International Conference on Control, Automation and Systems 2008
Oct. 14-17, 2008 in COEX, Seoul, Korea

Implementation of AUTOSAR I/O Driver Modules for a SSPS System

Gwangmin Park[1], Daehyun Kum[2], Sungho Jin[3] and Wooyoung Jung[4]

Division of Advanced Industrial Science & Technology, DGIST, Daegu, Korea
(Tel : +82-53-430-8465; E-mail: {ggangmin[1], kumdh[2], sungho[3], wyjung[4]}@dgist.ac.kr)

Abstract: The AUTOSAR Basic Software is composed of Service Layer, ECU Abstraction Layer (EAL), Microcontroller Abstraction Layer (MCAL) and Complex Driver. In these components, an IO Driver Module including I/O Hardware Abstraction (IoHwAb) is one of the most important components in a development of ECU-target applications. However, it is difficult to integrate and emulate it to ECU-targets in coincidence with the AUTOSAR MCAL/EAL architecture. In this paper, we present a simple implementation process of an AUTOSAR IO Driver module and system integration method. In addition, we apply this development process to the Speed Sensitive Power Steering (SSPS) system and evaluate its performance as a result of applied tests.

Keywords: AUTOSAR, I/O Driver module, MCAL, EAL, IoHwAb, SSPS system

1. INTRODUCTION

The AUTOSAR software architecture consists of a layer model shown in Fig. 1 that is basically divided into three different layers: the Application Layer, the Runtime Environment (RTE), and the Basic Software (BSW). To achieve AUTOSAR goals related to partitioning of the application software from base modules and functions, the vehicle electronics is abstracted and subdivided into several layers. Connections in an actual microcontroller and thereby its physical basis is represented by the Microcontroller Abstraction Layer, which maps the microcontroller's functions and periphery. Features that the microcontroller cannot offer can also be emulated in the software. Above this layer is the second layer, which is the ECU Abstraction Layer. It defines the ECU's internal hardware layout and provides drivers for the ECU's external periphery [1]. These two abstraction layers show the most Hardware-dependent. Thus, these are essential components to develop ECU-target applications. However, most E/E engineers take a long time to integrate and design ECUs including debugging processes related to IO driver modules, which include MCAL/EAL. In this paper, we introduce a simple ECU integration and implementation process of an AUTOSAR IO Driver module and test it by applying real application targets.

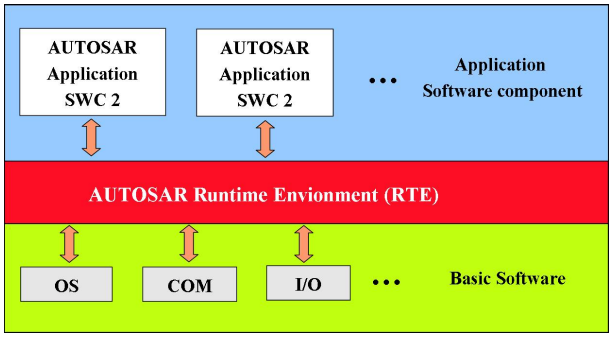

Fig. 1 AUTOSAR SW architecture.

2. MCAL/EAL

2.1 MCAL

The MCAL abstracts the microcontroller from above layers to make upper layers hardware-independent. According to the ECU software architecture, the microcontroller abstraction layer has been placed between the ECU abstraction layer and the real hardware. Access to the hardware is routed through the Microcontroller Abstraction Layer to avoid direct access to microcontroller registers from higher-level software.

MCAL is a hardware specific layer that ensures a standard interface to the components of the Basic Software. It manages the microcontroller peripherals and provides the components of the Basic Software with microcontroller independent values [2].

The Microcontroller Abstraction Layer consists of the following module group. It contains internal drivers, which are software modules with direct access to the microcontroller internal peripherals.

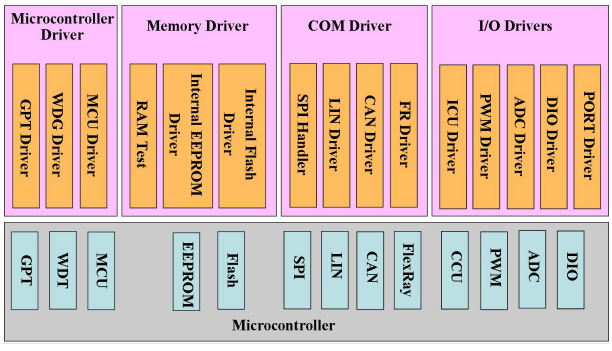

Fig. 2 Microcontroller Abstraction Layer.

The four module groups are implemented as follows:
1. I/O Drivers for analog and digital I/Os (e.g. ADC, PWM, DIO).
2. Communication Drivers for the ECU onboard (e.g. SPI, I2C) and vehicle communication (e.g. CAN).

3. Memory Drivers for on-chip memory devices (e.g. internal Flash, internal EEPROM) and memory mapped external memory devices (e.g. external Flash).
4. Microcontroller Drivers for internal peripherals (e.g. Watchdog, Clock Unit) and functions with direct microcontroller access (e.g. RAM test, Core test) [3].

From among these, the I/O Drivers module is the most important one for the application development to operate and test sensor/actuator components. The I/O driver module is for analog and digital input/output, it consists of internal peripheral devices such as ADC, PWM, and DIO.

2.2 EAL

The I/O Hardware Abstraction (IoHwAb) is a group of modules which abstracts from the location of peripheral I/O devices (on-chip or on-board) and the ECU hardware layout (e.g. microcontroller pin connections and signal level inversions). The I/O hardware abstraction does not abstract from the sensors/actuators. These components are main parts of hardware integration and ECU I/O porting due to their dependency to the hardware. The different I/O devices are accessed via an I/O signal interface.

The task of this group of modules is to represent I/O signals as they are connected to the ECU hardware (e.g. current, voltage, frequency), and to hide ECU hardware and layout properties from higher software layers [3].

In addition, abstraction of signal path of the ECU hardware (Layout, μC Pins, μC external devices like I/O ASIC) provides signal based interface, for example, static normalization/inversion of values according to their physical representation at the inputs/outputs of the ECU hardware (compensation of static influences caused with the path between ECU I/O and μC pin, e.g. voltage divider, hardware inversion) [4].

The I/O Hardware Abstraction is the most basic and essential Basic Software with I/O related drivers, such as port driver and I/O driver, to develop application systems for the ECU-target. For the microcontroller configuration and testing, it is need to configure the hardware drivers coincident with application system requirements and well-design the standardized interfaces to abstract the ECU layout from the above layer.

3. IO DRIVER MODULE IMPLEMENTATION

3.1 Integration

After completing the system design process and ECU configuration step including software components, task design, BSW configuration, and so on, ECU development engineers should integrate 3-Layer codes, which are Application Software code, RTE code, and BSW code.

In this Process, one of the most important things is to eliminate non-essential SW components, and then links one to other modules. In addition, based on the specification of each I/O driver module, the implementations of API and library function should be operated by using an optimized implementation method, such as task scheduling or driving sequence in I/O Hardware Abstraction layers.

3.2 IoHwAb Implementation

The I/O driver module implementation represents a primary role in an ECU hardware development process for implementing applications or functions related to vehicles. Applications have access to registers in the hardware through the Microcontroller Abstraction Layer.

To operate I/O hardware drivers, we should first initialize overall registers of the MCAL system module, and then initialize all drivers. Then, we can operate the drivers according to the purpose of each application by calling APIs or functions, which can configure register information.

The interface with the I/O Hardware Abstraction should be implemented at the Runnable of tasks in compliance with the sequence diagram in the specification. In the diagrams shown in Fig. 3, the diagrams show the sequences when it calls the API and service. They show normal operation and development modes with error conditions. For developing I/O drivers, a specific driving mode should be primarily selected, and an initialization function or a start function is executed. Subsequently, the operation of drivers starts by calling the main API.

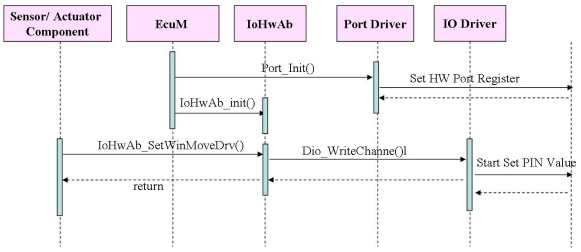

Fig. 3 Sequence diagram.

For actual implementations of I/O hardware driver modules, it is the most important issue how to implement interfaces related elements, such as calling function, timing control, scheduling, control method, and so on.

As shown in Fig. 4, the I/O Hardware Abstraction interfaces one side of the MCAL drivers via Standardized Interfaces and the other side of the RTE via AUTOSAR Interfaces. Hence, the IO Hardware Abstraction shall respect the virtual port concepts. Standardized Interfaces can be configured as an easy manner for implementing APIs, data types, and so forth when it communicates with many data in only one-ECU. However, AUTOSAR Interfaces have to comply with constraints to coincide with the specification since it should provide not only Intra-ECU communication but also Inter-ECU communication.

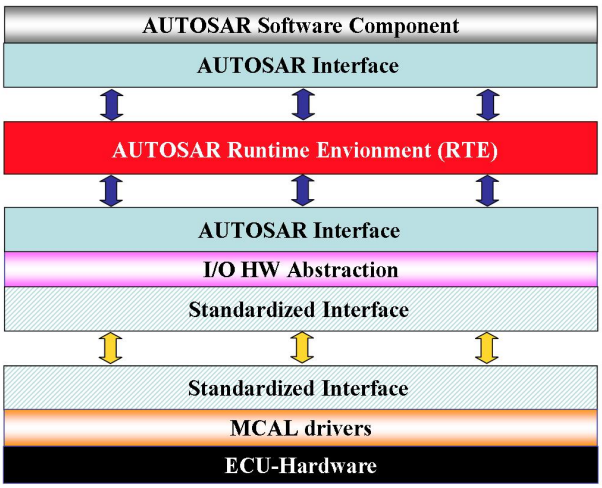

Fig. 4 Interface between SWC and I/O driver module.

There is no mandatory interface for the I/O Hardware Abstraction. The Standardized Interface depends on the implemented ECU signals and drivers as follows:
1. For an implementation of ECU discrete signals, it is necessary to have the DIO and PORT drivers.
2. For an implementation of ECU analog signals, it is necessary to have the ADC driver in the least.
3. For an implementation of ECU PWx signals, it is necessary to have the PWM and ICU drivers in the least [5].

The following source code shows the example of the interface implemented by using a DIO driver to operate digital output signals at the I/O Hardware Abstraction Layer.

```
…
Dio_WriteChannel(WIN_UP, STD_LOW);
Dio_WriteChannel(WIN_DOWN, STD_LOW);
…
```

If we want to implement the ADC driver, ICU driver, PWM driver, and so on, it can be performed using a provided API with the MCAL driver at the internal I/O Hardware Abstraction functions.

The I/O Hardware Abstraction provides functions and source codes that access to DIO channels for a specific signal. A software engineer should make a program of dummy-function to connect it using a specific driver. When a hardware specific design process related to the I/O Hardware Abstraction is finished, initialization functions, such as IoHwAb_Init() and IoHwAb_GetVersionInfo(), for receiving the feedback information of the ErrorHook and version description are generated automatically. The rest functions related to user-defined signals or data are generated as empty except for API names and return types. Thus, software integrators should make a program inside of the function directly. These functions are largely divided into some functions related to the OP_GET and OP_SET operations. The OP_GET operation performs input operations that transfer the data to the sensor software component above the RTE after receiving I/O driver inputs, such as ADC analog value, ICU input signal captures, and so on. Conversely, the OP_SET operation is responsible for sending commands from actuator software components to the BSW hardware driver layer.

Fig. 5 presents a series of processes that implement OP_GET and OP_SET operations in a layer structure. As this example, OP_GET and OP_SET operations depend on the access attribute considering an ECU Signal associated to a port. If this ECU signal is configured as an input, the OP_GET will be required. Reversely, if the ECU signal is configured as an output, it will be required to call the OP_SET operation. Basically, the functions related to OP_GET or OP_SET are generated with empty contents status in compliance with the AUTOSAR specification, and the function name is a form of IoHwAb_Set<signal name> (predefined output data type) or IoHwAb_Get<signal name> (predefined input data type).

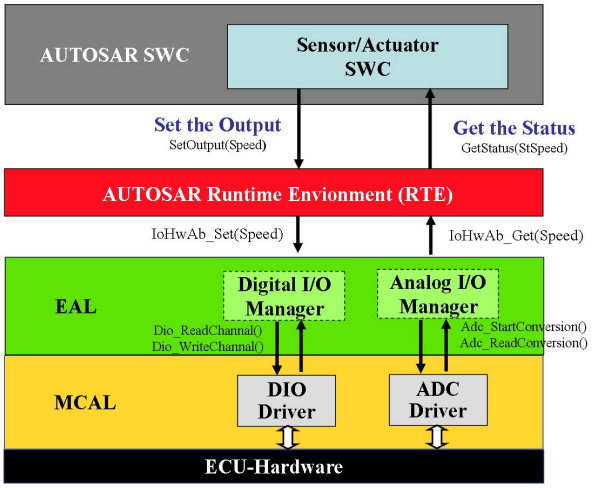

Fig. 5 Example of IoHwAb operation process.

The following example shows the implemented content of the source program for developing a real application system.

```
Std_ReturnType IoHwAb_SetCmdWrnSig_Warning
(   uint16   signal   )
{
     /* Signal OP_SET operation */
  if (signal == 0){
    Dio_WriteChannel(FAIL_SIGNAL, STD_LOW);
  }
else{
    Dio_WriteChannel(FAIL_SIGNAL, STD_HIGH);
  }
return E_OK;
}
```

At the application software layer, the implemented API as mentioned example codes is called middleware or RTE according to AUTOSAR interfaces. In actual, the implemented source code calling from the Runnable

of software components is as follows:

```
void ActRoomLamp_RE(void)
{
   /* Parameter Definition */
  t_brightness AcRmLamp;
  IoHwAb_PwxDutyCycleType RmLampPWM;
   /* Read From RTE */
  Rte_Read_ActRmLamp_cmd(&AcRmLamp);
   /* User Function */
  RmLampPWM=Act_RmLampPWM(AcRmLamp);
   /* Write to RTE (IoHwAb) */
  Rte_Call_RmLamp_OP_SET(RmLampPWM);
}
```

It is required to correctly use OP_GET and OP_SET functions to handle hardware-dependent I/O drivers in lower layers within the Runnable, which is assigned to Task units in the application software layer.

4. APPLICATION TO A SSPS SYSTEM

In this section, we confirm that a Speed Sensitive Power Steering (SSPS) system developed with the AUTOSAR simple I/O driving module is operated as a desired way. Moreover, we can reduce the development time compared with existing ECUs.

We developed a SSPS ECU with the migration of AUTOSAR platform according to the AUTOSAR I/O driver module implementation process as previously introduced. The SSPS system is composed of a steering wheel, sensor part, actuator part, and a single ECU controller. It is redeveloped using new AUTOSAR platform and new ECU hardware inclusive of new microcontroller core and MCAL module to migrate the conventional commercial products. Using generally used tool chains of Vector Inc., Mathworks Inc., Telelogic Inc., and Freescale Inc., we developed this application system according to the V-process as shown in Fig. 6. To reduce the time and effort spent in the development, we introduced several state-of-the-art technologies and methodologies, such as Model-Based-Design (MBD), Auto-Code-Generation (ACG), Hardware-In-the-Loop-Simulation (HILS), and Software-In-the-Loop-Simulation (SILS) [6].

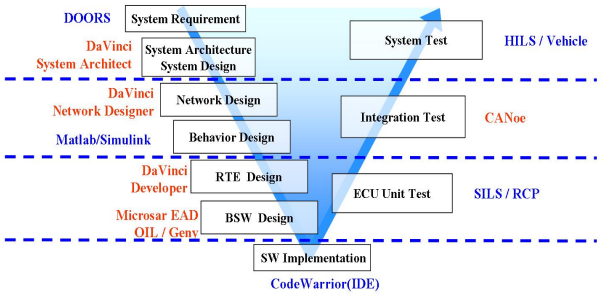

Fig. 6 Development process and tools.

4.1 System design

Requirements in a SSPS system are divided into four categories, such as steering wheel force on vehicle speed, steering wheel force on steering wheel rotational speed, reversion steering force, and requirements in warning lamp or self diagnostics. These are managed by DOORS from the Telelogic Inc.

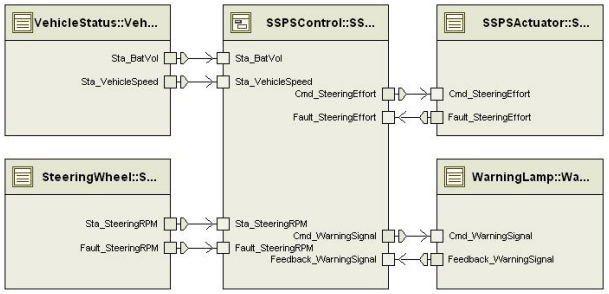

Fig. 7 SSPS system software architecture.

The SSPS system software is composed of four sensor/actuator components and one composite software component according to its control algorithm (see Fig. 7). Each component includes atomic software components by function, it is required to define a Runnable entity and design a specific content of atomic software component, such as trigger condition, periodic cycle, I/O interface, initialization, and other factors.

4.2 Function Modeling

As the system level design is finished, a function modeling process is needed to generate a model source code included in the software component. It can be modeled by building a program directly, or by using proper modeling tools, such as Matlab/Simulink or Statemate. Fig. 8 shows the example of the state diagram for a function modeling process using the Statemate tool from the Telelogic Inc.

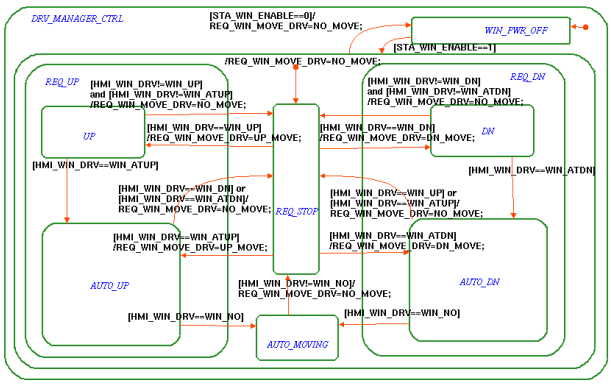

Fig. 8 Function modeling process using the Statemate.

4.3 Network design

In the case of the system configured by more than

one ECU, a network designer should select communication protocols and buses and define a network topology. In addition, several software components are assigned to specific ECU hardware. Since this SSPS system consists of a single ECU, the network design process is excluded in a real development process.

4.4 ECU design

After completing the system level design, an internal software design process for each ECU is required. First, we designed and defined the Runnable in the software component at an ECU level. Then it was assigned to specific tasks. When the software component design and the design related to RTE are finished, we have to configure the basic software components, such as I/O driver modules or memory and generate source codes. Fig. 9 shows the steps in a basic software configuration process using the EAD from the Microsar Inc.

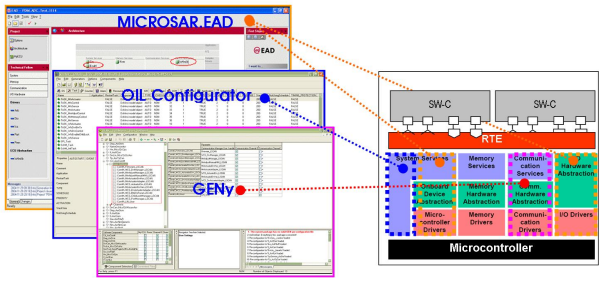

Fig. 9 Basic software configuration.

4.5 System Integration

Currently, the AUTOSAR tool chain does not support the functions of software integration and build. Thus, each source code in the application software, RTE, and BSW should be combined with the Integrated Development Environment (IDE), such as CodeWarrior, and then it should be linked to the real-target ECU through code compile process as illustrated in Fig. 10.

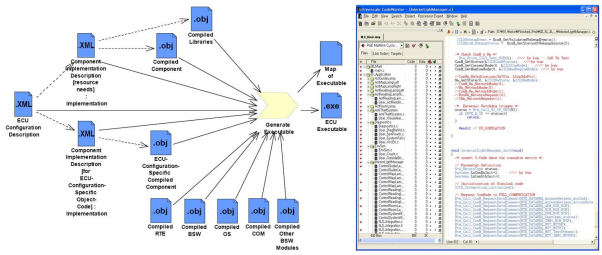

Fig. 10 System Integration using the Codewarrior.

In this development step, it is important to implement the interfaces between the RTE and hardware I/O drivers via the I/O hardware abstraction layer. For example, the source code shows the used API for the interface with the RTE in the SSPS system.

```
…
Rte_Read_EffortVSpeed_Effort(&fEffortVSpeed);
fEffortFinal=User_FinalEffort(WrnSig,
fEffortRecovery, fEffortStrRPM, fEffortVSpeed);
Rte_Write_Cmd_StrEffort_Effort(fEffortFinal);
…
```

The RTE layer uses an available API, such as Rte_Read_<signal name>, to interface with the lower layer, since the RTE cannot access variables owned by the IO hardware abstraction layer module.

4.6 Test and Validation

The system level test is an essential procedure due to the large number of interconnections between systems and components, and its reliability verification. We performed a PC-target function test for this project, which is finished software integration, before applying an ECU unit test. Using CANoe from the Vector, Inc., we made a virtual network node and I/O simulation panel for the verification of ECU input/output values. Subsequently, we performed a series of test based on the requirement test cases.

After applying SILS and HILS tests, as the last step of the project development and test, we applied this unit ECU to a real vehicle and proceeded to test in actual roads after completing SILS and HILS tests. We mounted the developed SSPS ECU to the real vehicle (Lacetti, GM Deawoo), and verified the steering sensibility results of a test engineer rather than numerical comparison with existing ECU products. When we tested on the road, basic functions of the steering force, which is changed with vehicle speed, was satisfied the requirements, and the test result of steering sensibility was also similar to currently used commercial products.

As the AUTOSAR platform is introduced, an OS kernel is newly added, and then communication parts, diagnostic part, etc. are included in the platform. Therefore the total code size largely increased, and then memory capacity is also required as more than that of the existing scale. In particular, the platform code largely occupies the memory volume where the application software code size is just about 2 Kbytes.

Table.1 Non-AUTOSAR vs. AUTOSAR platform.

		Non AUTOSAR Type	AUTOSAR Type
Microcontroller		MC68HC908EY16	MC9S12XDP512
Code Size	ROM	4.3 KBytes	43.1 KBytes
	RAM	0.3 KBytes	8.8 KBytes

However, if high quality platform is developed and code size is optimized, then it can be expected that the code size of the AUTOSAR platform system decreases. In addition, when we consider the memory development speed of semiconductor companies, the memory size will not make a problem while the existing system is migrated to the AUTOSAR platform.

5. CONCLUSION

For years now, the complexity of vehicle electronics continues to grow. Then, the shorter development time and software portability are required for the standards of software architectures. In order to cope with the increased needs, the AUTOSAR project is proposed one of the most comprehensive and promising solutions.

As we presented in this paper, the AUTOSAR provides means for an easy integration of software components based on interface methods between RTE and I/O drivers. In addition, the I/O driver module design and integration techniques can be used for detailed AUTOSAR specifications.

By applying these concepts to a SSPS system, an ECU-target integrator can configure the AUTOSAR hardware module more formally and simply in compliance with standard specifications. Additionally, thanks to the integration on C-code source level, the AUTOSAR integration platform becomes independent of hardware.

ACKKNOWLEDGMENTS

This work was supported by the Basic Research Program of Ministry of Education, Science and Technology, Korea. We express our heartfelt thanks to Eunkyung Gu and Seongho Lee at KDAC Corporation for cooperation and contributing to this paper.

REFERENCES

[1] M. Wernicke and J. Rein, “Integration of Existing ECU Software in the Autosar Architecture,” *ATZelektronik*, pp6-9, 01/2007.

[2] AUTOSAR Technical overview, 2007, AUTOSAR Specification Release 3.0, Retrieved on 28/11/2007.

[3] AUTOSAR Layered Software Architecture, 2007, AUTOSAR Specification Release 3.0, Retrieved on 15/11/2007.

[4] AUTOSAR Basic Software Module, 2007, AUTOSAR Specification Release 3.0, Retrieved on 05/12/2007.

[5] AUTOSAR I/O hardware Abstraction, 2007, AUTOSAR Specification Release 3.0, Retrieved on 13/12/2007.

[6] K.Nichikawa and K.Kajio “TOYOTA Electronic Architecture and AUTOSAR Pilot,” *2007 SAE international*, 2007-01-1614, 2007.

[7] H.Heinecke and J.Bielefeld, “AUTOSAR-Current results and preparations for exploitation,” *7th EUROFORUM conference* ‘Software in the vehicle’ Stuttgart, Germany, pp3-4, May 2006.