

System-Level Partitioning Using Mission-Level Design Tool for Electronic Valve Application

K. Manon McNair, Matthias Zens
Infineon Technologies AG

Horst Salzwedel
Ilmenau Technical University

Copyright © 2001 Society of Automotive Engineers, Inc.

ABSTRACT

In defining innovative and cost-effective chip sets for future automotive applications, system architects need high-level tools that allow them to rapidly determine the best silicon partitioning for a given application in terms of system performance as well as cost. The tool needs to be flexible, modular, and swift such that the system designer can perform abstract simulation iterations quickly for various functional partitioning scenarios, without requiring excessive computer resources. The tool must also be portable and adaptable to provide a simulation environment suitable to systems- or car-manufacturers for in-depth applications simulation and architecture assessment.

The semiconductor component definition process using such a "mission-level" design tool for the automotive application electronic valve will be demonstrated. Methods for the analysis of electronic valve control system architectures using mission-level simulation will be developed. Simulation results and corresponding analysis of electromagnetic valve control performance within two primary types of system architectures, centralized and mechatronic, will be provided. Control algorithms using various sampling frequencies and accuracies for position and current data acquisition are included within simulation. Analysis will determine processing power needed to effectively follow and manage current dynamic and valve position. Timing diagrams of communication will characterize bus traffic in order to evaluate speed of communication busses between power devices and valve microcontroller, as well as between the VCU and ECU. The type of instructions to be processed by the valve control algorithm to deliver the appropriate output to the electromagnetic valve will be determined so that an appropriate microprocessor can be selected. A semiconductor architecture recommendation based upon simulation results and analysis, in addition to cost

considerations, will identify a best-fit silicon strategy for the end application.

INTRODUCTION

The amount of electronics to control a camless engine using electromagnetic valve actuators is considerable. Although there are cost offsets to be gained with the removal of components such as the camshaft, the electronic throttle, and the exhaust gas recirculation valve, an electronic valve-equipped engine must have costs comparable to those of a mechanically driven engine to be marketable. With a demonstrated reduction in fuel consumption of up to 20%¹, the electromagnetic valve vehicle would achieve fuel savings of an estimated \$1000 over its life. Nonetheless, these savings are deemed to be insufficient incentive for consumers to pay a higher price due to increased electronics content for an electronic valve vehicle, despite the increasingly stringent governmental standards for fuel economy and exhaust gas emissions. Consequently, the costs of the additional electronics needed to control and actuate the valves must be optimized to meet this comparable price target. Optimizing these costs means optimizing the architecture. For example, as illustrated in Fig. 1, existing off-the-shelf chips can manage the actuation and control of an electromagnetic valve.

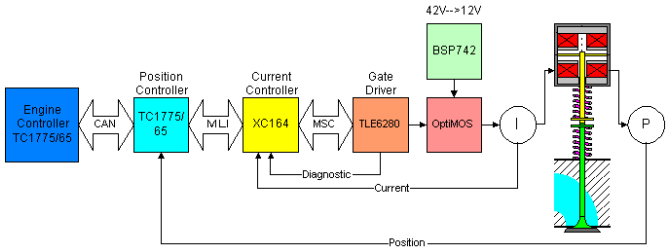


Figure 1: Valve Control using Existing Chip Set

However, these chips may contain modules, and thus additional silicon area, unnecessary for this control system. A new chip set can be tailored according to the typical instruction set, code size, communication rate, and other performance parameters characteristic of the system. An abstract system-level design tool can aid in optimizing chip partitioning for the given application by providing semiconductor concept engineers the flexibility to define the control and actuation system using object-oriented blocks. The chip architecture in Fig. 1 is described functionally in Fig. 2, without the boundaries of the semiconductor components. The mission then becomes to evaluate the different architectural scenarios and peripheral needs for the use case of electronic valve.

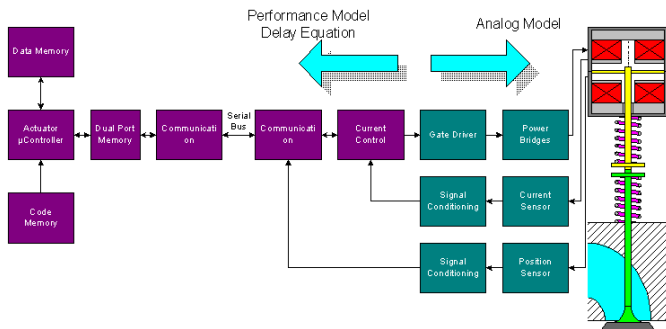


Figure 2: Functional Blocks for Valve Control

FUNCTIONAL MODEL

SIMULATION DOMAINS

To describe functional blocks of the proposed system architecture, these functions must be mapped into specialized design environments, known as “domains”, of the mission-level simulation tool, MLDesigner², as listed below. These domains can be mixed and nested and thereby allow more than silicon functionality to be modeled. The tool has the advantage of being able to model the complete system, including the actuator.

1. Discrete event domain (DE): The DE Domain models architectural performance, e.g., resource contention of CPU, bus, disk, etc. It also models network topology and protocols, e.g., traffic flows, arbitration, buffers, etc.
2. Synchronous and Dynamic Data Flow (SDF, DDF): SDF and DDF are used to describe wireless and broadband communication links, multimedia compression algorithms or DSP operations.
3. Finite State Machine (FSM): This domain models controllers and protocols.
4. Continuous Time Discrete Event Domain (CTDE): This domain is used to handle analog components.

VALVE ACTUATOR MODEL

The electronic valve actuator considered in this simulation is an electromagnetically actuated valve,

similar to that illustrated in Figures 1 and 2. The actuator has upper and lower magnetic coils, between which an anchor plate swings. The anchor plate has a pre-set tension determined by the spring surrounding the anchor shaft. The anchor shaft is separated by a small gap from the valve, which is also surrounded by a spring. Both an electromagnetic model and a spring-mass system model therefore describe the complete actuator model.

Electromagnetic Model

A $\frac{3}{4}$ H-Bridge drives each coil in the actuator, with a diode completing the fourth leg of the bridge, as illustrated in Fig. 3. At start-up, the upper MOSFET C and lower right MOSFET B are turned on at maximum duty cycle. This is the “on” phase, in which +42 V are applied across the coil. When the desired current is achieved, the lower left MOSFET A alternates with the upper MOSFET C to regulate the current. While A and B are on and C is off, zero volts are applied to the valve, and this state is known as the free-wheeling phase. In order to “brake” the valve, the MOSFET C and MOSFET B are turned off, MOSFET A remains on, and the current is reversed as -42 V are applied across the coil.

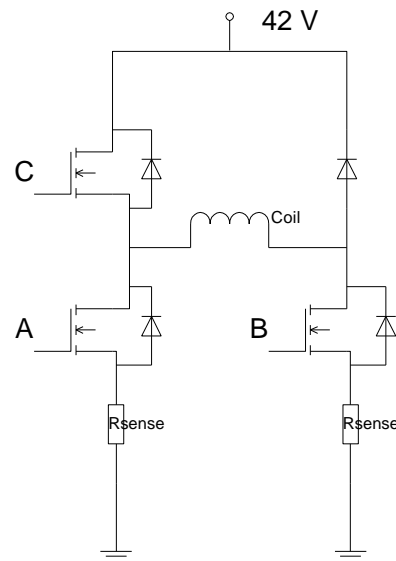


Figure 3: $\frac{3}{4}$ H-Bridge Configuration to Drive One Actuator Coil.

This diagram is further simplified in Fig. 4 for purposes of the model by treating the MOSFETs as switches. Since MOSFETs C and A will not be enabled simultaneously, the PWM signals controlling switches a and c can be considered as complements of one another. Thus, a complete actuator with two coils requires four PWM channels rather than six to control the H-bridges. The control phases can be described as three cases: on, free-wheeling, and reverse. In these cases, discrete voltages of +42 V, 0 V, and -42 V, respectively, will be applied to the coil. These voltages become the inputs to the electromagnetic model.

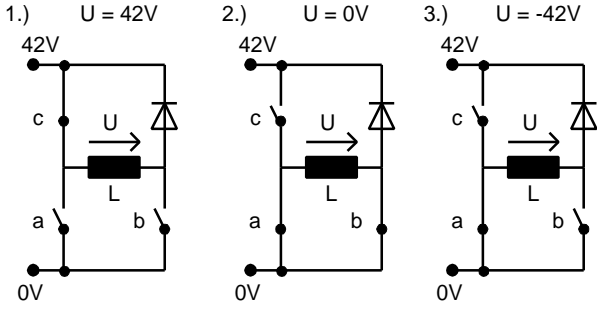


Figure 4: Three Cases of Valve Control

The magnetic flux, Ψ , through a coil depends on the current flowing through the coil, I_L , and the magnetic resistance, R_{MAG} . R_{MAG} depends upon the distance between the anchor and the iron core, or coil. The induced voltage, U , in the coil is dependent upon the speed, V , of the anchor, and the change in the coil current, I_L . The inductances L_i and L_δ influence this dependency and are also determined by I_L and the air gap between the anchor and coil. The implementation of the differential inductances describes only a single-point linearization of the model, from which the equations to determine the coil current, I_L , and the actual current, I , can be transformed in the complex variable domain. By using Laplace transformations³, the coil current and actual current can be defined as a sum of PT1 elements by expanding the differential equations for the currents into partial fractions. Equation 1 shows the PT1 Element, and equations 2 and 3 are the resulting partial fraction expansions, with T defined in eq. 4.

$$PT1(inSignal, inFactorA, inFactorB) =$$

$$[1] \quad \frac{inFactorA}{1 + inFactorB \cdot s} \cdot inSignal$$

[2]

$$I(s) = \frac{T_w}{R(T + T_w)} U(s) + \frac{T}{1 + (T + T_w)s} U(s) - \frac{L_d T}{L_i} \frac{V(s)}{1 + (T + T_w)s}$$

$$[3] \quad I_L(s) = \frac{1}{R} \frac{U(s)}{1 + (T + T_w)s} - \frac{(T + T_w) \frac{L_d}{L_i}}{1 + (T + T_w)s} V(s)$$

$$[4] \quad T = \frac{L_i}{R}$$

L_i , L_δ , and T_w are determined from a set of characteristic curves³, as well as the anchor force. The Eddy current resistance, R_w , depends upon whether the magnetic field is increasing or decreasing, whereby, according to the sign of the current direction, there are two different Eddy current time constants for T_w , one increasing and one decreasing.

The current, I , is that measured by the shunt resistors depicted in fig. 3. The coil current, I_L , which contributes to the generation of the force on the anchor, is smaller than I in the simulation. The difference between these two currents is considered to be the Eddy current³. These differential equations 2 and 3 are implemented as shown in Fig. 5 in the CTDE domain of the mission-level design tool, with the help of the PT1 structure in eq. 1.

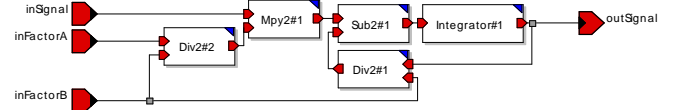


Figure 5: Implementation of the PT1 element in CTDE Domain.

Mechanical Model

The mechanical model consists of a spring-mass oscillator with a speed-dependent friction force and speed-independent friction³. The differential equation for the spring-mass system is then:

$$[5] \quad m\ddot{x} + d\dot{x} + cx + F_C |\dot{x}| = F_E$$

Equation 5 does not describe the displacement limitation of the anchor due to its impact against the magnetic coil. In order to create a suitable structure for its implementation in CTDE, it is recommended to transform the system into a first-order differential equation system, as given in Equations 6 and 7.

$$[6] \quad \dot{x}_0 = x_1$$

$$[7] \quad \dot{x}_1 = -\frac{d}{m} x_1 - \frac{c}{m} x_0 - \frac{F_C}{m} |x_1| + F_E$$

where x_0 represents the anchor position and x_1 the speed.

Figure 6 depicts the spring-mass model implemented in CTDE.

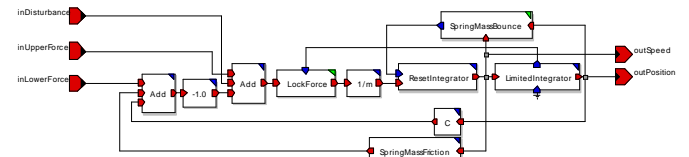


Figure 6: Spring-Mass System in CTDE Domain.

There are two feedback loops that consider the position limits and rebound of the anchor in the model. In the first loop, the primitive *SpringMassBounce* monitors the position and sets the limits of the integrators for speed, *ResetIntegrator*, and position, *LimitedIntegrator*, once the anchor has reached the coil, thereby accounting for anchor bounce. The surrounding blocks calculate the corresponding force. Part of the kinetic energy of the

anchor will be transformed into heat as it contacts the coil magnet, so the recoil speed of the anchor will be less than that of its impact. There is also a gap between the anchor shaft and the valve itself such that, when the valve is seated, the anchor continues to move upward until it touches the upper magnet. This gap is nonetheless relatively small and is thus not considered in the mechanical model. Block *LimitedIntegrator* saturates if the anchor remains in the maximum position; in which case the speed must also be held at 0. *LimitedIntegrator* controls, through the second feedback loop, the anchor impact force and acceleration in the integrator for speed, *ResetIntegrator*, via block *LockForce*. This latter block contains a variable force limit, which adjusts itself after receiving the position value and resets to zero the forces that *ResetIntegrator* would otherwise use to calculate the speed in the wrong direction. To validate this model, the scenario depicted in Fig. 7 was created, in which a square-wave force is applied to the anchor. The output of the block *LockForce* is also portrayed, where it can be seen that the calculated force decreases as the anchor swings toward the coil. As the anchor bounces against the coil, the force remains nearly stable because the position change is relatively small. Once the anchor position is fixed against the coil, *LockForce* resets its output to zero.

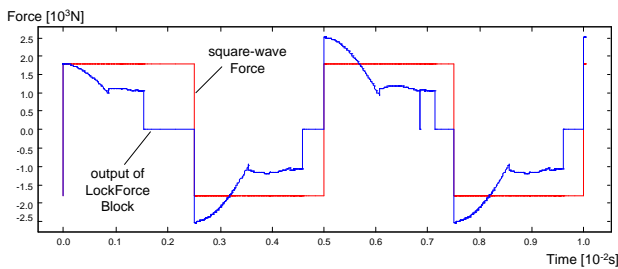


Figure 7: Trigger Input and Resulting Output of *LockForce*.

Figures 8 and 9 display the position and speed values related to Fig. 7. If the anchor has reached its position limit, the previous force is given a value of zero so that the acceleration and speed values also result in zero.

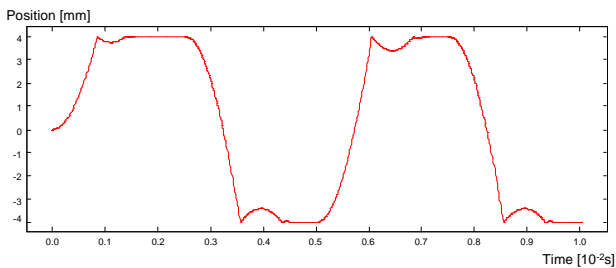


Figure 8: Position Output of *LimitedIntegrator*.

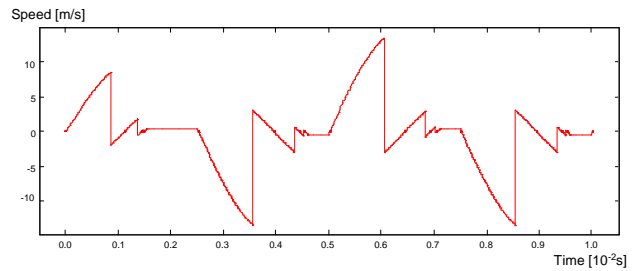


Figure 9: Speed Output of *ResetIntegrator*.

ARCHITECTURAL ANALYSIS METHODS

The Total System Model is described in the CTDE domain and consists of a Functional System Model, Environmental Model and Architectural Model, as well as driver and evaluation models. At the uppermost level, the blocks *Actuator* and *Controller Interface* define the system model portrayed in fig. 10.

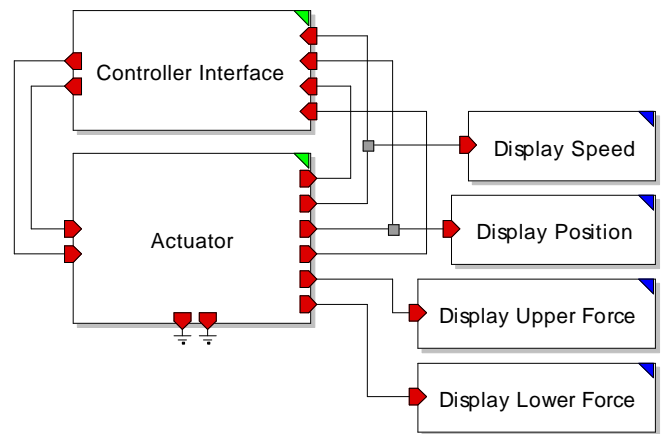


Figure 10: Total System Model in CTDE Domain.

The associated blocks belong to the evaluation model and allow the actuator status to be observed. The valve actuator model *Actuator*, consisting of electromagnetic and mechanical models, has already been described. *Controller Interface* contains the functional model, architectural model, as well as driver model, and the remainder of the evaluation/rating model. This latter block is shown in greater detail in fig. 11 and contains the interface between the Environmental Model and the control system.

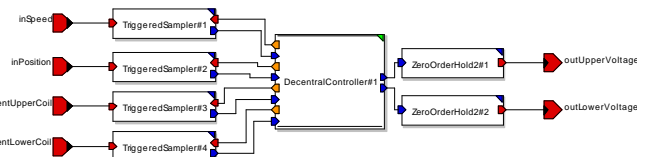


Figure 11: Interface between Continuous and Discrete Domains.

Within these blocks is a DE module that contains the actuator controller. The module dispatches actuator events at specific times to the blocks *TriggeredSampler*, which sample the status of the application environment. Similarly the primitive *ZeroOrderHold* sets the voltages at the upper and lower actuator coils. Due to the number of

architecture partitioning variations and associated control methods possible for the application, the block *DecentralController* must have a common interface and parameters such that it can be easily exchanged for evaluating these variations.

MECHATRONIC ARCHITECTURE

Figure 12 depicts a decentralized architecture implemented as one partitioning alternative in the *DecentralController* block. The block diagram illustrates the partitioning of resources per valve, in that one VCU microprocessor manages the position for 8 valves, one CCU manages the current for 2 valves, and 1.5 Gate Drivers are required to drive 6 MOSFETs. The MLI provides a fast inter-processor communication interface between the VCU and CCU. The boxes above and below the VCU block depict the number of bits, or packet size, transmitted for current and position control. The CCU module is enhanced from the XC164 to include two CAPCOM6 modules to provide enough PWM outputs for the Gate Drivers. The CCU model also includes two CAPCOM1/2 modules to provide sufficient position sample inputs.

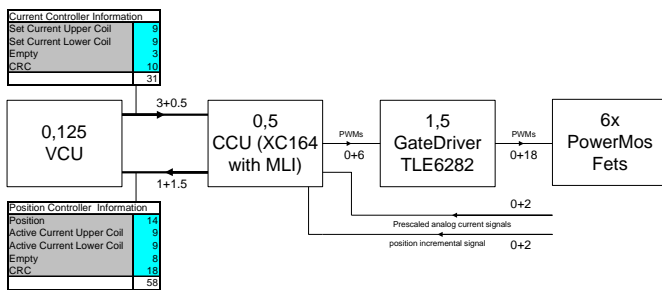


Figure 12: Valve Control using Decentralized Architecture.

CENTRALIZED ARCHITECTURE

The centralized architecture in fig. 13 differs from the decentralized in that there is no separate current control module. The position and current control are performed in a single processing unit. Thus the centralized architecture block diagram will have no *Current Controller* block. Instead, the current sense inputs will be fed directly to the Valve Controller AD converter and the position sense inputs to the PDL sub-module within the timer block, GPTA, of the TriCore. The simulation assumes the position sensor has a digital output which is sent to the PDL.

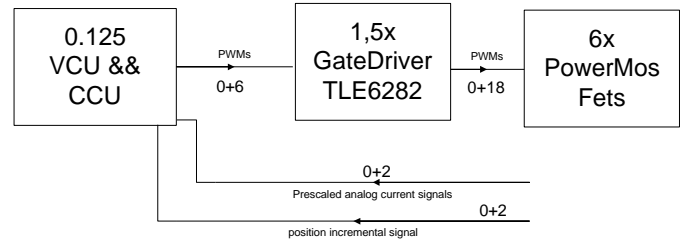


Figure 13: Valve Control using Centralized Architecture

VALVE CONTROL ALGORITHMS

The valve is managed by controlling the current as well as the position. The current of the upper and lower coils are controlled through an inner loop, while the position is controlled with an outer loop that feeds the current loop with the desired control current.

CURRENT CONTROL

PI Controller

A PI controller is a simple and robust way to control the current. A PID controller was also considered, but the differential portion may become quickly unstable. Since the current control algorithm needs to be implemented discretely within an actual microcontroller for the model to include an accurate execution time, a better implementation would be to deduce the system in z . First a Laplace transformation of the system as described in Equation 8 is performed, and then an approximation in z is made, where T is the increment. The transfer function in Laplace for the PI controller is

$$[8] G_R = K_R \frac{1 + T_R s}{s} = \frac{K_R}{s} + K_R T_R$$

As the PI controller can produce only limited outputs, the integral portion cannot increase without limit because with a fixed offset it otherwise achieves a very high value which is then difficult to reduce. If the controller limit is not reached, the integral portion must be fixed. Thus the control equation is described as a sum, in which each addend can be transformed and the outputs of the two filters summed up in one filter. The filter output is described as

$$[9] Y = Y_1(e) + Y_2(e),$$

where e is the offset/error. According to eq. 8, Y_2 is simply an amplification of the error of $K_R T_R$. Applying a rectangle approximation of s in eq. Y_1 yields

$$[10] Y_1 = K_R T e + Y_1 z^{-1}$$

When the anchor approaches the coil and the air gap becomes smaller, the current rise time is on the order of

a few milliseconds³. The time for the anchor to move from one coil to the other is in the range of 2-3.5 msec. Thus the electrical system is quite sluggish with respect to the mechanical system, making the current regulation more difficult. In order to optimize the system energy usage, it is necessary to minimize the amount of current required to hold the anchor against the coil. The more accurate the current control algorithm, the less overshoot in the actual current, resulting in less overall current-, and thus energy-, consumption.

To generate PWM signal inputs for a $\frac{3}{4}$ H-Bridge to drive the actuator, the PI regulator output must be transformed into a duty cycle equivalent. For positive currents, the regulator output will vary between 0V and 42V, and for negative currents it will vary between 0V and -42V. The PWM signal will be generated such that the average voltage value will be applied to the actuator.

POSITION CONTROL

To achieve a “soft landing” of the valve, the anchor speed must be held as close to 0 m/sec as possible when it reaches the upper coil. It is necessary to establish the hold current in the upper coil well before the anchor reaches this upper limit; otherwise the anchor will not be held against the coil. This speed can be managed by compensating for the additional energy absorption as the anchor approaches the coil. The energy of the anchor is described in eq. 11,

$$[11] E = \frac{m}{2} V_{Ist}^2 + \frac{c}{2} x_{Ist}^2.$$

The losses that occur from friction and electromagnetic disturbance as the anchor swings from the lower to the upper coil can be easily considered by using energy control as a method for position control. The energy losses as the anchor is released from the lower coil must also be considered. Once the anchor is at the upper position limit, the energy of the system ideally corresponds to the potential energy of the spring, in order to ensure that the anchor speed is indeed zero. Because the hold current increases while the anchor approaches the upper coil such that upon impact, the resulting magnetic force holds the anchor in place, the system energy continuously increases. As a result, the energy curve is parabolic, dipping near the midpoint between the two coils. Refer to fig. 27.

INSTRUCTION RESOURCE MODEL

The goal of the Instruction Resource Model (IRM) is to link the system function to the resource that will perform its execution. The functional model description generally consists of a hierarchical structure within the simulation tool. The data flow can be divided and the different functional elements treated as inputs. These elements can then execute another function independently from

one another. Synchronization points are determined from the data dependency and availability of operands. To execute the function using a particular resource, its execution within the functional element must be scheduled and the limited resources of the system mapped. The component models of IRM realize the special case of the existence of only one resource in the system, so the mapping of the resources onto a shared resource architecture cannot be automatically realized. However, several resources can be created by replicating IRM instances. The scheduling is used in the case of only one resource linearly running the operations from the data flow model. The basic idea of the IRM is to centralize the execution of a function in a server, as described according to the data flow. The functional elements in the data flow model send a request to the server for a function execution. The server returns a confirmation when the function has been executed by the resource. The function is therefore modeled in a functional element and is executed once the confirmation is received. Since the server has a request buffer, it can differentiate the run-time of the functional element from that of the server. To make the calculation of the run-time using the server within the IRM flexible, the run-time is calculated by an exchangeable sub-function only directly prior to request. The behavior of the server can be changed with little effort because the run-time calculation is centralized. Function mapping is then easier because the microprocessor block can simply be exchanged and the new processor block simulated. The IRM further allows a hierarchical implementation of the function as a data flow chart. Because the number of functional elements can become substantial, a routing mechanism for the forward and reverse connections to the server has been implemented which allows modules of any level and number to be connected, without a loss of simulation performance. The implementation of a real-time system often requires offline scheduling of the task execution on the core. Tasks can be run quasi-parallel on one resource by alternating with one another. The IRM supports this task management and stores statistical data regarding their status. The IRM can be applied to describe the algorithm execution on a particular hardware as well as to simulate bus systems. This possibility for time-accurate control of the processes is advantageous for bus system modeling when the network nodes of a Shared-Medium-Net, such as a CAN bus, are defined as processes.

IRM SYSTEM EXAMPLE

Figure 14 shows a simple example for the implementation of IRM elements.

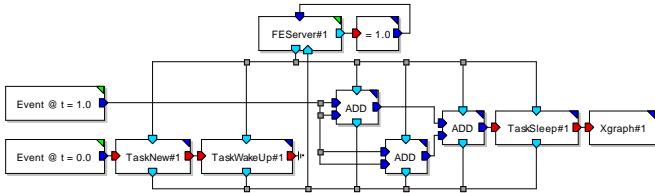


Figure 14: Simple Example for an IRM System.

The state flow for this example is drawn in fig. 15. At the start of the simulation a process with the identification number $TaskID=0$ is generated and activated. The functional elements form the three summation points portrayed as blocks *ADD*. At time $t=1.0$ these operands receive data inputs, and the server begins with the execution of the requirements of the two left functional elements. The processing time will in this case be determined from the task number generator directly to the server. After these two functional elements have had their functions sequentially performed, their results are made available to the third functional element, which in turn sends its request to the server. Once this third task is complete, the primitive *TaskSleep* assigns process $TaskID=0$ a wait-status.

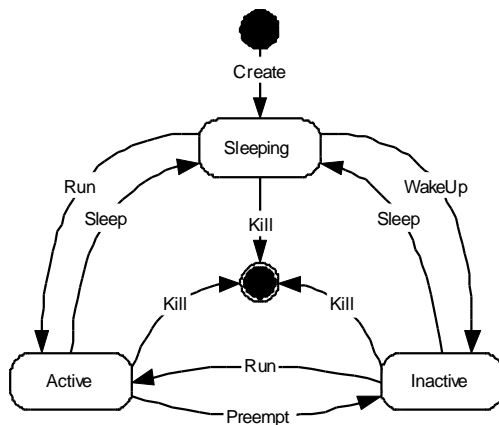


Figure 15: Task State Diagram.

FUNCTIONAL DESCRIPTION OF SERVER

The task of the server is to sequentially process the requests of the functional elements while considering the processes already running. As can be seen in fig. 16, the server consists primarily of a FIFO-type input request queue, *Server_Queue*, a processing unit, *ExecutionUnit*, an output-generating unit, *OutputUnit*, and the process controller, *TaskControl*.

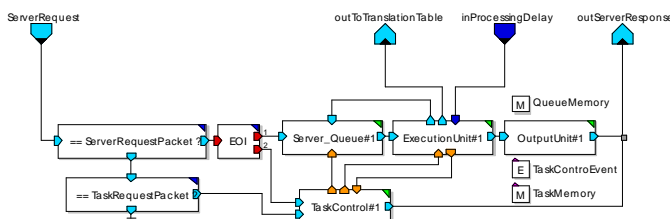


Figure 16: Block Diagram of Server.

The processing unit can process only one request at a time. The process controller transports the requests from the queue to the processing unit.

DATA STRUCTURE DEFINITION

The communication between functional elements, process control elements and the server occurs through the exchange of data particles on the network. There is a pipeline for the particles to the server and another pipeline for particles that are generated as answers by the server. These two channels could be combined, but this would mean a decrease in simulation speed, since the module would then have to additionally receive its own sent packets. How the functional elements filter out specific messages is explained later in paragraph Routing Functions and Filtering Mechanisms.

REQUEST PROCESSING

A vector of queues is realized in the module *Server_Queue* of fig. 16 through instances of the built-in model elements *GeneralQueue*. A queue is assigned to each process. If the processing unit is not processing a request, it sends a signal to the *Server_Queue* to release a request of the current process from the queue. If the packet has an execution time remaining of less than zero time units when it arrives at the execution unit, the time is calculated through a query to an external module connected to the server. The packet is delayed through an instance *InterruptDelay* and then sent to the output unit. If a task change occurs while a packet is being processed, an interrupt signal is sent, and the packet is put in the queue with the remaining process time according to the LIFO principle. If the server receives a request with a remaining process time greater than zero, the external module will not be queried, and the functional element determines its own processing time.

PROCESS MANAGEMENT

Besides managing the queue and the execution time, the module *TaskControl* must also manage the list of tasks. For this purpose, it uses the C++ primitive *TaskScheduler*. The data for a task are handled using an instance of the class *CTask*. As outlined in the UML class diagram of fig. 17, *TaskScheduler* has an object of the class *TaskMap*, which administers the list of tasks.

Since the task access rate using the identifier is high, the class *HashTable* is used to realize a set of key-string pairs and pointers to *CTask* instances. *TaskMap* uses this container class to store the *CTask* objects. *TaskControl* also defines *Events*, which can occur outside the server, such as a task change. In general, a task can take on the states in the UML diagram of fig. 17.

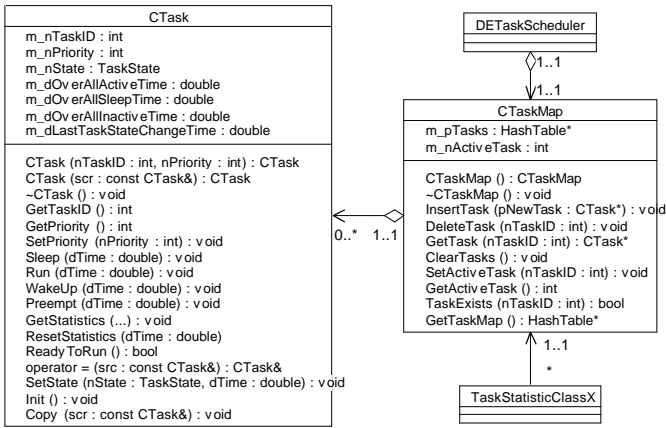


Figure 17: Process Management Classes.

The *TaskScheduler* primitive contains a preemptive mechanism that activates the highest-priority not-sleeping task and assigns the active task to the Inactive state. The displacement mechanism becomes active when the active task is set to Sleeping or a sleeping task is set to Inactive, as well as if the task priorities are changed.

RESOURCE MODEL FOR ALGORITHMS

Once the functional model of the control algorithms has been developed, the algorithm execution time must be determined and included in the system model. The algorithm must be described in the machine language of the target system hardware and executed in a processor model. The process flow is drawn in fig. 18.

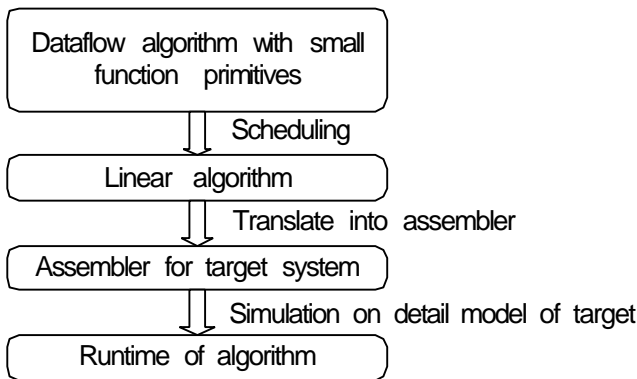


Figure 18: Flowchart for algorithm execution time determination.

Since the simulation tool uses C++, the algorithm primitives, when written in a compatible form, can be fed directly into the compiler of the target hardware. One source of error results from the integration of the C code for the different primitives into one source code. This approach has the disadvantage that it requires an exact processor model, which is beyond the scope of this simulation activity. Another possibility would be to run the algorithm in a VHDL simulator with a VHDL model of the target processor, which would then contain all determinisms; however the simulation tool does not presently have a VHDL interface. The VHDL approach would nonetheless not be advisable for the complete

system simulation because the simulation of the detailed processor model would be too time-consuming. As the process flow in fig. 19 shows, the algorithm run-time is determined from simulation on the detailed model of the target. This time is then incorporated in the overall system model. The system model is implemented using the IRM. An algorithm primitive corresponds to a functional element. The system function can be broken down into elements and the execution time determined from the detailed target model. It is important in this method of determining execution time that the total execution time is not affected by a change in the scheduling of the functional elements. The scope of the functional element must therefore be large enough that effects such as the pipeline or cache do not cause problems. The benchmark of the functional elements can be performed either directly on the target hardware or on the detailed target model. The functional elements could also meet the requirements of the target processor. In this case, a detailed component must have a run-time that is less than or equal to what the target specification requires. If the scope of the functional element is large, the error of the sequential executions will be minimized, but the flexibility and the degree of reusability are reduced. The C-code in a functional element must be optimized for the target processor such that a good execution time results, so it is worthwhile to convert floating-point operations to fixed-point, especially if the microprocessor does not have an FPU. The IRM can be used to create a library of functional elements whose execution times can be defined for several different microprocessors. The system architect can then use this library to easily analyze the performance of a given algorithm on different microprocessors.

RESOURCE MODEL FOR CURRENT CONTROL

To create a resource model for the current control algorithm described earlier, the exact processing time is needed and shall be determined following the process described in fig. 19. This resource model is implemented in a floating-point module and then in a fixed-point module. The interface for these modules is compatible so that they can be easily exchanged within the total system model, and the resulting behavioral simulation of different blocks can be compared.

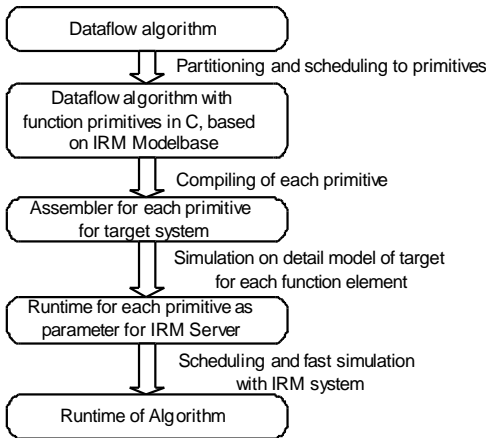


Figure 19: Process Flow for Run-Time Determination on Target Hardware.

Floating-Point Algorithm Implementation

Figure 20 displays the algorithm to control the PWM signals required to drive one actuator coil, in addition to the resource model. As illustrated in the flow diagram of fig. 19, the algorithm must first be partitioned into functional primitives. In this application, these primitives are the PI-Controller, a scaling of the controller output voltage to a range of -1.0 to 1.0, as well as a block which calculates the duty cycles of the two PWM signals. Since the function is not integrated into IRM elements, a delay block, *ProcessDelayFE*, is used in each primitive to model the delay of the server resource.

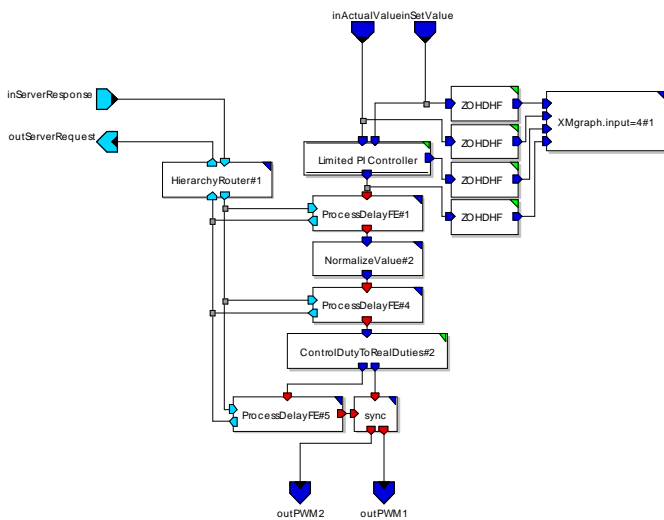


Figure 20: Floating-Point Resource Model of Current Control

The properties of the functional elements such as *FunctionID* or *FunctionClassID*, can be adjusted in *ProcessDelayFE* and thereby enable the calculation of the execution time on the server. Each algorithm primitive and a *ProcessDelayDE* block will be combined in one functional element for the run-time measurement on the target hardware. Once the run-times of the individual algorithm primitives within the detailed model are known, they can be defined in a look-up table of the

server. The other blocks in the module are part of the evaluation model, and record the signal shapes.

Fixed-Point Algorithm Implementation

In order to execute the algorithm on a decentralized architecture using the Infineon Pegasus (C166v2), a 16-bit core without FPU, the floating-point operations need to be converted to fixed-point. The algorithm must be optimized for the target hardware as well as the application scenarios. The full range of the fixed-point variable must be used in order to achieve a high precision, but if this range is exceeded, system failures can occur. This conversion process is sketched in fig. 21.

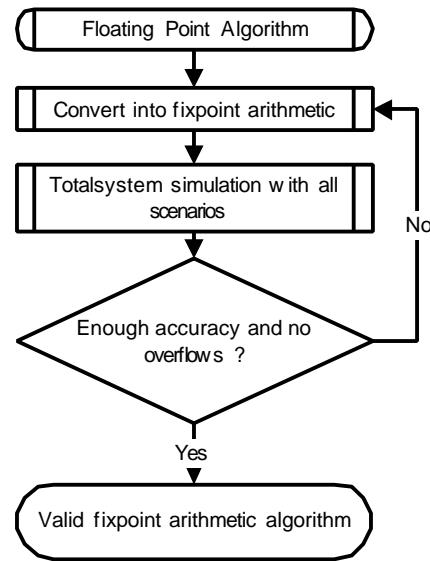


Figure 21: Floating- to Fixed-Point Conversion Process.

The preferred data type for this simulation is *Fix*, a fixed-point data type which stores values as integer bits g and decimal bits m . The total number of bits gives the word size n of the integer value, which is then run on the microprocessor. A positive floating point number f would be converted to an integer value z with a length $n=g+m$ as follows:

$$[12] \quad f = \underbrace{z_{n-2}2^{g-2} + \dots + z_{m+1}2^1 + z_m2^0}_{g-1 \text{ Bit}} + \underbrace{z_{m-1}2^{-1} + z_12^{-m+1} + z_02^{-m}}_{m \text{ Bit}}$$

The bit z_{n-1} is reserved for the algebraic sign.

Run-Time Measurement using C166V2 Starter Kit

As mentioned, to quantify the run-time of the detailed model, the individual control algorithm primitives have been implemented within functional elements. In each of these elements, the algorithm is written in C-code compatible with the target processor. The variable types

of the processor compiler are declared within the primitive. The total-system simulation subsequently assures that the C-code algorithm performs as its original, and programming failures are avoided as much as possible. Parts of the C code should be written such that the target processor compiler can generate good Assembler code. To convert the algorithm primitives into functional elements, it is necessary to arrange the functions in a linear succession of C-instructions according to the described data flow. Therefore, as drawn in fig. 19, a scheduling of the primitives must occur. If the decomposition of the overall function into algorithm primitives is done correctly, this scheduling will not be difficult to realize. The block diagram of the functional elements can be seen in fig. 22.

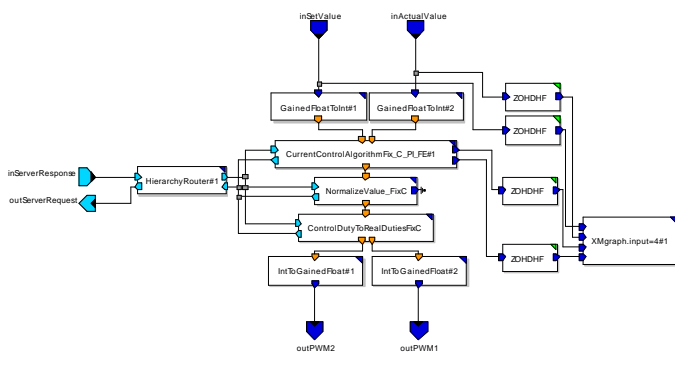


Figure 22: Functional Elements of Current Control Algorithm.

For execution on the detailed model, the methods within the functional elements must be converted with the microprocessor compiler to the machine language. In most cases it is also necessary to create a test environment for the function execution. In this case the “detailed model” exists in silicon, i.e., the XC164, so the execution time can simply be measured with an oscilloscope by probing the test board before and after a function call. The resulting measurements are summarized in Table 1.

Functional Element	PI-Controller	Normalized Value	ControlDuty toRealDuty
Run Time	1.827 μ s	1.533 μ s	1.077 μ s

Table 1: Run-Time Measurements of Current Control Algorithm on Pegasus.

SIMULATION RESULTS AND ANALYSIS

TIMING DIAGRAMS

Mechatronic Architecture

The plots below show the simulation results for the decentralized option illustrated in fig. 12. All performance

models are based on the IRM components and implemented with the DE-Domain. In the mechatronic architecture, data is transferred between the 32-bit VCU and 16-bit CCU over the MLI bus. Figure 23 depicts the system tasks for this architecture.

1. no activity
2. sampling on CCU (XC164)
3. transmission over MLI to VCU (TC1796)
4. position control algorithm processing on VCU
5. transmission over MLI to CCU
6. current control algorithm processing on CCU

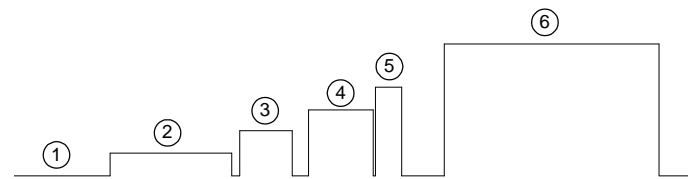


Figure 23: Loop Activity for Decentralized Architecture

The first plot in fig. 24 portrays the activity of the four CCU's, each controlling two of the eight valves. In the key, one color corresponds to one CCU. On the y-axis, -1 means the CCU is idle; 0 means the CCU is managing the current algorithm for valve i ; and 1 means the CCU is managing the current algorithm for valve $i+4$. The plot labeled “Loop Activity” explains how the data packet walks through the system resources for a complete control loop, as detailed in fig. 23.

Each “Set” item in the key represents one of the eight valves managed by the VCU. The third plot outlines the VCU algorithm processing activity. A y-axis value of -1 indicates that the microcontroller is not processing the position control algorithm but can be occupied with other tasks such as processing information from the ECU. The task values 0 to 7 indicate the VCU is processing the position control algorithm for valves 1 to 8, respectively. The bus traffic on the MLI can be seen in the fourth plot, where the colors indicate the send and receive activities, when the VCU master sends data to the CCU slave, or the VCU receives data from the CCU. A y-axis value of -1 indicates no activity, while values 0 to 3 indicate the master is communicating with CCU's 1 to 4, respectively. These plots confirm that the VCU and CCU microprocessors can manage the current and position control algorithms for all eight valves within the allotted loop cycle time.

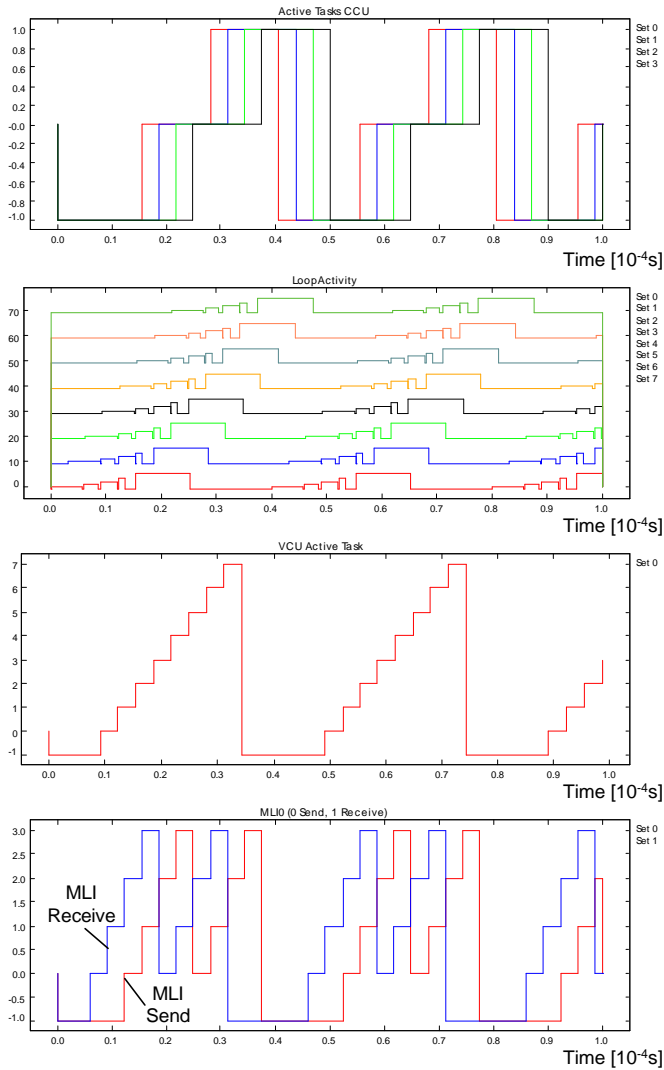


Figure 24: Simulation Results for Decentralized Architecture.

Centralized Architecture

The Loop Activity plot in fig. 25 demonstrates the data packet load on the system resources as it passes through the control loop, pre-defined as 40 μ sec. In contrast to fig. 23, the loop activity for each of the eight valves depicted has only three phases:

1. no activity
2. sampling
3. position and current control algorithms processing

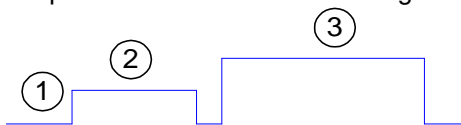


Figure 25: Loop Activity for Centralized Architecture

Timing results of the centralized architecture to control eight valves can be seen in fig. 26.

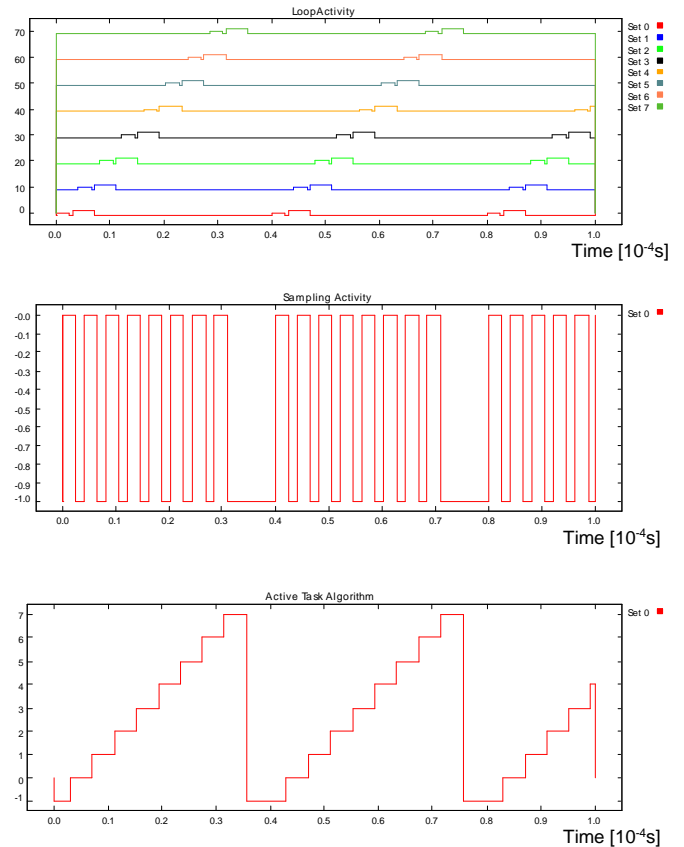


Figure 26: Simulation Results for Centralized Architecture.

The Sampling Activity plot displays the current sampling only because the position data is received as a digital incremental signal. The AD Converter is idle when the y-value is -1 . The third plot confirms that the VCU microprocessor can manage the current and position control algorithms for all eight valves within the allotted loop cycle time.

Valve Control

The plots in fig. 27 below represent the current and energy control achieved by the control algorithms. Since the mechatronic and centralized architectures can each manage the valve control within the 40 μ sec loop-time, these simulation results are the same for both architecture variations. The four current control curves are

- y0: desired current in upper coil [A]
- y1: actual current in upper coil [A]
- y2: control error (y0-y1)
- y3: current controller output [V]

where the current controller output is transformed into duty cycles for the Gate Driver PWM signals. The energy control curves are

- y0: set energy of spring mass system [J]
- y1: active energy of spring mass system [J]
- y2: controller error (y0-y1)

- y3: set current for upper current control [10A]

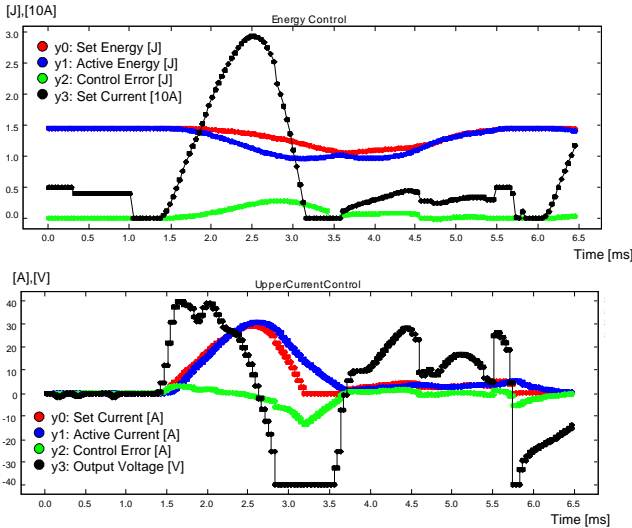


Figure 27: Current and Energy Control Diagrams

These control algorithms, together with the electromagnetic and mechanical models, achieve the landing speed shown in the following valve displacement profile.

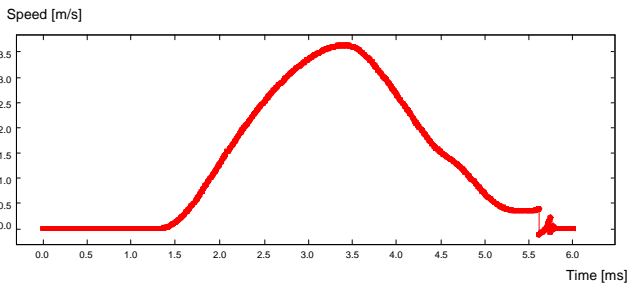


Figure 28: Landing Speed (m/s)

CONTROL INSTRUCTIONS

The diagrams in fig. 29 plot the maximum landing speed versus the number of data bits for the position, current, and PWM output. Based upon these results, the position data was set at 14 bits, current data at 9 bits, and PWM output at 4 bits. These plots are thus the basis for the choice of data packet size listed in fig. 12.

The position/energy control instructions are 32-bit, single-precision floating-point because this algorithm is managed on the 32-bit TriCore, which processes the valve control algorithm well within the given 40 μ sec control loop time. The current control instructions are optimized 16-bit fixed-point, whether running on a separate CCU or within the VCU microprocessor. These control instructions deliver the appropriate command output to the electromagnetic valve and achieve a controlled valve landing.

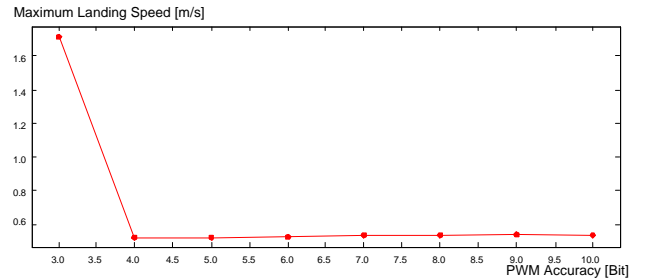
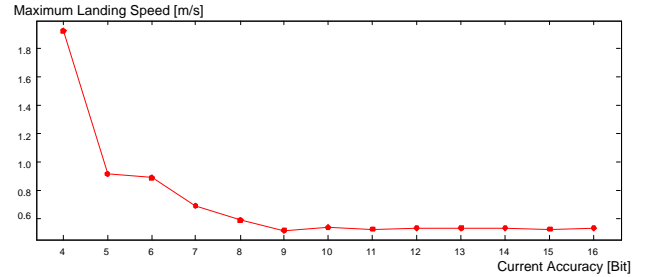
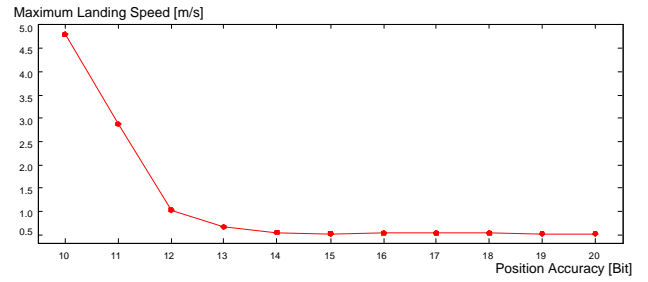


Figure 29: Landing Speed Accuracies

REQUIRED PROCESSING POWER

For the defined electromagnetic and mechanical models, the above diagrams suggest that the 32-bit microprocessor, modeled as the silicon definition presently exists, can manage the given control algorithms for 4 valves in a centralized configuration, or 8 valves if two PDL modules are added per GPTA for the position input data sampling. In a mechatronic partitioning, the 32-bit microprocessor manages the outer loop of the position/energy control algorithm while the 16-bit current controller autonomously handles the current control algorithm. Either system configuration provides the processing power needed to effectively follow and manage current dynamic and valve position.

CONCLUSION

For the given models, it is clear that when the 32-bit microprocessor with four additional PDL modules can alone manage both the position and current control algorithms to achieve a controlled valve landing for eight valves, this centralized solution requires less silicon than the mechatronic solution. Thus for the electromagnetic and mechanical models and control algorithms described in this paper, the centralized system partitioning is optimal. The partitioning can be easily re-examined with

the simulation tool for different actuator models and algorithms.

ACKNOWLEDGMENTS

The authors would like to thank Jens Barrenscheen of Infineon Technologies for his input regarding the current control algorithm. The authors would also like to thank Gunar Schorcht and Andreas Franck of MLDesign for their support of the simulation tool.

REFERENCES

1. Boie, Kemper, "Open and closed-loop control strategies for actuators of an electromechanical valve-train assembly." 3rd Symposium, Control Systems for the Powertrain of Motor Vehicles, 25-26 October 2001, Berlin.
2. MLDesigner Manual V2.2, <http://www.mldesigner.com>.
3. Straky, Isermann, Orthmann, Schöner, Wagner, "Model-aided design of robust actuator control for a fully variable electromechanical valve-train assembly." 3rd Symposium, Control Systems for the Powertrain of Motor Vehicles, 25-26 October 2001, Berlin.
4. C. Schernus, F. van der Staay, H. Janssen, J. Neumeister, B. Vogt, L. Donce, I. Estlimbaum, E. Nicole, C. Maerky, " Modeling of Exhaust Valve Opening in a Camless Engine", SAE 2002 World Congress, Detroit, Michigan, March 4-7, 2002.
5. Y. Wang, T. Megli, M. Haghgooe, K. S. Peterson, A. G. Stefanopoulou, " Modeling and Control of Electromechanical Valve Actuator", SAE 2002 World Congress, Detroit, Michigan, March 4-7, 2002.
6. G. Schorcht, Dissertation, "Entwurf integrierter Mobilkommunikationssysteme auf Missionsebene", 10. Juli 2000, TU-Ilmenau.
7. O. Föllinger, "Regelungstechnik Einführung in die Methoden und ihre Anwendung", 8. Auflage, 1994
8. W. Leonhard, "Digital Signalverarbeitung in der Meß- und Regelungstechnik", 1989.

9. B. Oestereich, "Objektorientierte Softwareentwicklung; Analyse und Design mit der Unified Modeling Language", 4. Auflage, 1998.

CONTACT

K. Manon McNair: Infineon Technologies AG, Automotive & Industrial, Balanstrasse 73, D-81541 Munich, Germany. Tel.: +49 89 23 42 93 94, FAX: +49 89 23 4955 4073 E-mail: manon.mcnair@infineon.com

Matthias Zens: Infineon Technologies AG, Automotive & Industrial, Balanstrasse 73, D-81541 Munich, Germany. Tel.: +49 89 23 48 40 13, FAX: +49 89 23 48 10 29 E-mail: matthias.zens@infineon.com

Horst Salzwedel: Ilmenau Technical University, Helmholtzring 1, D-98693 Ilmenau, Germany. Tel: +49 3677 69 13 16, FAX: +49 3677 69 12 85 E-mail: horst.salzwedel@theoinf.tu-ilmenau.de

DEFINITIONS, ACRONYMS, ABBREVIATIONS

CCU: Current Control Unit

ECU: Engine Control Unit

GPTA: General Purpose Timer Array

IRM: Instruction Resource Model

MLI: Micro Link Interface

PDL: Phase Discrimination Logic

UML: Unified Modeling Language

VCU: Valve Control Unit