Software-Defined Network Controllers for Industrial and Automotive Applications

Gianluca Cena, Ivan Cibrario Bertolotti, and Adriano Valenzano

{gianluca.cena, ivan.cibrario, adriano.valenzano}@ieiit.cnr.it

CNR – National Research Council of Italy, IEIIT, Torino (Italy)

IWES 2016

1st Italian Workshop on Embedded Systems
19–20 September 2016, Pisa, Italy

Outline

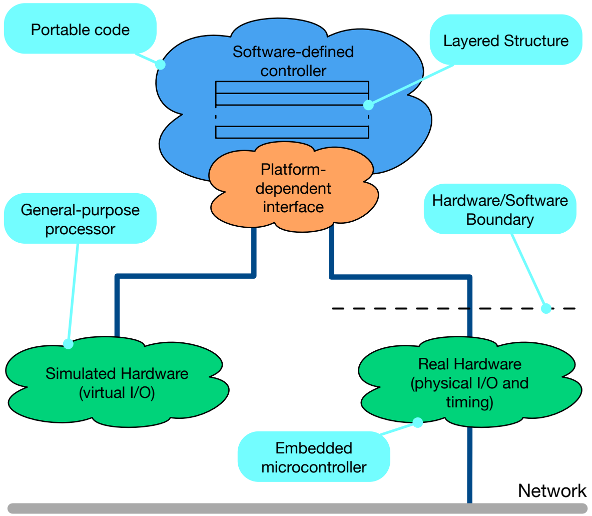
- Introduction and Motivation
- Design Issues
- Case Study: A Software-Defined CAN Controller
- Conclusion and Future Work

Motivation

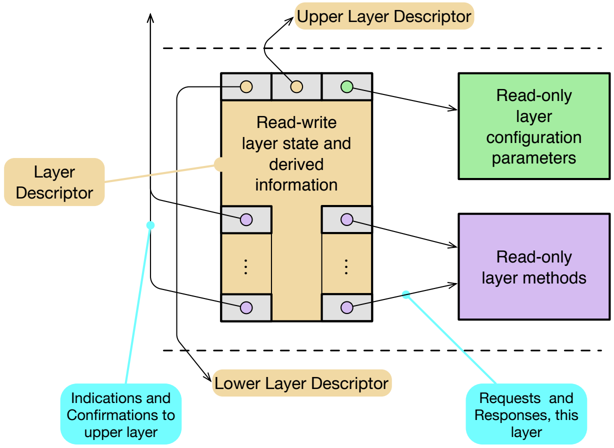
- Experimental work on the data-link protocol layer has customarily been confined to the hardware domain
- FPGA-based implementations are feasible in most cases, but require specialized programming tools. . . and an on-board FPGA.
- A completely software-defined network controller would be more flexible and easier to use, without the disadvantages of a simulator

- Develop an exemplar controller and assess its performance
- Similar to a Software-Defined Radio (SDR)

General Organization


Portable code
Software-defined controller
Layered Structure
Platform-dependent interface
General-purpose processor
Hardware/Software Boundary
Simulated Hardware (virtual I/O)
Real Hardware (physical I/O and timing)
Embedded microcontroller
Network

Internal Structure of a Layer


Upper Layer Descriptor
Read-write layer state and derived information
Read-only layer configuration parameters
Layer Descriptor
Read-only layer methods
Lower Layer Descriptor
Indications and Confirmations to upper layer
Requests and Responses, this layer

🟡 Pointer to read-write data (layer descriptor)

🟢 Pointer to read-only data

🟣 Pointer to (read-only) function

Memory Management

- Mark read-only and read-write data (as well as code) and keep them separate from each other
- This enables data structures and code to be placed in different memory regions
- It makes the optimizer's work easier (and better)
- Any GCC `__attribute` can easily be disabled if unsupported

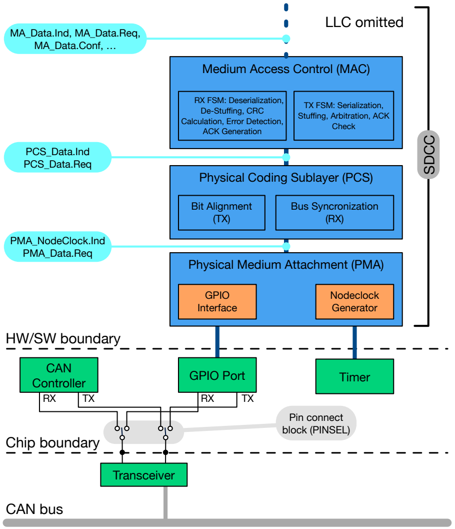
The memory management strategy does not affect portability (even to non-GCC compilers)

Source-Level Code Optimization

- Understand how compilers (especially optimizers) work and “help” them. . .
- . . . staying within the boundary of the language standard
- Often, updates to the code are minimal and improve performance significantly, even on dissimilar architectures
- Effective alternative to using the assembly language for critical real-time modules

Good experience in the past with embedded CAN payload codecs for stuff bit prevention (ZSC)

A Software-Defined CAN Controller (SDCC)


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

- Consists of about 2100 lines of portable C code
- Connected to a real CAN transceiver through an on-chip GPIO port
- The only platform-specific parts are the GPIO interface and the clock generator
- Able to process up to 780000 quanta per second (Cortex-M3 @ 100 MHz)

Advantages of Portability

The same code runs on diverse platforms, for instance:

- NXP LPC1768 (low-end μC for automotive/embedded applications, ARM Cortex-M3 @ 100 MHz, bare metal)
- Intel T9400 (mobile processor for laptop computers, Penryn @ 2.53 GHz, dual core, OS X)

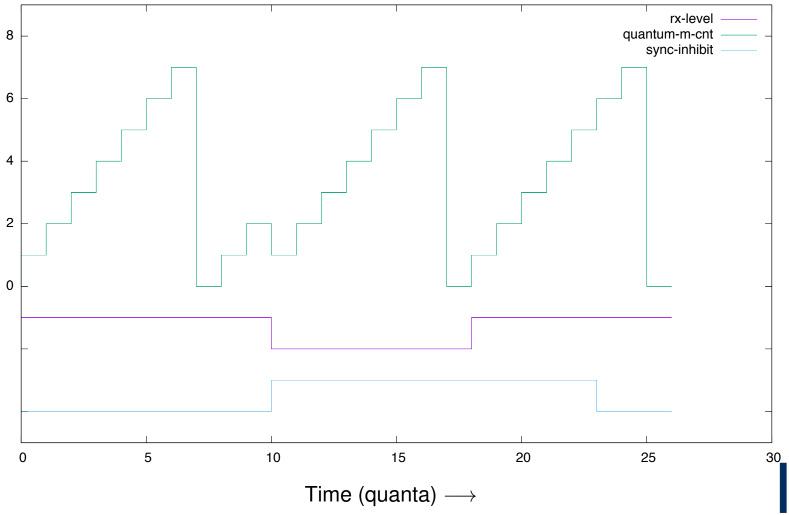
- The only difference lies in the interface with the CAN transceiver and clock generator:
  - Real hardware on the LPC1768
  - Simulated hardware on the T9400
- Portability speeds up software development and improves its quality

Simulation and Observability

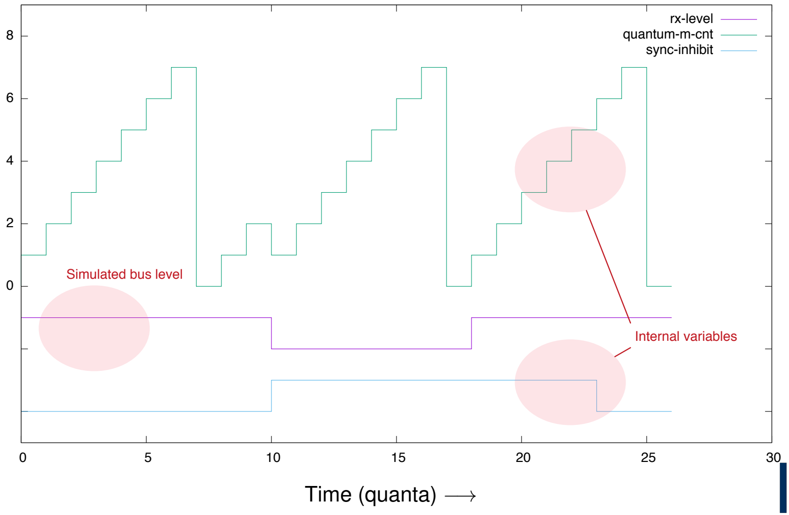
- When SDCC is compiled in simulation mode
  - It works in simulated time (much faster than real time)
  - Its inputs can be driven at will by means of stimuli files
- Useful to explore borderline scenarios that may be hard to reach on a real bus
- Its outputs and (more interesting) internal state are completely observable and can be conveniently visualized (`gnuplot`)

Simulation results are completely faithful because, barring compiler bugs, the SDCC code is the same code that runs on the real hardware

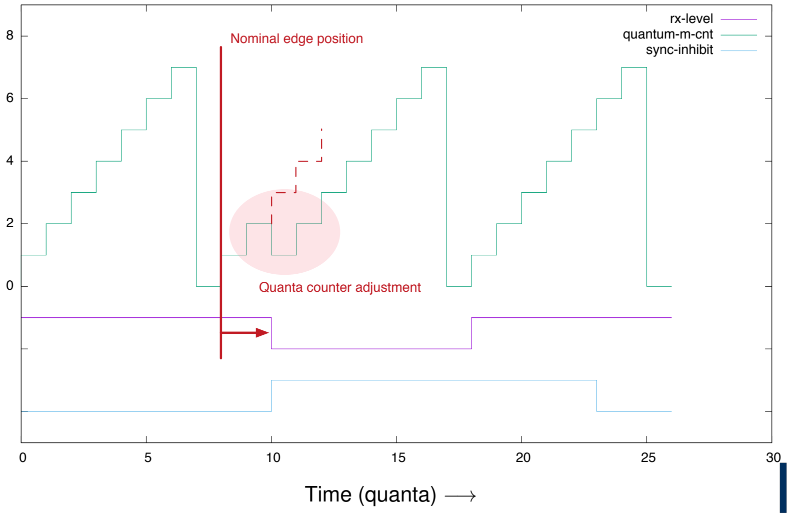
Example: Bit (Re)synchronization in SDCC



Example: Bit (Re)synchronization in SDCC



Example: Bit (Re)synchronization in SDCC



Memory Footprint and Performance

Module	Code Size [B] LPC1768	Code Size [B] T9400
Medium Access Control (MAC)	5096	10183
Physical Coding Sub-Layer (PCS)	1305	2863
Physical Medium Attachment (PMA)	835	940
Total	*7236*	*13986*

- Data/Bss segments are empty in both cases (not a surprise, considering the software design)
- On the LPC1768, SDCC is able to process one quantum every about 128 clock cycles
- Fast enough to drive a real CAN bus at 62.5 kb/s

CAN XR

SDCC has been used to implement CAN XR, an extension of the CAN (FD) protocol that supports in-frame replies

- Simulation mode was used for debugging the protocol during development
- The SDCC-based CAN XR controller was then connected to a real CAN bus (together with standard hardware controllers) to confirm its backward compatibility
- A proof-of-concept CAN XR implementation requires about 150 additional lines of code (in the MAC layer)

Conclusion and Future Work

Software-defined controllers are a valid alternative to hardware and FPGA-based designs due to improved convenience and portability

- Explore further extensions to the CAN standard
- Use software-defined controllers as low-cost bus analyzers. . .
- . . . and (possibly unwelcome) bus sniffers
- Assess whether a C++ implementation is workable (performance issues)
- Improve CAN security at (or by means of) the data-link level (Howard University, USA)

THANK YOU FOR YOUR ATTENTION