An Embedded CAN Controller for a Vehicle Networking Course Project

Gabriela Breaban, Martijn Koedam, Jeroen Voeten, and Kees Goossens
Eindhoven University of Technology, The Netherlands
{g.breaban,m.l.p.j.koedam,j.p.m.voeten,k.g.w.goossens}@tue.nl

ABSTRACT

The automotive industry advances quickly, with new functionalities continuously being introducted. The Eindhoven University of Technology's Bachelor Automotive Programme prepares students for the subsequent Master education, for industry, and research. In this paper we present the infrastructure and the organisation of the third-year Vehicle Networking course that introduces the current and future automotive networks to students. In the practical part of the course the students use a multiprocessor platform to implement and test an embedded CAN controller. We present requirements and how we address them in the platform architecture, the server-based FPGA infrastructure, and how students design, debug, and analyse their CAN controller. We conclude with lessons learnt and future improvements.

1. INTRODUCTION

Automotive is an important application domain with high innovation. New technologies are added at a high rate to modern vehicles. Automotive engineers must be skilled in many state-of-the-art technologies. This motivated the Eindhoven University of Technology (TUE) to create specialised automotive programs at Bachelor (BSc), Master (MSc), and Professional Doctorate (PDEng) levels. The students following the automotive specialisations will be educated in the latest and emerging concepts and technologies. For Electrical Engineering two technology domains stand out: computation, implemented on Electronic Control Units (ECU), and communication between sensors, actuators, and ECUs.

Typical communication protocols include LIN, CAN, FlexRay, TTEthernet, etc. Of these, CAN is the most widely used network [7], with automotive Ethernet variants poised to take over. In addition, dependable wireless intra- and inter-vehicle networks are emerging. Regarding computation on the ECUs, a shift from the single-processor platforms to multi-processor platforms in underway [10]. This is evidenced by the AUTOSAR standard supporting the multi-core paradigm by extending the basic software architecture to include the multi-core concepts as of release 4.1.1 [1].

In this paper we present the Bachelor Vehicle Networking course (5AIC0) from the Electrical Engineering department of the Eindhoven University of Technology (TUE). The 5-ECTS course has been given for two years to the third-year Bachelor students.

1.1 Course Goals

The vehicle-networking course aims to teach *general networking concepts* (protocol stacks, performance, routing, congestion, etc.) and techniques (framing, pipelining, multiplexing) to implement communication services with a certain quality. Students must understand how these concepts are used in traditional networks such as ATM, Internet, and wireless networks. Next, the automotive domain is introduced with its particular characteristics, such as harsh electronic environments, real-time, robustness, reliability, and safety. Students must understand how *networking techniques are applied in automotive networks* such as LIN, CAN, time-triggered CAN, FlexRay, time-triggered Ethernet, and wireless IEEE 802.15.4, and so on. Next to understanding concepts, students must be able to *apply their knowledge* in making mathematical models of networks to compute and/or optimise service parameters. For automotive networks this includes CAN and TTEthernet performance analysis and scheduling techniques. Finally, students must be able to apply their knowledge by implementing part of a network protocol stack, and reporting on it.

1.2 Course Structure

Traditional lectures are used to teach general and automotive networking concepts and techniques. Application of knowledge is encouraged with analytical exercises and homework. A group project ensures practical hands-on application of theory. In terms of organisation, the course takes eight weeks, with two four-hour slots per week. Figure 1 illustrates the dove-tailing of the teaching of general and automotive networking with the project, to practically apply new knowledge as soon as possible.

This paper focuses on the project and the innovation in how it is technically organised. In the project students work in pairs to implement the Medium Access Control (MAC) layer of a CAN driver in C. Students must deliver C code and a report describing their design, implementation, and verification/experiments. CAN was chosen because it is widely used. Its MAC layer illustrates CAN's unique carrier-sense multiple access with conflict resolution (CSMA/CR) technique, and is not too difficult to implement.

ESWEEK '16 Pittsburgh, PA USA

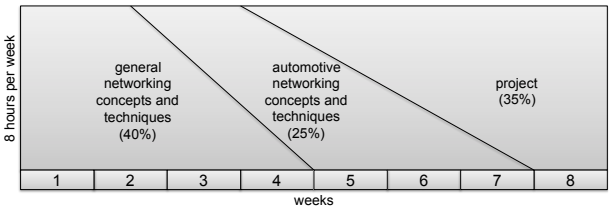

Figure 1: Distribution of topics over the course duration of eight weeks with eight hours per week.

1.3 Project Goals & Innovation

We aim to achieve the following goals with the project:

1. Students understand the PHY and MAC layers of the CAN protocol, as evidenced by implementing the MAC layer. It also refreshes the C programming skills of students, taught earlier in the curriculum.
2. Students can analyse CAN performance, evidenced by allocating CAN priorities to achieve real-time requirements of a set of communicating sensors and actuators.
3. Students are aware of real-time processor performance constraints, as evidenced by implementing the CAN MAC layer in the limited time budget available to process each CAN PHY symbol. They must also take into account the TDM microkernel on each processor.
4. Students are aware of inter-process communication, which is achieved by a communication API between sensor/actuator and CAN driver.
5. Students can work on their projects from any location over the Internet.
6. There is no need to buy CAN hardware and software tooling. This and the previous bullet are required to scale the course with increasing student numbers.

These goals are achieved by a predictable and composable CompSOC platform [4] containing multiple processors extended with *an embedded CAN bus* and associated APIs. The platform is shown in Figure 2.

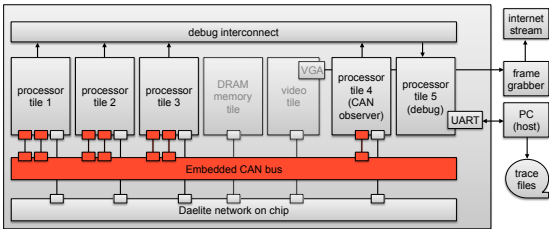

Figure 2: The CompSOC Platform with an Embedded CAN bus.

1. Each processor can host one or more *sensors and/or actuators*, and each sensor/actuator has an accompanying CAN driver on the same processor.
2. The CoMik microkernel on each processor ensures composability: *students can develop their CAN driver independently from other applications* (sensors, actuators) running on the same platform and/or processor.
3. The CoMik microkernel is predictable, and students achieve *real-time performance* for their CAN drivers.
4. A novel *embedded CAN bus* interconnects the memory-mapped CAN interfaces on all processor tiles in the platform. The speed of the CAN bus can be scaled to make the real-time performance requirements on the CAN driver easier or harder [2].
5. Students are given simple APIs for inter-process communication (between sensor/actuator and CAN driver) and to access the CAN PHY (from driver to the CAN hardware interface).
6. The edit, compile, and debug cycle takes place on a linux server, to which students connect over the Internet (using ssh). Students use only the standard C tool chain and a simple "make fpga-run" command. Traditional ECUs require a more complex flashing and debugging tool chain.
7. Multiple CompSOC platforms, each implemented on an Xilinx ML605 board, are permanently available to students over the Internet through servers. The board server load balances, loads student programs in the CompSOC platform, executes programs, and returns the results of execution either via video output (available over the Internet) or a log file with the TTY output. This server set-up has been used for a number of years in several other courses (Embedded Systems Lab, Embedded Control) [6, 9]. A classical automotive set-up with discrete ECUs and CAN bus would require scheduled access to labs, which is less flexible.

The following sections first give background information on the CompSOC platform (Section 2). Following that we describe the hardware and software additions made to the platform for the course in Section 3. Section 4 describes the server infrastructure, and Section 5 the test cases given to students. Section 6 gives the pedagogical lessons learnt from the course, and Section 8 draws conclusions.

2. THE COMPSOC PLATFORM

The CompSOC is a multi-processor platform prototyped on the Xilinx ML605 FPGA. We give the minimum background, details are in, e.g., [4]. A CompSOC platform consists of multiple tiles, interconnected by a Network on Chip (NoC) [11]. Memory tiles contain SRAM or DRAM to offer a distributed shared memory programming model. Processor tiles contain a Microblaze processor with a instruction, data, and communication memories, as well as one or more DMAs. The video tile reads video frames from the DRAM and sends them to a VGA interface that can be connected to a monitor or screen grabber. The debug tile is a processor tile with access to a UART interface that is connected to a PC, allowing bidirectional communication. Using the debug interconnect, each tile can send (limited) information to the debug tile, which is then forwarded to the PC.

A platform can run multiple applications concurrently. Each application can use a NOC, multiple processors, memory tiles, etc. All resources, except the DMAs are shared between applications. Sharing between applications is composable, which eliminates any interference between applications, and predictable for real-time performance.

Figure 2 illustrates the platform used in the course, which runs at 100 MHz. Components newly created and added for the course are coloured, and described in the next Section.

Each processor runs the CoMik [8] microkernel, which creates non-interfering partitions, i.e. cycle-accurate temporal isolation. This is achieved by using TDM and allocating a number of slots to each partition. In addition, each partition has its own stack and heap for spatial isolation of partitions.

Communication between tasks of an application on the same or different processor tiles is performed with a specially

developed sampling channel, described in Section 3.4.

A debug library allows applications to send information to the PC via the debug tile and UART, albeit at a low bandwidth. The debug API automatically time-stamps the debug data, to ease debugging of real-time performance of distributed (multi-core) applications.

3. THE CAN MAC DRIVER PROJECT

Students have to design and implement a CAN MAC driver, and test it on a number of test cases, and document their work in a report. To allow the students to focus only on the CAN driver, we extended the standard CompSOC platform and design flow with the following:

1. *An embedded CAN bus.* Processor tiles 1-3 have two embedded CAN bus interfaces each. The CAN observer tile has one CAN bus interface. They are illustrated in red in Figure 2.
2. *A CAN PHY library* allows applications to sense/read the current symbol on the CAN bus and to send a symbol on the CAN bus, using the memory-mapped access to the CAN interface. This is the vertical API "V" in Figure 3 (which only shows one CAN interface, sensor, actuator, and driver, for space reasons).
3. Processor tiles 1, 2, and 3 each run four independent applications: two *sensor/actuator applications* and two CAN driver applications. The behaviour of sensor and actuator applications, i.e. priority, period and offset of sending and receiving messages, can be configured easily by students in a C header file per sensor/actuator (s.h and a.h in the figure). Each sensor/actuator has a dedicated CAN driver.
4. *An inter-task communication library*, allowing sensors to send messages to the MAC driver, and allowing actuators to receive messages from the MAC driver. This is the horizontal API "H" in Figure 3.
5. A simple software tool to *present debug information* received by the PC from the debug processor in an easy-to-read format.
6. A simple directory structure and makefile; students need only write one C function for the CAN MAC driver, and configure the sensor and actuator header files. The driver function (`dr.c`) in the figure 3 is automatically inserted in all CAN applications.
7. A server-based infrastructure to program applications and to execute them on FPGAs.
8. A series of progressively more advanced *tests*, allowing students to incrementally build and test their drivers.

We will describe all components in the following sections.

3.1 Embedded CAN Bus Hardware Interface

The CompSOC platform was extended to include an on-chip CAN bus with an associated CAN observer tile. Each processor tile can have zero or more CAN hardware interfaces (two in the platform used in the course). A processor uses memory-mapped I/O on the local peripheral AXI bus (as shown in the figure 3) to access the CAN hardware interface. Three registers are exposed to the CAN PHY API (described below): time, read, and write. Processor transactions to these registers result in writing a dominant or recessive symbol on the CAN bus, or read the symbol status of the bus. As shown in Figure 3, the CAN bus consists of a wired-AND with a single bit input from all CAN hardware interfaces. The AND's output is computed by sampling its

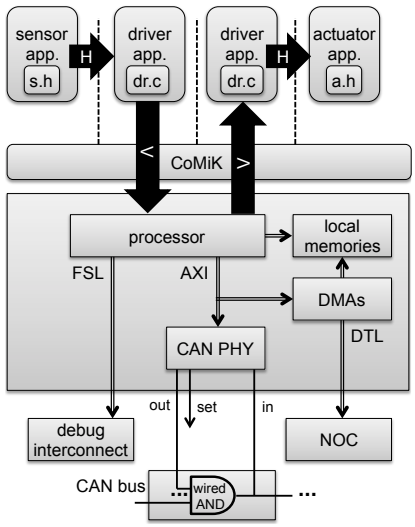

Figure 3: Tile Hardware & Software Architecture.

input every CAN symbol (4000000 cycles at 100 MHz), and then distributed to the CAN hardware interface of the processor tiles. The CAN bus clock is subsampled from the system clock. For the course, the CAN bit rate is only 25 bits per second, to make sure that the students's driver does not have to be very fast. We have shown in [2] that speeds up to 100 kbits per second are possible.

The CAN hardware interface also has a "set" output that indicates whether the processor tile has updated the current symbol or not. These bits are routed to the CAN hardware interface on the CAN observer tile, which forwards this information to the debug processor. This feature greatly helps in debugging, as we describe below.

The CAN bus and the CAN hardware interface in the tile are automatically generated by our design flow together with the rest of the platform [5].

3.2 Embedded CAN Bus Software API

We offer a simple software API, shown as the vertical interface "V" in Figure 3, to facilitate software access to the CAN hardware interface. `can_phy_tx_symbol` (over)writes a dominant or recessive value on the bus in the current symbol duration. The default output of the CAN hardware interface is recessive, even when no symbol is written. `can_phy_rx_symbol_blocking` waits until the end of current symbol and then returns the result of the bus arbitration. Both functions operate on a (pointer to) a CAN hardware interface structure, to allow multiple interfaces on a single processor tile. The time field (register) is used by the software API to block until the end of the current symbol when reading. At the start of each CAN symbol the hardware resets the "set" bit used for debugging. Since the CAN bus operates on a subsampled system clock, the CAN bus's symbol length can be expressed in clock cycles.

3.3 Sensor and Actuator Specification

Sensors and actuators essentially produce, resp. consume, CAN messages periodically. Their behaviours are specified with a simple C header file (`s.h` and `a.h` in Figure 3) containing message priority, period, offset, and size (for sensors), or period, offset, and priorities (for actuators). Sensors produce messages with one identifier (priority) only, where actuators

```
typedef enum {RECESSIVE=1, DOMINANT=0} CAN_SYMBOL;
/* CAN symbol length in processor clock cycles */
#define CAN_PHY_SYMBOL_LENGTH 1000
/* memory-mapped CAN hw interface registers */
typedef struct {
  uint32_t time, read, write;
} CAN_PORT;
void can_phy_tx_symbol(
            volatile CAN_PORT *pPort,
            CAN_SYMBOL state);
void can_phy_rx_symbol_blocking(
            volatile CAN_PORT *pPort,
            CAN_SYMBOL *pSymbol);
```

Figure 4: CAN PHY software API (can_phy.h).

can receive messages with different priorities. Students determine these parameters as part of their project (Section 5).

```
/* Sensor (CAN bus master) schedule */
#define CAN_MSG_PRIO_1      0
#define CAN_MSG_TX_PERIOD_1 6750000000
#define CAN_MSG_TX_OFFSET_1 0
#define CAN_MSG_TX_DLC_1    3
/* Actuator (CAN bus slave) schedule */
#define CAN_MSG_RX_PERIOD_1       6750000000
/* 2x worst-case CAN frame = 2x137 symbols */
#define CAN_MSG_RX_OFFSET_1       3500000000
#define CAN_MSG_RX_PRIOS_1  4
int rxPrioFilt_1[CAN_MSG_RX_PRIOS_1] = {0,1,3};
/* actuator 1 receives message IDs 0,1,3 */
```

Figure 5: Specifications of Sensor 1 and Actuator 1.

3.4 Inter-Task Communication API

As mentioned before, each processor runs four applications, in composable partitions. Each sensor and actuator must have its own driver. In the course, (rather arbitrarily) processor tile 1 has two sensors, tile 2 and 4 have a sensor and actuator, and tile 3 has two actuators. The CAN MAC driver must accept messages (CAN frames) from sensors, and convert them to a series of symbols to be placed on the bus. Conversely, the driver must reconstitute symbols on the bus to CAN frames, and send them to the actuator if they have the required message identifier. The inter-task communication between sensors/actuators and driver is sampling, and thus potentially lossy. If the driver does not consume messages quickly enough from the sensor, they are overwritten and lost. Similarly, if the driver would send messages faster than the actuator's sampling rate, then messages would be overwritten and lost.

Figure 6 describes the API on the driver's side ("H" in Figure 3), which students use. The driver has a pointer to a lossy communication channel (`ppSensor` and `ppActuator`) over which messages (CAN frames) are received from sensors or sent to actuators. `can_mac_rx_next_frame` is non-blocking.

When implementing the MAC driver, the students need to use the CAN PHY API to create the frame bit by bit and the communication API to access the transmit and receive message buffers. Furthermore, they receive the template code shown in 8 inside of which they need to place the driver code. The template includes a short example showing how the CAN PHY and MAC interfaces can be used to get a CAN message from the sensor and transmit it bit by bit.

```
typedef struct {
  uint32_t ID;
  uint32_t DLC;
  uint64_t Data;
  uint32_t CRC;
} CAN_FRAME;
/* API used by the CAN MAC driver */
bool can_mac_rx_next_frame(
            CAN_FRAME * volatile * ppSensor,
            CAN_FRAME * pTxFrame);
void can_mac_tx_next_frame(
            CAN_FRAME * volatile * ppActuator,
            CAN_FRAME * pRxFrame);
/* API used by sensors (not for students) */
bool CAN_write(uint32_t sensorId,
               const CAN_FRAME * txFrame);
/* API used by actuators (not for students) */
bool CAN_read(const CAN_FRAME * rxFrame);
```

Figure 6: Inter-Task software API (can_mac.h).

Each CAN MAC driver is associated with either a sensor or an actuator. Since the sensor only transmits frames and the actuator only receives frames, the MAC driver can be either in transmission or in reception mode. To determine this, the students can check the content of the receive ID filter: if it contains a negative value, then the driver belongs to a sensor, otherwise it belongs to an actuator.

3.5 CAN MAC Driver Template

The CAN MAC driver uses the inter-task software API to receive messages from sensors to convert them to a sequence of CAN symbols that must be sent over the embedded CAN bus. At the same time, it must recognise any CAN messages arriving over the bus that are addressed to the actuators. For simplicity, in the course each driver communicates with either a single sensor or actuator. ([2] shows how a single driver can implement a CAN gateway for multiple CAN clients.) Figure 8 illustrates the basic CAN MAC driver template given to students. It is automatically inserted in the larger software stack (application partition, etc.), as illustrated by the `dr.h` in Figure 3. They only need to fill in the `while(1)` loop, making use of the inter-task API (Figure 6) and CAN PHY API (Figure 4).

The assignment involves polling for an incoming sensor message, and converting it to a sequence of CAN symbols. Or, listening for messages that must be sent to the actuator. Several CAN techniques must be implemented:

1. determining the start of transmissions by other drivers;
2. recognising and decoding sequences of CAN symbols into CAN frames, including variable message lengths;
3. aligning the sending of CAN frames with other transmissions and implementing CAN's unique symbol-wise priority resolution;
4. bit stuffing;
5. cyclic redundancy check (CRC);
6. acknowledgement;
7. inter-frame spacing and error handling.

3.6 Embedded CAN Bus Tracing

The data that is received by the PC over the UART is post-processed and two kinds of reports are generated. The first, shown in Figure 9 (left) is a trace of all CAN symbols sent to the CAN bus by the processors, and the resulting CAN symbol after dominant/recessive resolution. This al-

lows for low-level debugging, which is useful at the start of the project. The first time value represents the system time in clock cycles, the second value is the CAN symbol (bit) count; 'out' is the output of the CAN bus (output of the wired AND). The remaining fields show the CAN port values received on each CAN interface from the drivers running on the tile (displayed as "tile.interface=value"). "R" stands for recessive and "D" for dominant. 'U' stands for unset, meaning that that CAN interface did not update its default "R" value. To be able to display this, the embedded CAN bus has a single "set" bit for each CAN interface (cf. Figure 3). This feature greatly helps in debugging, for example to determine when a driver is too late producing a symbol, or accidently not sending a symbol.

The second output is per frame, shown in Figure 9(right), and is more useful for longer traces and when debugging frame-level priority resolution, etc. This report is produced for each sensor and actuator independently.

The CompSOC platform also has a video output connected to a frame grabber, as shown in Figure 2. The CAN observer can send a real-time wave-form trace of the CAN bus to the video output. However, for this real-time wave-form trace to be useful, the bit rate of the CAN bus has to be very low (seconds per symbol). Since live debugging is only useful at the very beginning of the project we decided to only offer debugging using the log files.

4. THE SERVER INFRASTRUCTURE

Six ML-605 FPGA boards are available in total for students and researchers via a board server. The board server implements access control, load balancing, and priority queueing. Jobs are submitted to the board server from a linux server that can be accessed on the TUE (virtual private) network (VPN) using ssh. Student(groups) are given an account on the linux server. Students are expected to collaborate using git version management, and also submit their final code by tagging their code in git.

Users of the FPGA farm can upload either a complete CompSOC platform (a bitstream containing hardware, system software, user applications) or they can upload (ELFs of) user applications that are dynamically loaded in a CompSOC platform that persists on the FPGA boards. The former is typically used by researchers and allows modification to both hardware, system software, and user applications. The latter is typically used by students, because they do not (and should not) modify the CompSOC platform but only change the user applications.

The second option is also much faster, since platform generation, RTL synthesis, and compilation of platform software is not required. With 6 boards, we sustain 480 jobs per hour.

In the vehicle networking course students only modify the user applications i.e. the sensors, actuators, and drivers (`s.h`, `a.h`, and `dr.h` files in a given directory structure) as described before. These are then dynamically loaded on a preloaded CompSOC platform instance that includes the embedded CAN bus. A Makefile is provided with targets for compiling, linking, running on the FPGA, and converting the logging output (cf. previous section).

The CompSOC platform with the embedded CAN bus has a UART output (921600 bits/sec) to connect to the linux host to upload user applications, and for debugging output. A few FPGA boards also have video output that is captured by a frame grabber and offered as a live video stream on the Internet.

This FPGA farm and server setup is used in several other MSc courses in the faculty, including the Embedded Systems Lab [6] and the Embedded Control Systems course. Students can collaborate (using git) and work from anywhere by logging in (using VPN) on the linux server.

5. TESTING & REAL-TIME PERFORMANCE

After implementing the MAC driver, the students have to test the implementation by running 11 test schedules, of which 6 are basic and 5 are advanced. The basic tests check basic protocol features, such as the arbitration between messages with different priorities or that the reception of the messages happens according to the reception filter. The advanced tests check different relations between message periods (e.g. periods that are not a multiple of each other), different message offsets, and also messages with different payload sizes. Every given test case comprises a set of CAN senders (sensors) with their own priorities and periods. To implement these test cases, the students need to understand and apply the notion of priority-based schedule. They are also exposed to the notion of worst-case execution time (WCET). The WCET depends on different components such as bit stuffing, (variable) message length, CDMA/CR arbitration, and the sender priorities. The students need to have a good understanding of how the CAN protocol works as well as the priority-based arbitration as used in CAN (periods, priorities, offsets, deadlines).

Figure 7 shows an example of a basic test case. The parameters are the message priorities, the message periods and offsets, and the worst-case payload sizes. It also shows the expected schedule output and the expected delay per message. The period is measured in terms of worst-case number of clock cycles for one frame with a payload of one byte, including bit stuffing. With this specification, the students are supposed to configure the periods, priorities, and offsets of sensor and actuator applications accordingly (`s.h` and `a.h` in Figure 3). They then run the test case and explain the obtained results (in particular, whether the test gives the expected output or not and why).

While in the majority of cases, the students do receive the results as specified, it was also possible that their output did not match the expected output. The test case of Figure 7 is such an example. The message periods of sensors A, B, C are based on the worst-case frame length, but the actual length of the messages can be shorter than that due to less data or less bit stuffing. As a result, the actual bus schedule may be not aligned with the message period, yielding a different schedule than in the worst case. For example, if sensors A, B, and C start sending at the same time, the messages are transmitted in the expected order (A B C). However, in the next round, when B and C start sending, C loses the arbitration and waits until B completes. If the message sent by B is shorter than the worst case, it finishes the transmission before A restarts. Then C can start transmitting before A, resulting in the trace B C A rather than B A C. Students are supposed to analyse and understand the actual behaviour of the system versus the worst case and correctly explain what they observed.

6. LESSONS LEARNT

The CAN driver project has been given once to third-year Bachelor students of the Automotive Programme. We observed a number of points. First, students learnt C at the start of the first year, and many had not programmed in C since then, especially bit-level operations (shifting, masking, OR, AND, etc.). This year, we will help student to refresh the C skills before the project starts (concurrently with the general networking, see Figure 1).

Second, students had not done any embedded and multi-threaded programming earlier in the curriculum. As a result, students had little or no experience with, for example, volatile pointers to hardware registers and shared data structures. Although we abstracted the memory-mapped interface with the CAN driver to a simple CAN PHY API, and the inter-task communication to the simple CAN MAC API, this part of the project required longer than expected.

Third, offering a set of tests of increasing complexity allowed students to test and deliver drivers with a subset of all required features, rather than an all-or-nothing approach. Together with the written report, this allowed students to be fine graded, with grades ranging from 3.2 to 8.8 on a scale from 1-10. 70% of 29 students passed.

Fourth, although the server-based infrastructure has great operational advantages, most students had not used linux before. In particular concepts such as servers, remote login, (secure) shell, and RSA key generation were new to them, as were basic linux commands (`ls`, `cd`, etc.). We will give more instructions to students in future editions of the course. Next to that, in the first-year C course we will switch to a linux-based programming environment (virtual machine with Ubuntu) to ensure that all Electrical Engineering and Automotive students have basic linux skills.

Finally, to close the learning cycle, students peer-reviewed the reports that they had to deliver. To stimulate autonomy and creativity students received quite high-level instructions on the scoring criteria, such as: does the code compile & run, which test cases are correct; does the report have a good structure, can you understand the code by reading the report, are features & limitations defined; does the code have interesting features/tricks. The quality of the peer reviews was high, and students gave good differentiated grades that were close to the grades of the lecturers.

7. RELATED WORK

Related courses can be found at several other universities. Michigan Technological University offers an automotive communications network course for EE and ME students that covers the theory for the CAN, LIN, FlexRay and MOST buses [12]. The lab consists of several assignments for CAN where use Arduino boards to configure and program a network of 2 to 3 CAN nodes. The CAN nodes connect to sensors and actuators such as push bottons, obstacle detectors and DC motors to emulate parts of the car. Keterring University partners with Vector CANtech, a company that provides software tools for CAN and dSPACE, a company that offers complete ECU solutions, for the lab for its Distributed Embedded Systems course in the Computer Engineering department. The assignments introduce the students to distributed systems, starting with the simulation environment (Vector's CANoe tool) followed by real ECUs. The concepts of network scheduling are taught both within simulation and also within a mixed environment including simulated and real ECUs. The Computer Science department at TUE teaches automotive real-time concepts, including CAN, in the Real-Time Architectures course [3]. In the lab, the students use Freescale microcontrollers together with the Code Warrior IDE and develop the CAN driver under the μC-OS operating system.

Our course covers the same basic concepts, such as network scheduling and interfacing with sensors and actuators. The main difference is that we use a research platform for the laboratory, as opposed to commercial tools and emphasise the development of the CAN driver rather than its use.

8. CONCLUSIONS

In this paper we presented the third-year vehicle-networking course of the Eindhoven University of Technology's Bachelor Automotive Programme. We defined the goals, course structure, and the innovation in the course. In particular, we embedded a CAN bus in a multi-processor platform, and required students to develop a CAN MAC driver. Our server-based FPGA infrastructure allows students to access course hardware and develop and debug their driver over the Internet at any time. By offering a set of test cases, students were encouraged to develop their driver one feature at a time, resulting in a range of grades, and 70% pass rate.

This work was partially funded by projects CATRENE ARTEMIS 621429 EMC2, 621353 DEWI, 621439 ALMARVI.

9. REFERENCES

[1] AUTOSAR Release 4.1 - Guide to Multi-Core Systems. Technical report.

[2] G. Breaban, M. Koedam, S. Stuijk, and K. Goossens. Virtualization and emulation of a CAN device on a multi-processor system on chip. In *MECO*, 2016.

[3] R. J. Bril and M. J. Holenderski. 2IN60 Real-Time Architectures in the Automotive Technology Master at the Eindhoven University of Technology, 2008-2014.

[4] K. Goossens, et al. Virtual execution platforms for mixed-time-criticality systems: The CompSOC architecture and design flow. *SIGBED Review*, 2013.

[5] S. Goossens, et al. The CompSOC design flow for virtual execution platforms. In *FPGA World*, 2013.

[6] A. Hansson, B. Akesson, and J. van Meerbergen. Multi-processor programming in the embedded system curriculum. *SIGBED Review*, 6(1):9:1–9:9, 2009.

[7] ISO11989-1:2015 road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. Technical report.

[8] A. Nelson, et al. CoMik: A predictable and cycle-accurately composable real-time microkernel. In *DATE*, 2014.

[9] A. Nelson, et al. Embedded computer architecture laboratory: A hands-on experience programming embedded systems with resource and energy constraints. In *WESE*, 2012.

[10] D. Reinhardt and M. Kucera. Domain Controlled Architecture - A new approach for large scale software integrated automotive systems. In *PECCS*, 2013.

[11] R. Stefan, et al. dAElite: A TDM NoC supporting QoS, multicast, and fast connection set-up. *IEEE Trans. on Computers*, 63(3):583–594, May 2014.

[12] A. Oliveira. Development of a Low-cost Automotive Communications Network Course for EE and ME Students. In *ASEE*, 2016.

priority	period	dlc/size	offset	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
0	4	1	0	A				A				A				A				A				A				A				A
1	3	1	0	B			B			B			B			B			B			B			B			B			B	
2	3	1	0	C			C			C			C			C			C			C			C			C			C	
			schedule	A	B	C	B	A	C	B	C	A	B	C		A	B	C	B	A	C	B	C	A	B	C		A	B	C	B	A
			delay		1	2			2		1			1			1	2			2		1			1			1	2		

Figure 7: A basic CAN Test Schedule.

```
static void hw_can_mac_driver(
                      volatile CAN_PORT *pPortID,
                      CAN_FRAME * volatile *pSensor,
                      CAN_FRAME * volatile *pActuator,
                      int *rxPrioFilters, uint32_t rxPrioFiltersLen)
{
  CAN_FRAME TxFrame, RxFrame;
  bool gotFrame;
  CAN_SYMBOL TxSymbol, RxSymbol;
  while (1) {
    /* if rxPrioFilter < 0 then we're master else slave
     * as a master, to get the next frame to send on the bus from the sensor use:
     * gotFrame = can_mac_rx_next_frame(pSensor, &TxFrame);
     * as a slave, to send a frame received on the bus with priority rxPrioFilter to the actuator use:
     * can_mac_tx_next_frame(pActuator,&RxFrame);
     */

    /* to send a CAN symbol on the CAN bus use: can_phy_tx_symbol(pPortID, TxSymbol)
     * to receive a CAN symbol from the CAN bus use: can_phy_rx_symbol_blocking(pPortID,&RxSymbol)
     * this function blocks until a new symbol is available on the bus
     */
    gotFrame = can_mac_rx_next_frame(pSensor, &TxFrame);
    if (gotFrame == true) {
      /* if you wish to send different test sequences from different sensors,
       * then this is how you can do this. Note that this should not be used in the final driver,
       * since there all sensors use the same driver code.
       */
      if (TxFrame.ID == 0) {
        /* sensor 1 with priority/ID 0 */
        can_phy_tx_symbol(pPortID, DOMINANT);
        can_phy_rx_symbol_blocking(can_port_id,&RxSymbol);
        can_phy_tx_symbol(pPortID, DOMINANT);
        can_phy_rx_symbol_blocking(can_port_id,&RxSymbol);
        can_phy_tx_symbol(pPortID, RECESSIVE);
        can_phy_rx_symbol_blocking(can_port_id,&RxSymbol);
      } else {
        /* the other active sensors */
        can_phy_tx_symbol(pPortID, RECESSIVE);
        can_phy_rx_symbol_blocking(pPortID,&RxSymbol);
      }
    }
  }
```

Figure 8: CAN MAC driver template, given to students.

```
CAN PHY MON: time[cc]=    4000241 time[symb]=   1 out=R 1.1=U 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=    8000232 time[symb]=   2 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   12000234 time[symb]=   3 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   16000236 time[symb]=   4 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   20000238 time[symb]=   5 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   24000240 time[symb]=   6 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   28000242 time[symb]=   7 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   32000233 time[symb]=   8 out=R 1.1=R 1.2=R 2.1=R 2.2=U 3.1=U 3.2=U 4.1=R 4.2=U
CAN PHY MON: time[cc]=   36000235 time[symb]=   9 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   40000237 time[symb]=  10 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   44000239 time[symb]=  11 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   48000241 time[symb]=  12 out=D 1.1=D 1.2=D 2.1=D 2.2=U 3.1=U 3.2=U 4.1=D 4.2=U
CAN PHY MON: time[cc]=   52000232 time[symb]=  13 out=D 1.1=D 1.2=D 2.1=R 2.2=U 3.1=U 3.2=U 4.1=R 4.2=U
CAN PHY MON: time[cc]=   56000234 time[symb]=  14 out=R 1.1=R 1.2=R 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   60000236 time[symb]=  15 out=D 1.1=D 1.2=R 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   64000238 time[symb]=  16 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   68000240 time[symb]=  17 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   72000242 time[symb]=  18 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   76000233 time[symb]=  19 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   80000235 time[symb]=  20 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   84000237 time[symb]=  21 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   88000239 time[symb]=  22 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   92000241 time[symb]=  23 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=   96000232 time[symb]=  24 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  100000234 time[symb]=  25 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  104000236 time[symb]=  26 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  108000238 time[symb]=  27 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  112000240 time[symb]=  28 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  116000242 time[symb]=  29 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  120000233 time[symb]=  30 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  124000235 time[symb]=  31 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  128000237 time[symb]=  32 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  132000239 time[symb]=  33 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  136000241 time[symb]=  34 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  140000232 time[symb]=  35 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  144000234 time[symb]=  36 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  148000236 time[symb]=  37 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  152000238 time[symb]=  38 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  156000240 time[symb]=  39 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  160000242 time[symb]=  40 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  164000233 time[symb]=  41 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  168000235 time[symb]=  42 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  172000237 time[symb]=  43 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  176000239 time[symb]=  44 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  180000241 time[symb]=  45 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  184000232 time[symb]=  46 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  188000234 time[symb]=  47 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  192000236 time[symb]=  48 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  196000238 time[symb]=  49 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  200000240 time[symb]=  50 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  204000242 time[symb]=  51 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  208000233 time[symb]=  52 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  212000235 time[symb]=  53 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  216000237 time[symb]=  54 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  220000239 time[symb]=  55 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  224000241 time[symb]=  56 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  228000232 time[symb]=  57 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  232000234 time[symb]=  58 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  236000236 time[symb]=  59 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  240000238 time[symb]=  60 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  244000240 time[symb]=  61 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  248000242 time[symb]=  62 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  252000233 time[symb]=  63 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  256000235 time[symb]=  64 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  260000237 time[symb]=  65 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  264000239 time[symb]=  66 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  268000241 time[symb]=  67 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  272000232 time[symb]=  68 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  276000234 time[symb]=  69 out=D 1.1=D 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  280000236 time[symb]=  70 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  284000238 time[symb]=  71 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  288000240 time[symb]=  72 out=D 1.1=R 1.2=U 2.1=U 2.2=D 3.1=D 3.2=D 4.1=U 4.2=U
CAN PHY MON: time[cc]=  292000242 time[symb]=  73 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  296000233 time[symb]=  74 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  300000235 time[symb]=  75 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  304000237 time[symb]=  76 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  308000239 time[symb]=  77 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  312000241 time[symb]=  78 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  316000232 time[symb]=  79 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  320000234 time[symb]=  80 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  324000236 time[symb]=  81 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
CAN PHY MON: time[cc]=  328000238 time[symb]=  82 out=R 1.1=R 1.2=U 2.1=U 2.2=U 3.1=U 3.2=U 4.1=U 4.2=U
```

```
Sensor 1 Frame 0
ID = 00000000
DLC = 00000003
Data = 00000000
CRC = 00000000

Sensor 2 Frame 0
ID = 00000001
DLC = 00000003
Data = 00000000
CRC = 00000000

Sensor 3 Frame 0
ID = 00000002
DLC = 00000003
Data = 00000000
CRC = 00000000

Sensor 4 Frame 0
ID = 00000003
DLC = 00000003
Data = 00000000
CRC = 00000000

Actuator 1 Frame 0
ID = 00000000
DLC = 00000003
Data = 00000000
CRC = 00000000

Actuator 2 Frame 0
ID = 00000000
DLC = 00000003
Data = 00000000
CRC = 00000000

Actuator 4 Frame 0
ID = 00000000
DLC = 00000003
Data = 00000000
CRC = 00000000
```

Figure 9: CAN bit-level debug output (left) and frame-level debug output (right).