the transceiver, through the corresponding GPIO pin, by means of the function `gpio_rx_pin`. Then, it forwards the value to the PCS by invoking the method registered (by the PCS itself) in the `primitives.nodeclock_ind` element of the method vector stored within the PMA layer descriptor `pma`.

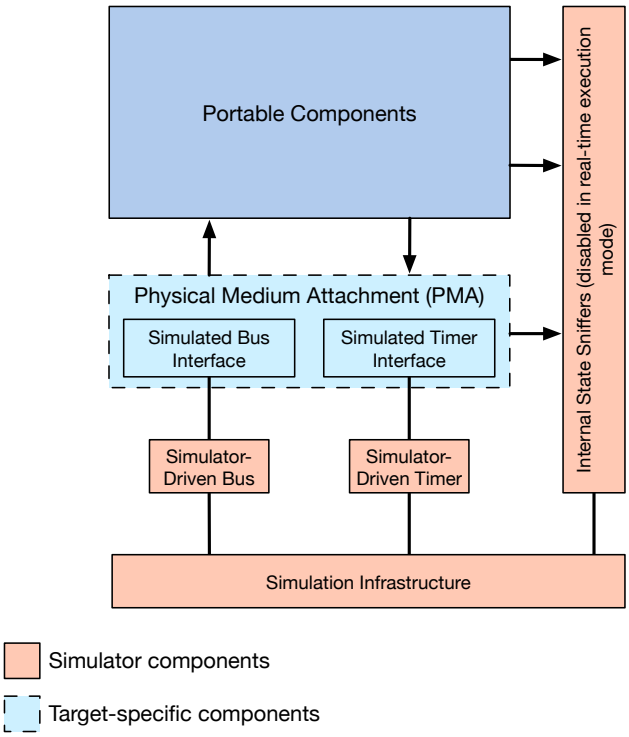


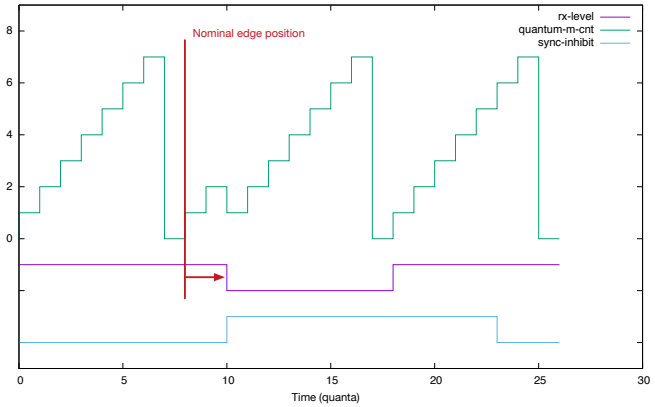

Figure 6: SDCC in simulation mode.

As explained in Section 4, the method receives a pointer to its own layer descriptor (the PCS in this case) as first argument, and the bus level as second argument. The PCS layer descriptor can be conveniently and efficiently accessed from the PMA layer descriptor by following the upward pointer shown in Figure 3, that is, `pma->pcs`. After the method returns, the PMA prepares for the next timer synchronization by incrementing `x`, which represents its own notion of node clock number. The availability of this information provides a handy opportunity to detect a schedule overflow, that is, a method invocation that takes more than one node clock period to return.

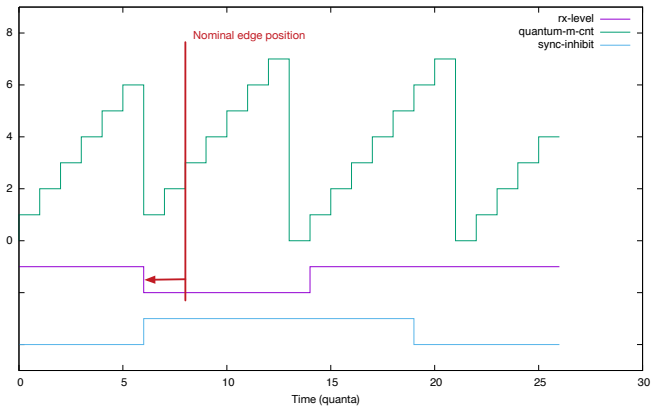
Any overflow is a critical error and must be flagged because it implies that SDCC was unable to meet its real-time execution constraints. In this case, the code compares `x` with the timer's notion of current node clock number, provided by a fresh invocation of `read_ts`. If these two values are the same, the PMA is "in time," that is, the method returned before the beginning of the next node clock period, otherwise an overflow just occurred. The result of the comparison is used to drive another GPIO output by means of the functions `LED_ON` and `LED_OFF`, so called because the output is physically connected to a LED on the evaluation board used as a test bed. This output can then be checked visually or, more effectively, with an oscilloscope: Any spike implies that the PMA lost synchronization with the hardware timer for at least one node clock period.

## 6. Simulation Mode

Besides being executed in real-time mode on an embedded system, as described in Section 5, the same SDCC portable lay-




*a) positive phase error, $e = 2$*




*b) negative phase error, $e = -2$*

Figure 7: Example of SDCC internal state observation (PCS layer).

ers can also be compiled for a personal computer and work in simulation mode with the help of a different PMA, shown in Figure 6. This feature is convenient, for instance, during the early stages of development of a new protocol idea, because it offers significantly better observation and debugging aids than real-time execution without sacrificing fidelity, because mostly the same code base runs in both scenarios. Timing fidelity is mostly preserved, too, because SDCC execution is still accurate down to the node clock period within the simulated time line. The only timing-related aspect that is not checked in simulation mode with respect to real-time mode is the SDCC execution time overflow described at the end of Section 5.

The simulation-mode PMA connects SDCC to a simulated CAN bus, which can be driven by the simulator according to a programmable stimuli file. Moreover, it interfaces with a simulated timer to generate `PMA_NodeClock.Ind` indications when triggered by the simulator. In addition, all SDCC layers, both portable and non-portable, support "sniffers" that work on layer descriptors and make part of their state variables available to the simulator. The simulation infrastructure then simulates the passage of time, gathers all this state information, and finally makes it available in a file suitable to be given as input to `gnuplot` [26] or other post-processing tools for further analysis.

Figure 7 shows an example of observation of some SDCC state variables taken from the PCS and PMA layers and pertaining to the synchronization process discussed in Section 3.2. The state variables being observed are the quantum counter $c$ (denoted as `quantum_m_cnt` in the figure) and `sync_inhibit` (also discussed in Table 1) from the PCS, and `rx_level` from the simulated PMA. The latter simply represents the current CAN bus level received from the transceiver.

For this experiment, the PCS has been configured with `quanta_per_bit=8`, that is, it considers that a bit period is nominally made of 8 quanta, numbered from 0 to 7. Accordingly, `quantum_m_cnt` also counts from 0 to 7, included, the SP has been set between the 5th and the 6th quantum of a bit, and the nominal edge position is considered to be at the transition between the 0th and the 1st quantum.

The upper part of the figure (Figure 7a) shows the PCS behavior when the incoming edge is "late" with respect to the nominal position, that is, it has a positive phase error $e = 2$. More specifically, it shows how `quantum_m_cnt` is decremented by 2 upon edge detection, thus updating the PCS notion of where bit boundaries are with respect to the incoming bit stream. Instead, the lower part of the figure (Figure 7b) shows how the PCS adjusts `quantum_m_cnt` when the incoming edge is "early," that is, it has a negative phase error $e = -2$. In this case, at edge arrival, the PCS assumes a new bit has begun and, to compensate the phase error, unconditionally sets `quantum_m_cnt` to the value 1. In both examples, we assume that the SJW has been set to a value $\geq 2$, so that it does not cap the `quantum_m_cnt` adjustment in any way.

The same figure also demonstrates that the PCS correctly sets `sync_inhibit` when it performs a synchronization and, as required by the CAN specification, resets it only when it samples a recessive bit. In both examples, this happens at the SP of the next bit. As a consequence, further synchronizations, possibly due to noise spikes, are inhibited in the meantime.

## 7. Experimental Evaluation

The characteristics of SDCC have been experimentally evaluated in three different areas: memory footprint, real-time execution performance, and compatibility with commercial CAN controllers.

### *7.1. Memory Footprint*

In order to determine its memory footprint, SDCC has been compiled for two dissimilar architectures.

1. On the LPC1768 microcontroller, with the real-time PMA described in Section 5, using `gcc` version 4.9.3 as cross-compiler.
2. On an Intel T9400 processor, with the simulated PMA presented in Section 6 and simulation mode enabled, using the native Apple C compiler `clang` version 900.0.39.2.

The first scenario is deemed to be representative of a typical embedded system, while the second constitutes a common example of personal computing environment. The main results

Table 2: SDCC Memory Footprint, in bytes

| *Layer* | *LPC1768* | | | *T9400* |
|---|---|---|---|---|
| | *Text* | *Conf.* | *BSS* | *Total* |
| Medium Access Control (MAC) | 1728 | 0 | 136 | 10920 |
| Phys. Coding Sub-Layer (PCS) | 376 | 24 | 80 | 3232 |
| Phys. Medium Attachment (PMA) | 340 | 0 | 20 | 1156 |
| *Total* | *2444* | *24* | *236* | *15308* |

are shown in Table 2. For the LPC1768, the memory footprint has been divided into three categories:

- text section, containing code and other read-only data,
- layer configuration data, typically read-only, and
- read-write, uninitialized data, traditionally called BSS.

This distinction is important because, as explained in Section 4, items belonging to the first two categories can be allocated in Flash memory instead of RAM. On the other hand, only RAM memory is accessible on the T9400 system, given that the processor is managed by a general-purpose operating system. Therefore, only total sizes have been shown.

In both cases, memory footprint has been determined by means of the `size` command, executed on individual object modules, plus manual inspection of the linker map files to achieve a finer level of granularity and determine the size of layer configuration parameters. The first observation that can be drawn from the experimental data shown in the table is the significant difference between the footprint of SDCC on the two targets, amounting to a factor of about 5. There are several main reasons for this:

- When compiled in simulation mode, as is the case for the T9400, SDCC layers contain extra code to support the sniffers mentioned in Section 6, as well as other debugging aids, for instance, a facility to trace main SDCC activities and generate a log file.
- The LPC1768 embeds an *ARM Cortex-M3* processor core, with a 32-bit architecture, whereas the T9400 belongs to the Intel Core 2 Duo family and implements the *Intel 64* 64-bit architecture. As a consequence, several commonly-used data types (for instance, pointers) are twice as big on the T9400, with respect to the LPC1768.
- Cortex-M3 instructions are either 16 or 32 bits in size, whereas Intel 64 instructions vary in length from 1 to 15 bytes. Moreover, the two compilers have different optimization goals on the two architectures, that is, they tend to optimize code size on the Cortex-M3 and execution speed on the T9400.

On the other hand, footprint measurements confirm the relatively small size of SDCC when cross-compiled in real-time execution mode (less than 3 KB of memory in total, most of it read-only), and hence, its suitability even for low-cost embedded systems with limited memory resources.

### *7.2. Real-time Execution Performance*

Real-time performance assessment has been performed on the LPC1768 microcontroller, configuring the processor to run at the maximum clock speed, that is, 100 MHz. As a first approximation, the code shown in Figure 5 has been used to detect schedule overflows while SDCC was running at several standard bit rates. The check was done with SDCC connected to a physical bus, which other CAN nodes saturated by issuing back-to-back frame transmission requests. In this way it has been determined that SDCC starts exhibiting sporadic schedule overflows when the bit rate is brought from 50 to 100 kb/s. Then, to double-check and refine the evaluation, the total number of clock cycles that SDCC requires to handle a `PMA_NodeClock.Ind` indication has been measured, by means of cycle-accurate timestamps. As described in Section 3, this entails the execution of the whole PCS layer and all the four FSMs within the MAC layer.

Experimental results indicate that, in the worst case, SDCC can completely process a `PMA_NodeClock.Ind` in 130 clock cycles, fast enough to sustain communication on a CAN bus running at 62.5 kb/s when allocating 100% of the core cycles to this task. This is a remarkable result, above all if we consider that, by present-day standards, the LPC1768 is placed at the low end of the spectrum of microcontrollers used in typical embedded applications. On the other hand, it also reveals that core utilization is the main limiting factor of SDCC in actual applications, given that the microcontroller has to execute other tasks as well, because it grows linearly with the bit rate.

Instead, as shown in Section 7.1, memory footprint is modest and does not depend on it. When higher bit rates are required, the only ways out of this limitation are to further optimize SDCC code, or use a microcontroller with a higher clock frequency and a more powerful instruction set. On the contrary, if SDCC is seen as a replacement of LIN, as proposed in Section 1, we can assume it would also operate at the maximum LIN bit rate, that is, 20 kb/s. In this case, even on a low-end LPC1768, SDCC is going to consume only about 32% of core cycles, thus leaving a significant share of computing power to other tasks.

### *7.3. Compatibility with Commercial Controllers*

While working on CAN protocol enhancements, both at the data link and application layers [7, 15], SDCC has been experimentally confirmed to be compatible with several commercial CAN controllers dissimilar from each other, that is, the ones embedded in the NXP LPC1768, LPC2468 and LPC4357, as well as the Freescale MCF5282 (now also acquired by NXP). Namely, more than 10 million frames have been successfully exchanged among multiple instances of SDCC and the above-mentioned commercial controllers without detecting any problems. Although a thorough evaluation of conformance to the CAN specification, by means of the relevant ISO standard [27], would be necessary to prove the interoperability between SDCC and other CAN implementations, those results still provide some evidence that no major issues are likely to exist.

## 8. Conclusion

This paper described SDCC, a software-defined CAN controller that operates in real time and can be connected to a physical CAN bus. When executed on a Cortex-M3 processor running at a clock speed of 100 MHz, it supports bus bit rates of up to 62.5 kb/s. The same code base can also work in simulation mode on an ordinary PC for initial protocol development, testing and debugging. In this way, it provides a valuable, flexible, and inexpensive tool to experiment with novel ideas concerning the CAN data link layer protocol. As an example, the prototype implementation of a CAN-based security protocol [15] on top of SDCC required only about 280 additional lines of C code, thus shortening protocol development time and freeing valuable resources within the research team.

## References


[1] U. Kiencke, S. Dais, M. Litschel, Automotive serial controller area network, in: Proc. SAE International Congress and Exposition, SAE International, Reading, UK, 1986, pp. 1–8.

[2] ISO, ISO 11898-1:1993 – Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling, International Organization for Standardization (Nov. 1993).

[3] CiA, CiA 301 V4.2.0 – CANopen application layer and communication profile, CAN in Automation e.V. (Feb. 2011).

[4] ISO, ISO 11898-4 – Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication, International Organization for Standardization (Aug. 2004).

[5] 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).

[6] T. Ziermann, S. Wildermann, J. Teich, CAN+: A new backward-compatible controller area network (CAN) protocol with up to 16x higher data rates, in: Proc. Design, Automation & Test in Europe Conference Exhibition (DATE), European Design and Automation Association, Leuven, Belgium, 2009, pp. 1088–1093.

[7] G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano, CAN with extensible in-frame reply: Protocol definition and prototype implementation, IEEE Transactions on Industrial Informatics 13 (5) (2017) 2436–2446. doi:10.1109/TII.2017.2714183.

[8] Robert Bosch GmbH, VHDL reference CAN, Available online, at `http://www.bosch-semiconductors.com/ip-modules/can-ip-modules/vhdl-reference-can/` (2018).

[9] T. Ulversoy, Software defined radio: Challenges and opportunities, IEEE Communications Surveys Tutorials 12 (4) (2010) 531–550. doi:10.1109/SURV.2010.032910.00019.

[10] International Organization for Standardization and International Electrotechnical Commission, ISO/IEC 9899, Programming Languages — C, 3rd Edition (Dec. 2011).

[11] P. Mundhenk, A. Mrowca, S. Steinhorst, M. Lukasiewycz, S. A. Fahmy, S. Chakraborty, Open source model and simulator for real-time performance analysis of automotive network security, SIGBED Rev. 13 (3) (2016) 8–13. doi:10.1145/2983185.2983186.

[12] The MathWorks, Inc., Vehicle Network Toolbox, Available online, at `https://it.mathworks.com/products/vehicle-network.html` (2018).

[13] M. Kleine-Budde, SocketCAN — the official CAN API of the Linux kernel, in: Proc. Intl. CAN Conference (iCC), 2012, pp. 5-17–5-22.

[14] A. Mueller, T. Lothspeich, Plug-and-secure communication for CAN, in: Proc. of the Intl. CAN Conference (iCC), CAN in Automation, Nürnberg, Germany, 2015, pp. 06-6–06-14.

[15] G. Bloom, G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano, Supporting security protocols on CAN-based networks, in: Proc. 18th IEEE International Conference on Industrial Technology (ICIT), IEEE, Piscataway, NJ, 2017, pp. 1334–1339.

[16] ISO, ISO 17987-1:2016 – Road vehicles – Local Interconnect Network (LIN) – Part 1: General information and use case definition, International Organization for Standardization (Aug. 2016).

[17] SAE, SAE J2411 – Single Wire Can Network for Vehicle Applications, SAE International (Feb. 2000).

[18] L. D. Xu, W. He, S. Li, Internet of things in industries: A survey, IEEE Transactions on Industrial Informatics 10 (4) (2014) 2233–2243. doi:10.1109/TII.2014.2300753.

[19] G. Bloom, G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano, Optimized event notification in CAN through in-frame replies and Bloom filters, in: Proc. 13th IEEE International Workshop on Factory Communication Systems (WFCS), 2017, pp. 1–10. doi:10.1109/WFCS.2017.7991963.

[20] P. Cach, P. Fiedler, Internet draft – IP over CAN, Available online, at `http://mirror.physik-pool.tu-berlin.de/pub/ietf/ietf-tools.html/draft-cafi-can-ip-00.html`, expires: Sept. 2001 (Mar. 2001).

[21] G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano, Seamless integration of CAN in intranets, Computer Standards & Interfaces 46 (2016) 1–14. doi:http://dx.doi.org/10.1016/j.csi.2015.11.004.

[22] ISO/IEC, ISO/IEC 7498-1:1994 – Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model, International Organization for Standardization (Nov. 1994).

[23] S. Chamberlain, I. L. Taylor, The GNU linker ld (GNU binutils) Version 2.26, Free Software Foundation, Inc. (2015).

[24] G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano, Fixed-length payload encoding for low-jitter Controller Area Network communication, IEEE Transactions on Industrial Informatics 9 (4) (2013) 2155–2164. doi:10.1109/TII.2013.2240310.

[25] NXP B.V., LPC17XX User manual, UM10360 rev. 2 (Aug. 2010).

[26] T. Williams, C. Kelley, Gnuplot 5.0 — An Interactive Plotting Program (2015).

[27] ISO, ISO 16845 – Road vehicles – Controller area network (CAN) – Conformance test plan, International Organization for Standardization (2004).



**Gianluca Cena** received the Laurea degree and the Ph.D. degree in information and system engineering from the Politecnico di Torino, Italy. 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. His research interests include wired and wireless industrial communication systems, and real-time protocols. He has coauthored more than 130 technical papers and one international patent. Dr. Cena served as a Program Co-Chairman for IEEE WFCS 2006 and 2008. Since 2009 he has been an Associate Editor of the IEEE Transactions on Industrial Informatics.



**Ivan Cibrario Bertolotti** has been a Researcher with the Institute of Electronics, Computer and Telecommunication Engineering of the National Research Council of Italy (CNR–IEIIT), Turin, since 1996. He has co-authored two books on real-time operating systems and regularly serves as a Technical Referee for primary international journals and conferences. His research interests include real-time operating system and communication, 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** received her master degree in Computer Engineering in 2010 and PhD degree in Computer and Control Engineering in 2015 from Politecnico di Torino, Turin, Italy. Between 2010 and 2016, she also worked as a research fellow with the National Research Council of Italy. Since December 2016, she became a post-doc researcher of University of Luxembourg. Her primary research interest concerns real-time embedded systems and industrial communication protocols. Currently, she is focusing on model driven engineering for safety-critical systems. She serves as program committee member and technical referee for several primary journals and conferences in her research area.

**Adriano Valenzano** is Director of Research with the National Research Council of Italy (CNR). He is currently with CNR-IEIIT, Torino, Italy, where he is the supervisor of the Computer Engineering and Networks group. He is the recipient of the 2013 IEEE IES and ABB Lifetime Contribution to Factory Automation Award. In 2017 he has been awarded for the best paper published in the IEEE Transactions on Industrial Informatics (TII). He also received the Best Paper Awards at the 5th, 8th and 13th IEEE Workshops on Factory Communication Systems. Since 2007, he has been serving as an Associate Editor for the IEEE TII.