Embedded Software V&V using Virtual Platforms for Powertrain applications

P Cuenot, B Tavernier, J. Talbot

Embedded Software V&V using Virtual Platforms for Powertrain applications

P. Cuenot[1], B. Tavernier[2], J.M. Talbot[2]

1: Siemens VDO Automotive SAS, a Continental Corporation company, 1av. Paul Ourliac, BP 1149, 31036 Toulouse, France

2: VaST Systems Technology, 18 rue de La Tuilerie, 38170 Seyssinet-Pariset, France

Abstract:

Current development trends for automotive products are driven by time to market reduction, cost optimization, and quality improvement. Dual to these business constraints are demands for innovation and safety conformance which impose increasing complexity on embedded systems. To address these challenges impacting software and hardware to improve system dependability, new methodology and tools need to be set-up. The use of representative virtual platforms combining speed and accuracy allows earlier software development, improved system testing, and fault injection analysis, with a high potential for reuse of system IPs (including both hardware and software). In this paper, we will present investigation on new methods and associated results using a simplified virtual platform to test a powertrain application.

Keywords: TLM, Verification, virtual platform

1. Introduction

Current development trends in embedded automotive software for Powertrain systems are towards increased complexity due to the very high level of optimization in the control/command. Systems control complexity ranges from low end applications such as gasoline port injection, to more sophisticated applications such as high pressure diesel engine, to very high end applications such as hybrid and power management control. As a consequence, the algorithms applied increase in complexity and require more resources to execute with the level of confidence required by a safety-critical system.

In parallel, the introduction of new features for passenger comfort, the addition of new sensors and actuators to comply with pollution regulation, and interconnection with in-vehicle network communication drives increasing software and hardware complexity which makes the system more difficult to test on a classical test bench. To face the challenge of the greater complexity of both software and hardware, new methodology and tools need to be set-up. In particular, the use of representative virtual platforms which combine speed and accuracy facilitates earlier software development and better testing of the systems. Another industrial issue is hardware design, followed by software development often places the software development in the critical project planning path. Virtual platforms allow early software development in general and hardware dependant software (e.g. device drivers) in particular. Virtual platforms also simplify the distribution of reference hardware architectures throughout geographically distributed development groups within a company. Virtual platforms also benefit the entire supply chain.

Virtual platforms are providing the automotive industry with additional means to face its economical challenges in facilitating Design to Cost optimisation. HW/SW trade-offs, effect of hardware configuration can be assessed by early and fast platform prototyping. New hardware architectures embedding ASICs can be explored to optimize performance.

The key issue for such technology is the availability of hardware models and their conformance to their silicon implementation. Simulation performance and standardization of hardware modelling style are mandatory for industrial usage in automotive embedded systems. We believe that Transaction Level Modelling is adequate to reach these goals as supported by the systemC OSCI initiative [1].

In this paper, we will present a virtual platform based on a simplified version of the Copperhead micro-controller (MPC5554 from Freescale), including an external ASIC model developed in SystemC by Continental. This platform was developed using the CoMET tool from VaST Systems Technology. We focused our study on demonstrating on how this platform can facilitate early software development and non-regression testing of the system without the need for physical hardware.

In the first section, the methodology used for hardware modelling with a TLM style is presented, while section two describes the use case study and a platform overview. Section three details the technical results obtained and compare them to the initial objectives. Finally, the conclusion outlines the

benefits of such approach and possible extensions for industrial application.

2. Modeling Technology

A virtual platform enabling software development for powertrain application (and more generally safety critical application) must be fast and accurate. With a TLM approach, VaST core models like the e200z6 are able to run classical benchmarks like Viterbi at more than 80 MIPS (with more than 99% cycle accuracy) on a 2 GHz Pentium 4 machine with 1 Go of RAM. Keeping that range of speed at the platform level can be achieved by abstracting the signals and the behaviour to a higher level than RTL without compromising the timing accuracy at the system level.

In this chapter, we will present some modelling principles to maximize the simulation speed in the platform. First we will describe how clock signals are modelled and how communication channels like a serial link, an internal bus, or a CAN bus can be abstracted at a transaction level for modelling (TLM). Then we will propose generic principles for reducing the number of events in a SystemC peripheral model like a timer. We will see that this model can be used for acquisition of frequency inputs (like event capture in powertrain applications) with both speed and accuracy.

Clock Modelling

Clocks are one of the most critical signals we need to abstract for having a fast and accurate virtual platform. Since simulation kernels for modelling virtual platforms are event driven, a pulsing clock will generate an event on every edge, even if most of these events are useless.

For example, if a peripheral is in an inactive state but has a pulsing clock as an input triggering a callback function (for example a SC_METHOD sensitive to positive edges), some code will be executed by the simulation kernel even if it will have no effect on the platform behaviour.

To improve the simulation speed, clock signals can be abstracted by their period, implying the following advantages:

- An event is generated only when the period of the clock signal changes. The occurrence of that kind of event is far less frequent than the clock pulse.
- We can keep full cycle accuracy by replacing clock driven behaviour by clock computation and timing annotation and generate only observable events.
- With that kind of abstraction, dynamic clocking in peripheral models comes for free.

With its modelling API, VaST simulation tools provides a set of useful functions for reconstructing clock signals from a period-based clock representation, which helps in raising the level of modelling abstraction to TLM. This API also facilitates standard operations for peripheral modelling like programming a callback in a specific number of clock ticks.

In SystemC modelling, sc_clock primitive objects are pulsing clock models. They should be avoided and replaced by an integer signal modelling the clock period. This kind of abstraction will require more computation than using the VaST API, but this can be easily managed by a helper class in charge of all the conversions between number of clock ticks and absolute time (as required in SystemC).

Communication Channels

The way to abstract communications over a specific channel follow the same principles:

- Data frames are passed from the producer to the consumer in one step.
- Only observable parts of the protocol are modelled and (if required) associated with events. If the producer (or consumer) does not require these internal events (like partial phases of a burst transaction), the events are suppressed and this makes the protocol even faster.
- Channels with arbitration are modelled by a specific “protocol engine” in charge of scheduling each transaction and granting the bus access to each master.

Serial Protocol

A serial transfer is abstracted to convey only the information relevant from the system point of view. Typically, on the simplest cases, only the following information is relevant at the system level:

- Baud rate of the transfer.
- Timestamp of the end of the transfer.
- Data transferred.

So instead of modelling each bit transfer at each clock tick, we can abstract all of these events by a single transaction transferring the data in a single event with the timing calculated by taking the baud rate period multiplied by the number of bits transferred.

Bus Abstraction

Bus transactions are abstracted by observable events from a master or a slave point of view. For example, in an internal bus transaction, only the following events are observable from a master requesting a bus transaction:

- Getting access to the bus (grant event)

- If required, completion of intermediate phases of the transaction such as in the case of a burst access (partial completion event)
- Completing the transaction (complete event)

Independent of these events associated with the bus protocol, the data can be transferred in a single step.

The same kind of abstraction are used in the CAN bus model where a specific "Protocol Engine" model is in charge of resolving priority questions build from message configuration. Only observable events are modelled such as when a frame is accepted, or when a frame has been successfully received.

Efficient SystemC Modelling

Since SystemC modelling style is event driven, efficiency in implementing a peripheral is obtained by limiting the number of events generated by the model. This is mainly done by combining modelling techniques like:

- Event prediction. This means using the current context to predict an event which will happen in the future. Of course, a change in the peripheral context (for example a change in the registers of the peripheral, or a modification of the reference clock period) must be taken into account and future events must be programmed again.
- Context storing and computation. This means storing into the peripheral model any information necessary to update the peripheral state when required (e.g. when the user needs it). In our timer example above, we will see that storing a timestamp can save hundreds of events without losing accuracy.

Another important issue, more related to SystemC implementation than an abstraction principle, is: since a SC_THREAD is implemented as a thread in the OSCI kernel, this kind of construct implies an overhead due to task switching. In most cases, this can be avoided using a "callback" modelling approach by combining a SC_METHOD and an internal event which is used to trigger the callback.

A simple timer model for counting events

In this paragraph, we will see how the above general principles can be implemented in a simple timer and how this methodology offers dynamic clocking capabilities "for free". Then, we will see that this capability can be used to count and measure events, where the input could be a clock signal with a variable frequency (common case in automotive applications).

Timer external interface

The simple timer interface will contain:

- A bus interface composed of a Bus port and a Bus Clock port. The bus interface will be connected to the peripheral bus accessible from the Bus ports of the e200z6 core.
- A "Reset" input.
- A "TimerClock" input used to provide the clock reference used by the timer.
- A "TimerInt" output. This port is a logical output used to request an interrupt. This output is typically connected to the Interrupt controller.

Timer registers

Remark: In this paper, we will not talk about the particular TLM protocol used to access internal registers. Simulation Tools such as VaST's support several bus protocols and automatically add a Cycle-Accurate Bus transactor to support PV (Programmer View) untimed models. Here we will just suppose that the TLM protocol enables a bus access for reading or writing internal registers. More details can be found in [4].

Our simple timer will use 5 registers:

- GTR (Global Time Register) contains the current number of clock ticks elapsed.
- OS_PERIOD contains the clock ticks target value for generating an interrupt.
- INT_ENABLE is used to enable interrupts when GTR = OS_PERIOD.
- INT_FLAG reflects the TimerInt output status. This register is used to clear the interrupt.
- TIMER_ENABLE is used to enable/disable the timer.

Timer behaviour modelling

The specification of this very simple timer is the following:
1 – When a reset occurs, all register values are set to zero and the TimerInt output is set to zero.
2 – When the timer is enabled, the GTR register can be read from the bus interface to get the current number of edges since the latest interrupt.
3 – An interrupt is raised when GTR = OS_PERIOD and INT_ENABLE = 1. Raising an interrupt means writing 1 on the TimerInt logical output and in the INT_FLAG register.
4 – TimerInt output can be cleared by writing zero in the INT_FLAG register.
5 – Dynamic clocking must be taken into account.

Even with such a simple specification, an inefficient modelling style could lead to poor simulation performance. For example, modelling this timer using a pulsing clock used to increment the GTR register could imply the simulation speed to be in the KIPS (Kilo Instruction Per Second) range instead of the MIPS range, even if the e200z6 core model can run at more than 60 MIPS. Furthermore, it is useless trying to keep a continuously updated value in the GTR register, especially if the user (ie the software) does not read it.

We will model the behaviour using the three principles described above:

- No SC_THREAD. Our behaviour will use the following SC_METHOD:
  - The Reset method, executed when the reset input port changes
  - The ClockChanged method, executed when the clock port value changes. Since the clock will be abstracted by its period, this method will be triggered only when the period of the clock changes.
  - A MatchEvent method. This one will be triggered using an internal event used for event prediction.
- Context storing and computation: As explained above, we don't have to update the current timer value (GTR register). We can just store a timestamp associated with the value in the GTR register. When the user reads the register, we can calculate the proper GTR value by adding the number of clock ticks corresponding to the time elapsed from the stored GTR timestamp and the current time.

```
void cTimer::UpdateCurrentTime(void) {
  if (mTIMER_ENABLE) {
    sc_time NowTime = sc_time_stamp();
    mGTR += (int)((NowTime - mTimeStamp)
         /mClockPeriod);
    mTimeStamp = NowTime;
  }
}
```

- Event Prediction: when the GTR value is consistent, we can easily predict when the interrupt need to be raised. If the timer is enabled, it will occur in OS_PERIOD – GTR ticks; so using the clock period, we can easily schedule the interrupt event in the future. Of course, if the context changes (clock period change, register configuration changes…), this event needs to be unscheduled and rescheduled.

```
void cTimer::SetupNextMatch(void) {
  mTimerEvent.cancel();
  if (TIMER_ENABLE) {
    int matchTick = mOS_PERIOD-mGTR;
    if (matchTick>=0)
      mTimerEvent.notify(
       (double)(matchTick*mClockPeriod.value()),
        SC_PS
      );
    }
}
```

If nothing changes in the context, the mTimerEvent will occur at the scheduled time. This one will trigger the TimerEvent SC_METHOD where we write 1 on the TimerInt output port and in the TIMER_INT register.

Dynamic clocking behaviour

For dynamic clocking, we must take into account that the frequency of the input clock may change. When this occurs, we need to:

- Update the current GTR value: since the ClockPeriod is used to compute the GTR value, the ClockPeriod must be constant since the GTR value and timestamp were last stored.
- Reschedule the match event: the prediction has been done with the previous period value.

```
void cTimer::ClockChanged(void) {
  UpdateCurrentTime();
  mClockPeriod=mTimerClockPort.read() *
      sc_get_time_resolution();
  SetupNextMatch();
}
```

Event counting

The timer described above can now be used to generate interrupts at a frequency multiple of the frequency of the input. This control is typical from powertrain applications to build internal system clock from sensor information.

The missing part is the "testbench" generation, providing the clock period evolution scenario. This can typically done using a system level modelling tool like Matlab / Simulink or Saber connected to the virtual platform. This test bench will be sampled at a rate defined by the synchronisation period between the system simulation environment and the virtual platform simulation kernel. The input event frequency will be converted to a clock period in picoseconds (VaST kernel resolution) and passed to the timer defined above.

3. Platform Presentation

The previous chapter has presented a modelling methodology for reaching the TLM level of abstraction by reducing the number of events to the minimum required from an external point of view. This methodology has been illustrated on a very simple example but can be applied to complex peripheral modelling. This chapter will describe a platform made for evaluating the usage of virtual platforms for powertrain application development.

Digital core description

The goal of this platform is to enable early software development on a system based on a Freescale MPC5554 microcontroller (Copperhead) and a customized ASIC design by CONTINENTAL,

connected through a serial link to the copperhead, with the DSPI peripheral.

The need to have a SPI link fully functional implies some other peripherals to be added to the platform:

- The MPC5554 DSPI itself [2]. Only the SPI behaviour will be modelled. The interface of the DSPI will define 3 32 bit signals to abstract the communication :
    - SDI signal will contain all the input bits of a reception
    - SDO signal will contain all the output bits of a transmission
    - SCI_CONTROL will be used by the master to manage the transaction, provide the cycle accurate timing of the complete transaction, and encode configuration information of the DSPI [2] (chip select, control bit status…).

  No buffering mechanism will be modelled since the eDMA will be used to automatically transfer input and output frames from the memory to the DSPI registers.
- The MPC5554 eDMA. The model will be restricted to only 2 channels (32 and 33) which are used to automatically transfer data to and from the DSPI [2].
- The simple timer presented in the previous chapter. Our software is reactive and interrupt driven. This simple timer will be used to generate interrupts at a fixed or dynamic rate. In a full copperhead model, this should be managed by the eTPU.
- The MPC5554 INTC [2]. This model will implement software interrupts and will be connected to the DSPI, eDMA, and timer.

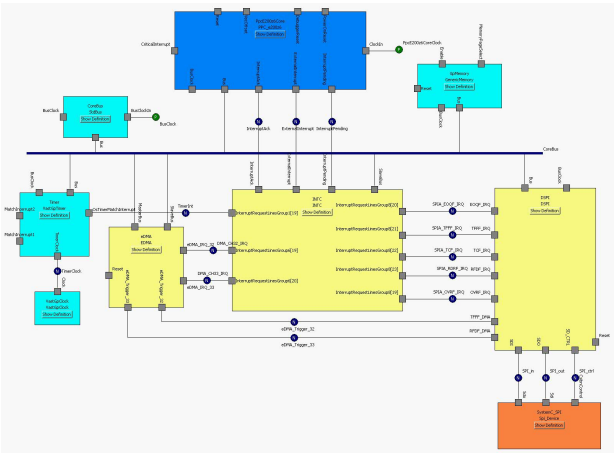

Figure 1: MPC5554 simplified platform

The platform will be based on the VaST e200z6 core model and the interconnection will be simplified as a single bus. This is a source of inaccuracy at the platform level since the MPC5554 uses a crossbar which can support simultaneous DMA transfers and e200z6 core memory accesses.

ASIC description

Connected to the simplified model of the MPC5554, a power stage output driver is modelled in SystemC/TLM style. As depicted by the hardware schematic in Figure 2, it is a simple component controlled in slave mode through a SPI interface of the microcontroller. The commands transmitted from the master as serial data are converted to parallel digital outputs. The peripheral sends back the diagnosis status of the outputs. Detection of open circuit or short circuit is performed by the peripheral from internal current analysis.

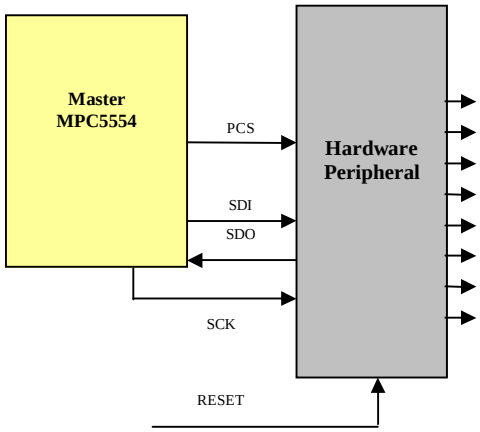

Figure 2: Description of Power stage driver

As explained in digital core description, the SPI interface is abstracted with three signals: sc_in SDO (for output transmission), sc_out SDI (for input transmission) and sc_inout SCI_CONTROL. SC_METHOD of the peripheral model is sensitive to SCI_CONTROL, and manages the transaction by decoding configuration of the cycle accuracy transmission (chip select, first bit transferred, last bit transferred). The diagnosis part is abstracted by a text file. Data from the text file tagged with a transmission date, are copied in the SDO signal and transmitted at the correct time using engine simulation services.

This ASIC model has been validated out of the context of virtual platform with a software bench implemented in systemC.

Software description

An industrial software of engine management system is used and adapted to perform the evaluation. The code size of the original software is 8 MB executed on a MPC5554 cadenced at 80MHz. For peripherals, all eTPU channels are configured, software controls communication interfaces (FlexCAN, DPSI, eSCI) and simple or complex

input/output interfaces (EMIOS, eQADC, Port of SIU).

Two software versions are adapted to cover the different objectives of the study: one for capability to debug software and another one for performance and accuracy measurement.

For the first one, the software is depopulated to remove all access to the MPC5554 peripherals not implemented in the virtual platform (see digital core description above).

The second software is build by removing the code for all the peripherals control and the eTPU code. Performance and accuracy are measured on core and internal memory accesses (flash and RAM). Software is executed from flash into the e200z6 core using a standard cache configuration of 64KB.

Debug capabilities of the MPC5554 platform

One of the main advantages of the virtual platform is the capability to offer to the end-user much more debug features than the real hardware. This includes complete visibility inside the virtual hardware for purposes such as analysing the cache behaviour (hits / misses, cache evictions) of a specific software routine. It allows optimizing aspects of software like interrupt handlers or high performance routines (to reduce interrupt latency or minimize pipeline bubbles in assembler). All the examples above can be performed by mixing a “user friendly” software debug environment and an extended observability of the hardware, especially the core micro-architecture.

VaST cores are interfaced to industry software debuggers like T32 from Lauterbach and a tool called Metrix enables getting and correlating all the events happening in the simulation kernel. VCD traces (Value Change Dump) can also be easily generated to facilitate analysis of behaviour such as latencies inside the hardware.

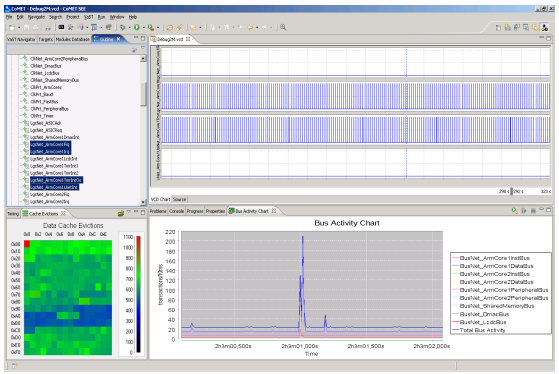

Figure 3: Example of hardware analysis with Metrix tool (Cache, VCD, Bus monitor)

Using Lauterbach T32 or VaST MFA (Metrix Function Analyser) tools, profiling trees can be obtained in a couple of minutes to identify software bottlenecks.

4. Results

The results of the study are based on experiment performed inside Continental Powertrain Division. With detailed criteria functionality, accuracy and performance presented below, we aim to identify if virtual platform could be adequate to powertrain software context.

Functionality

Functional accuracy has been demonstrated using Lauterbach T32 debugger on the virtual platform. Debugging environment was possible with the same user interface and behaviour as the one used with real target. In respect to limitation of the DLL and service introduced in the e200z6 core model, standard features as break point, trace, patch of code, display of microcontroller and peripheral registers were possible. SPI data interface was also captured using text file with time stamp value in accordance with real peripheral behaviour using observation precision programmed as 1 ms of precision (enough for functional observation of communication speed).

Performance

Simulation performance was calculated from an execution loop of the software performance measured to 1 second on the target. On a Laptop equipped with Dual Core T2400 running at 1.83GHz and 1 GB memory, simulation time for loop execution was measured around 3.5 seconds. A 15% overhead has been measured when the simulator is driven by Lauterbach T32. The table bellow also gives the results of peripheral integration test software intensively reading and writing to the SPI device using the eDMA.

Configuration	Virtual HW / Real HW ratio	MIPS (s)
Peripherals test software on T2400 with 1Go of Memory	1.7	60 MIPS
Continental SW on T2400 with 1Go of Memory	3.5	29 MIPS
Continental SW on T2400 with 1Go of Memory under Lauterbach T32	4.0	25 MIPS

Figure 4: Measurement for speed

Note: A recommended configuration is at least 1GB of memory. As an example, the Continental SW running on an Intel Pentium M at 1.6GHz with 512

MB leads to a Virtual HW / Real HW ratio of 15 (e.g. 7 MIPS).

Accuracy

Measurement of accuracy was calculated by comparison of number of cycles between real target and simulation target. Since the memory model in this platform was a very simple one, we needed to introduce an adjustment factor that was estimated to a worst case value of 1.34. It was calculated from real wait state figures in regard of the flash memory configuration, and using cache hit / miss ratio for the number of memory read and write operations of the software. The accuracy was finally estimated to 92.3% as shown below in Figure 5.

	Frequency	Cycle number
Target	80 MHz	10502131
Simulator (Simple Memory model)	80 MHz	7232456
Simulator corrected with ratio (1,34)	80 MHz	9691491
Accuracy Error		-7.7 %

Figure 5: Measurement for accuracy

As the crossbar switch configuration and memory models were not accurate in the virtual model, the correlation with measurement value is sufficient. Indeed in parallel to this experiment, complete model of MPC5554 was developed and is depicted below in Figure 6.

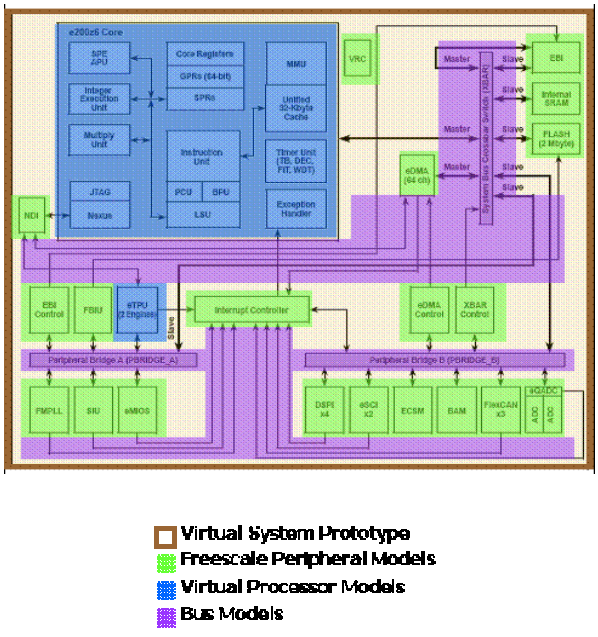

Figure 6: Complete MPC5554 VaST Model

5. Conclusion

The objective of the study has been achieved by demonstrating the use of TLM modelling style with the VaST environment to build a virtual platform with acceptable performance and accuracy.

Such virtual platform including ASIC modelling facilitates debugging software of an engine control application. It allows to integrate ASIC models and to close the HW/SW development loop for device drivers. It also provides a solution for designing ASICs by starting with an abstract model implemented in SystemC/TLM. This abstract model could be considered as the golden reference model and specification requirement for hardware development in a RTL modelling style.

In order to face the complexity of industrial application such as engine control software, a scaling demonstration has to be conducted to prove that acceptable overall simulation performance can be obtained. This will be achieved using a complete model of a MPC5554 connected with several ASICs and stimulation bench to activate all software functionality.

Such methods and tools could also be used for architecture exploration of new product family. Anyway, fundamental condition is the availability of the hardware models in advance of silicon delivery. A typical use case, facing potential performance issue, is the foreseen introduction of dual core processor into next generation of automotive embedded systems.

6. Acknowledgement

This work was supported by active contribution of Continental colleagues Christophe Marigo and Norbert Mignot. We wish to acknowledge them for their work and positive exchange during this study.

7. References

[1] OSCI http:\\www.osci.org

[2] Freescale: "*MPC5553/MPC5554 Microcontroller Reference Manual*", MPC5553/4RM, Rev.3.1, 10/2005

[3] M. Schnieringer, K. Brand: " *SystemC: Key modeling concepts besides TLM to boost your simulation performance*", IP SOC, Grenoble - France, 11/2007.

[4] F. Ghenassia (Ed.): *"Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems"*, Springer, 11/2005.

8. Glossary

eDMA: Enhanced Direct Memory Access.
eTPU: Enhanced Time Processing Unit.
INTC: Interrupt Controller.
SPI: Serial Peripheral Interface
TLM: Transaction Layer Modelling.
VCD: Value Change Dump.