Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification

Mattias O'Nils
Royal Institute of Technology
Department of Electronics, ESD Lab.
Electrum 229, S-164 40 Kista, Sweden
mattias@ele.kth.se

Axel Jantsch
Royal Institute of Technology
Department of Electronics, ESD Lab.
Electrum 229, S-164 40 Kista, Sweden
axel@ele.kth.se

Abstract

We present a method for generation of the software part of a HW/SW interface (i.e. the device drivers), which separates the behaviour of the interface from the architecture dependent parts. We do this by modelling the behaviour in ProGram (a grammar based protocol specification language) and capture the processor and OS kernel parts in separate libraries. By separating the behaviour from the architectural specific parts, compared to other approaches up to 50% development time can be saved the first time the component is used, and up to 98% for each time the interfaced component is reused.

1. Introduction

As elaborated by Tuggle in [10], for a given device, a device driver must be rewritten for every operating system it will be used with. For hierarchical device drivers like SCSI systems, the number of device drivers needed for a device is the product of the number of supported operating systems and the number of supported host adapters. If we also consider different implementation styles of a device driver, the number of possible implementations grows even more sharply. One quickly realizes, that development, verification, and maintenance of all these device drivers is a huge task.

Existing approaches address this problem only partially. CoWare [11] and Polis [1] concentrate on the case where the whole design functionality is captured within their environment and then during the system synthesis the communication is refined, i.e. the device drivers are generated together with the custom HW and operating system. But if the user wants to use IP blocks and an off-the-shelf RTOS, he/she will face the same troubles as for manual design [10]. In Chinook [2], the definition of a device driver also includes the bus interface, which makes it very architecture dependent. MakeApp [6] is a tool for generating device drivers for different devices and processors matching user defined configurations. Both Chinook and MakeApp solve only part of the problems described in [10], since they generate code only for a specific real-time kernel (Chinook) or no kernel (MakeApp).

To solve this problem we have proposed techniques for modelling of device drivers independent of both the real-time kernel and the processor [9]. In this paper we extend that with a synthesis technique to generate device drivers from three inputs: (1) an architecture independent protocol described in ProGram [12]; (2) a characterization of the operating system; (3) a characterization of the processor and its bus interface. With architecture we mean both the processor and the operating system.

The next section presents the proposed design flow for device drivers. Then we discuss the architecture dependent parts of a device driver. The fourth section presents a technique for modelling software architectures for device driver generation. In the fifth section we present the synthesis method. Finally, we conclude the paper with two case studies.

2. Design flow

As described in [9], we model device drivers independently of the architecture using ProGram [12]. ProGram, is a grammar based notation for protocol applications. Specifications in ProGram deal with sequences of allowed events as opposed to states and state transitions in FSM model.

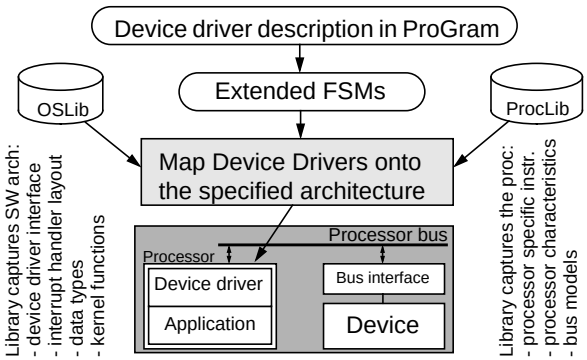

Figure 1.Device driver design flow.

The ProGram description is synthesised into an untimed extended state machine (described in [9,12]), which in turn is the input to the architecture mapping procedure described in section 5. The mapping procedure uses data from two libraries to generate the architecture specific code. The first of these libraries is the library that captures the information on the OS architecture, the second captures the processor specific characteristics, see figure 1.

3. Architecture dependent parts

In the following sections we discuss architecture specific parts of device drivers, i.e. synchronization, execution delays, interrupt handling, mutual exclusion, and the device driver interface.

3.1. Synchronization with external events

The waiting for an external event, i.e. the synchronization of the device driver with an external event, can be implemented in three ways, as illustrated in figure 2: (a) polling of device signals, (b) wait for a fixed time (when the device is known to generate an event), or (c) wait for an operating system event generated by an interrupt that is triggered by the device. An example of where synchronization is needed is an analog to digital converter. There we first initiate a conversion by writing some control data, and then wait for the conversion to take place, i.e. synchronize with the conversion ready event. This is done by polling a status register, wait for the specified conversion time, or let the device generate an interrupt signal. After that, read the sampled value from the device. Polling is the only one of these three synchronization schemes that can be implemented independent of the software and OS architecture.

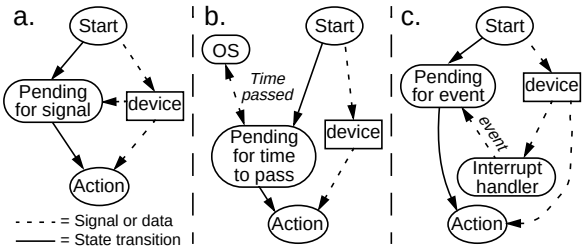

Figure 2.Synchronizing with external device.

3.2. Delay execution

As described in the previous section, in some situations a device driver has to halt itself for a certain amount of time. This could be necessary when waiting for an analog-digital conversion, but also to wait for coprocessors during the initialization process. The wait function is provided by some kernels, hence this function is architecture dependent.

3.3. Interrupt handling

Interrupt service routines can be used to synchronize a device driver with a hardware event as described above. Sometimes they are also independent routines with their own behaviour, e.g. when an interrupt handler receives data from a communication device and writes it to a buffer. There are two possible ways to implement this: (a) the interrupt service routine informs the task about the event when the interrupt is activated and the task handles the action. (b) The interrupt routine also performs the action itself, see figure 3.

The implementation in figure 3b is more suited for actions with short execution time since alternative 3a introduces a communication overhead. Alternative 3b has also a

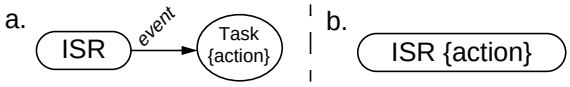

Figure 3.Implementation of an interrupt routine.

shorter and deterministic response time but introduce indeterminism in the kernel behaviour. 3a's response time is dependent on the priority of the task responsible for the action.

3.4. Mutual exclusion

There are two possible ways to prevent several application threads accessing the same device simultaneously: (1) with help of semaphores and (2) to disable the interrupt mechanism. Both have there advantages and disadvantages. Disabling interrupts is fast and do not add overhead to the execution but it disables the normal behaviour of the kernel, i.e. introduce indeterminism. Semaphores do not affect the kernel behaviour but introduce communication overhead. So the conclusion is that one should use semaphores for complex device drivers and disable interrupt for simple ones.

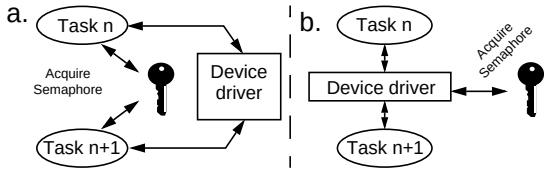

Figure 4.Handling mutual exclusion.

The control for mutual exclusion can be placed within or outside the device driver, see figure 4 for an example with a semaphore for handling mutual exclusion. The benefit of having it outside (i.e. within the application code) the device driver is that the device driver will not be dependent on the architecture for handling of mutual exclusion, the draw back is that the code will be less structured.

3.5. Device driver interface

For device drivers developed for single threaded software with no operating system, the interface to the application software (i.e. how in and out parameters to the device driver are handled, driver component naming style) is determined by designer, project or company coding style. There are systems with real-time kernels like the µC/OS or RT-kernel where the interface to the application is also defined by the designer. For other systems with kernels/ operating systems like UNIX, OS9, VxWorks and OS/2 the interface to the application/operating system is defined by the operating system [10].

4. Modelling of the architecture

To be able to map architecture dependent parts of a device driver to a specific architecture, the characteristics and behaviour of the architecture have to be captured. As seen in figure 1, the architecture is captured in two parts: the kernel specific parts (further described in section 4.1)

and the processor specific parts (further described in section 4.2). Throughout this section, we will use the µC/OS [7] and MC68000 [8] as examples.

4.1. Software environment

The model of the software environment (i.e. the OS characteristics and functionality) is divided into three parts: (1) the environment characteristics, (2) the interface to the application/OS, and (3) the macro for accessing system services.

4.1.1 SW environment characteristics

This part contains the environment name, type, timing, and include files needed for this architecture. The type is captured since it affects the behaviour of the interrupt routines. The characteristics for the model is defined in the table below.

Characteristics	**Value(s)**
name	Name of the architecture (string)
kernel_type	{preemptive, not-preemptive, single thread}
tick_time_period	Time between two clock ticks
interrupt_supported	{true, false}
header_files	{empty, list of header files}

Example 1 *A system with µC/OS kernel and a timing interrupt period of 10 ms, would have this characteristics model:*

```
name = uCOS;
kernel_type = preemptive;
tick_time_period = 10 ms;
interrupt_supported = true;
header_files = “uCOS.h”;
```

4.1.2 SW environment interface

With SW environment interface we mean the naming style of the different device driver functions and how data is transferred to and from these functions. The interface model is composed of three interface definitions for generation of function declarations: (1) interrupt service routine, (2) device driver function, and (3) task function. The definitions describe how the function names and parameter declarations should be generated. There is an additional translation table for type conversion, i.e. that translates the bit vector type to a C data type with respect to the bit width.

Definition/Table	**Outcome**
interrupt_routine_dec	interrupt routine declaration
driver_routine_dec	driver routine declaration
task_dec	task declaration
type_conversion_table	data typed signals

4.1.3 SW environment services

SW environment services are equal to system call functions offered by the kernel. Services that have to be modelled are discussed in section 3 and the system functions to provide these services are enumerated in the table below.

System call	**Input**
enable_interrupts	non
disable_interrupts	non
create_binary_semaphore	name (string)
delete_binary_semaphore	name (string)
pending_semaphore	non
set_semaphore	non
pending_time	no of ticks (integer)
enter_interrupt	non (*For preemptive kernels*)
exit_interrupt	non (*For preemptive kernels*)

Example 2 *A model of some of the uC/OS service functions:*

```
pending_time(n) = OSTimeDly(n);
disable_interrupts = OS_ENTER_CRITICAL();
enable_interrupts = OS_EXIT_CRITICAL();
```

4.2. Processor features

The processor features are captured by two parts: the processor characteristics and processor specific routines.

4.2.1 Processor characteristics

A processor can be characterized in many ways, but what is interesting for this approach is the internal and external data bus width since these will affect the address calculation of the device address. Also, the device access type has to be captured, i.e. if it uses a port mapped or memory mapped device access method. The table below defines the characteristics that need to be captured.

Characteristic	**Value**
name	Name of the architecture (string)
device_access_type	{memory mapped, port mapped}
internal_data_bus_width	{8,16,32}
external_data_bus_width	{8,16,32}

Example 3 *Model of characteristics for the MC68000:*

```
name = MC68000;
device_access_type = memory_mapped;
internal_data_bus_width = 32;
external_data_bus_width = 16;
```

4.2.2 Processor specific routines

For device driver generation some processor specific code must be provided. Specifically, the saving and restoring of registers must be captured. If enabling and disabling of interrupts is not supported by the kernel, then the processor specific code has to be used. The table below defines the processor specific code needed to be captured.

Assembler macro	**Parameter**
push_proc_register	non
pop_proc_register	non
enable_interrupts	non
disable_interrupts	non
return_from_interrupt	non
function_call	function label
read_port_mapped	data (*output*)
write_port_mapped	data (*input*)

Example 4 *Push the register onto the stack(MC68000):*

```
push_proc_register="move.l D0-D7/A0-A7,-(ssp)"
```

5. Mapping of device driver models

Several untimed FSMs are synthesized from the device driver description in ProGram. These FSMs are synthesized using the same technique as for synthesis of HW interfaces [12]. From there the implementation is created in two steps: (1) the transformation rules (section 5.1) perform optimizations and transformations that are independent of SW architecture and processor; (2) the code generation (section 5.2) uses the libraries *OSLib* and *ProcLib* to generate the device driver in C.

5.1. Transformation rules

Each transformation rule described in this section is applied to all states in the synthesised state machines. These rules map the FSM behaviour onto a general software architecture.

5.1.1 External synchronization

External synchronization points in ProGram are translated to wait statements in the generated FSM, with a transition upon the receipt of the synchronization signal, i.e. the t_j transition in the figure 5 (a) is taken. The condition for t_i is complementary to the one for t_j, i.e. t_i fires when t_j does not and vice versa. Synchronization with external signals are implemented by means of an interrupt routine signalling an event to the waiting device driver.

Rule 1: State S_i has two transitions (t_i and t_j), t_i is a transition to S_i and t_j to S_j (figure 5, a). If the conditions for t_i and t_j are complementary and composed of an external signal, then remove t_i, set the condition for t_j to true and insert a wait-for-event from the library into the code of S_i. Generate an interrupt routine, that sends this event upon activation (figure 5, b). The condition for t_j will be the interrupt signal that triggers the interrupt routine.

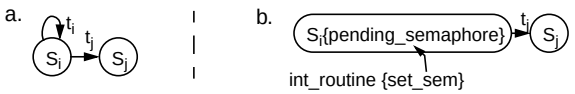

Figure 5.State transformation.

5.1.2 Identify internal synchronization

Internal synchronization points (i.e. synchronization between different parts of the device driver) are translated as the polling of an internal signal. As stated in [9] this construct can not be allowed between the different access functions of the device driver, since it could result in a deadlock. The only situation, where it will work and is necessarily, is when the signal is driven by an explicitly declared interrupt routine.

Rule 2: State S_i has two transitions (t_i and t_j), t_i is the transition to S_i and t_j to S_j (figure 5a). If the conditions for t_i and t_j are complementary and composed of an internal signal (5a), then remove t_i and set the condition for t_j to true and replace the assignments of the signal by a set and clear event from the library.

5.1.3 Partitioning of interrupt routines

As mentioned in section 3.3, an interrupt routine can either be implemented as an ISR with behaviour (figure 3b) or as an ISR sending an event to a task (figure 3a). Here we let the designer to interactively select this, in the future this could be determined by some heuristic.

Rule 3: If state S_i is an entry state to an interrupt routine and the user marked the routine for partitioning then replace the state "entry" by a state "task entry" and insert a wait-for-event from the library into the code of S_i and generate an interrupt routine that sends this event upon activation.

5.2. Code generation

The first action in the code generation is to translate bit vector types to C data types, by using the translation table from OSLib. The code generation for the different parts of a device driver follows certain assumptions about the structure of each component.

5.2.1 Device driver function

First the head of the function declaration is generated, i.e. function name and parameters, by invoking the driver function definition. The body of the function consists of routines for mutual exclusion from the OSLib and the C code for the FSM. The FSM code contains OS and processor specific code, based on entries in OSLib and ProcLib, respectively.

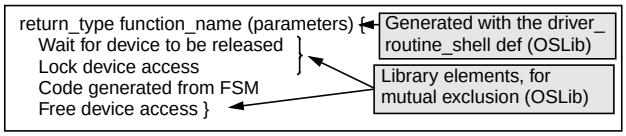

Figure 6.Code structure for an access function

5.2.2 Interrupt routine

Entry and exit of an interrupt routine must be in assembler language, since there is no support for interrupt in the C language. In the code generation of interrupt routines, the routine starts with storing and ends with restoring the processor registers. For a preemptive kernel the interrupt should also notify the kernel when it enters and leaves the interrupt routine, see figure 7.

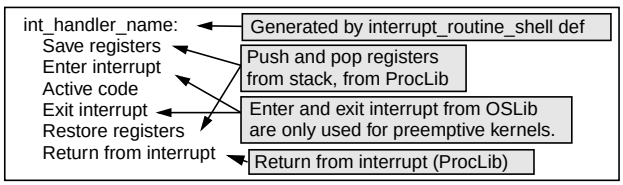

Figure 7.Code structure for an interrupt routine.

The code part of the interrupt routine is either code generated from an FSM that corresponds to the interrupt routine or just a library function sending an event. Either way, it will be a call to a C function.

5.2.3 Task

A task is generated only when an ISR is partitioned. In this context a task is only another implementation of an interrupt routine. Thus, its structure is very simple and basically an infinite loop containing a wait-for-event statement and the code for the corresponding FSM.

5.2.4 Miscellaneous

The enabling and disabling of interrupts can be captured in both libraries, OSLib and ProcLib, as described in section 4. If it exists in OSLib, the code generation uses it, otherwise the ProcLib is consulted.

Wait statements in the FSMs are replaced with the library macro from OSLib. But the real-time has to be translated into clock ticks, i.e. clock_ticks = time / clock_tick_period.

At the end of the code generation, a setup function is generated. This function creates and initializes the semaphores used by the other components. It also calculates the absolute addresses for the different device registers by using the base address and the declared relative addresses.

Example 5 *A device driver for the MAX197. We consider only the read value function, which is modelled in ProGram in figure 8a. This is synthesized into the state machine in figure 8b. After the transformation rules are applied we get the FSM in figure 8c and according to the optimization methods described in [9] we remove the redundant delay states and get the FSM in 8d. The coarse structure of the generated C code for the read function mapped onto µC/OS and MC68000 is illustrated in figure 8e.*

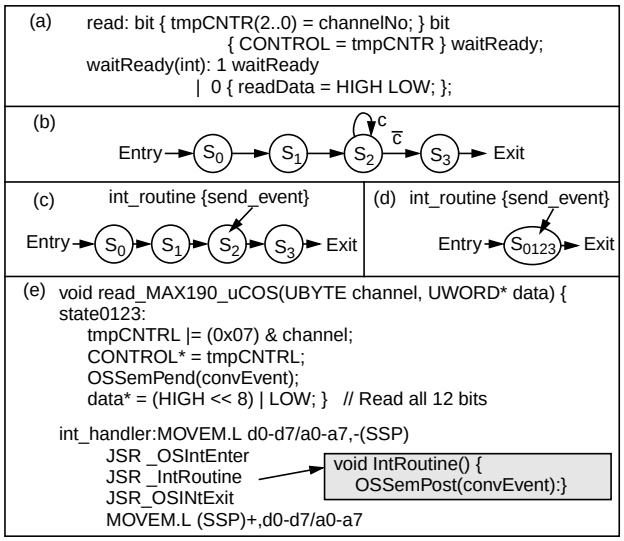

Figure 8.The read function of MAX197 from ProGram specification to C implementation.

6. Case study

We use two designs for the case study, a channel decoder of a transceiver in a D-AMPS base station and an operation and maintenance block (OAM) of an ATM network.

6.1. D-AMPS

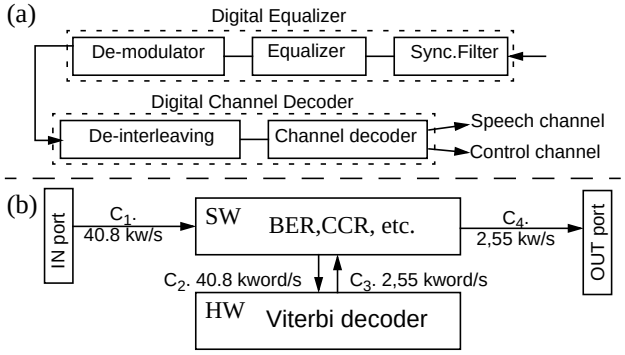

Figure 9.Receiver in IS-54B base station (a), and selected HW/SW partitioning (b).

D-AMPS [4] is a time division multiple access (TDMA) cellular standard. The standard has 832 channels per carrier in the 800 MHz range and a total of 1164 channels. The channel decoder block receives 272 bit frames that should be processed in 6.67ms. The process consists of: de-interleaving, Viterbi decoding, CRC-sum calculation, bit error rate estimation, sorting of speech data, and masking of bad speech frames.

We decided to implement the Viterbi decoder in hardware and the rest of the functionality in software. Figure 9b illustrates the hardware/software communication in the channel decoder design.

6.2. ATM Operation and Maintenance Block

The Operation And Maintenance (OAM) functionality

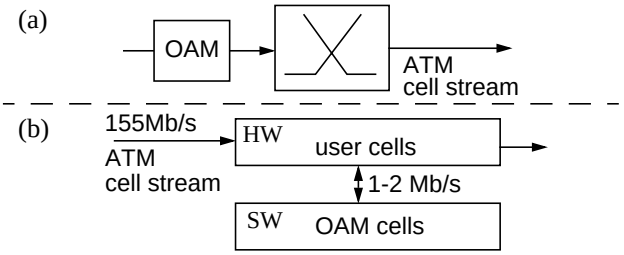

Figure 10.OAM block in an ATM network (a) and selected HW/SW partitioning (b).

[3] in an ATM network is situated near a switch. Specific ATM cells, OAM cells, are used to identify network problems, measure network performance, and communicate the state of the network between network nodes. OAM functions analyse, transform, and generate OAM cells. Figure 10b shows the selected HW/SW distribution, where ATM user cells are received and re-transmitted in custom hardware, while OAM cells are analysed and transformed in software. OAM cells typically constitute about 1% of the total traffic.

6.3. Results

We compare our approach to code development in C, since this is what a designer has to do if she/he wants to us an IP component and an RTOS in Coware [11], Polis [1] and MakeApp [6]. The comparison shows similar figures

Approach	C_1	$C_2 + C_3$	C_4	OAM HW/SW channel
C [LoC]	56	105	62	81
ProGram [LoC]	27	58	25	43
Difference [%]	48%	55%	40%	53%
Performance[a]	1.01	1.05	1.02	1.05

Table 1. D-AMPS channel decoder and ATM-OAM modelled in C and ProGram.

a. t_{exec}(C generated from ProGram) / t_{exec}(C)

for performance and code size, but a significant advantage of our approach in terms of designer productivity.

Table 1 shows the number of lines of C and ProGram code to describe the different protocols for the two designs. This shows that the amount of code in ProGram is typically 50% of the C code, because ProGram code does not contain any architecture dependent parts. The table also shows that the performance figures are comparable.

However, the true benefit comes from the reusability of a protocol specification in ProGram. If the same protocol should be implemented and maintained in several product versions with different RTOS and processors, the ProGram model can be fully reused, while the C code in any of the other approaches cannot. Figure 11 a and b give the amount of code to be written to implement a single protocol on a number of processor-RTOS combinations. On the X axes the processors and kernels are changed for even and odd generations, respectively. Figures 11 c and d translate these numbers into design time assuming a productivity of 16 LoC (Lines of Code) per day for ordinary systems and 4 LoC per day for embedded application code, left and right y-axis respectively, using the COCOMO model [13] for ordinary software systems and for embedded systems.

7. Conclusion

By separating architecture dependent from architecture independent parts of a device driver, we can significantly increase the reuse potential of models describing protocols, OS interfaces and processor interfaces. This addresses a critical step in IP based design, the specification and implementation of interfaces in the context third party HW blocks and SW components, e.g. RTOS.

We have presented a device driver synthesis technique which generates device driver implementations from a protocol, an OS and a processor description. The method is based on several interactive design decisions, e.g. the designer can select the implementation of interrupt routines and mutual exclusion handling. As shown in [9] and table 1 the efficiency of the generated code is close to hand written C code even though the code is not optimized.

Our approach will result in 50% less design time a protocol is used and 98% less for every product variant with a new processor or operating system, compared to CoWare, Polis and MakeApp.

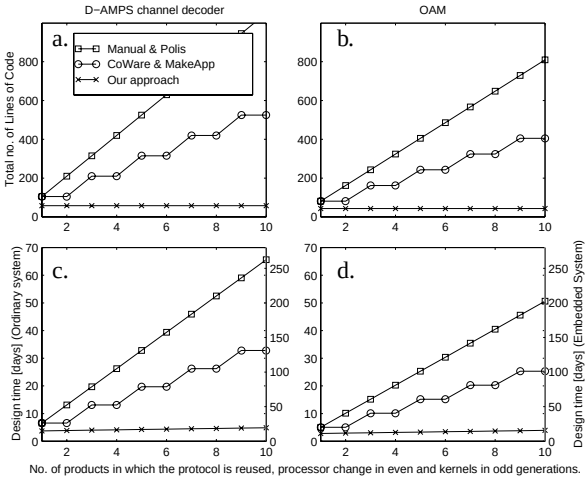

Figure 11.Lines of code and design time for channel decoder and OAM for reuse of protocol.

8. Reference

[1] F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, June 1997.

[2] P. Chou, R. Ortega, G. Borriello, “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems”, In *Proc. of the IEEE/ACM ICCAD*, Nov. 1992, pp. 488-495.

[3] M. De Prycker, *Asynchronous Transfer Mode*, Prentice Hall, 1995.

[4] *EIA/TIA Interim Standard, IS-54-B*, April, 1992.

[5] S. Gal-Oz, A. Cohen, “The Hazards of Device Driver Design”, *Embedded Systems Programming*, May 1997, pp. 34-46.

[6] R. Grehan, “Driver assistance”, *Computer design*, Nov. 01 1997, vol. 36, no. 1, pp. 75,76,78,80.

[7] J.J. Labrosse, *µC/OS, The Real-Time Kernel*, R&D Publications, ISBN 0-87930-444-8.

[8] *MC68000 - Reference Manual*, Motoral Inc.

[9] M. O'Nils, J. Öberg, A. Jantsch, "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces", In *Proc. of Euromicro Workshop*, Aug. 1998.

[10] E. Tuggle, “Writing Device Drivers”, *Embedded Systems Programming*, Jan. 1993, pp. 42-65.

[11] S. Vercauuteren, B. Lin, “Hardware/Software Communication and System Integration for Embedded Architectures”, in *Design Automation for Embedded Systems*, Vol 2, No. 3/4, May 1997.

[12] J. Öberg, A. Kumar, A. Hemani, “Grammar-based Hardware Synthesis of Data Communication Protocols”, In *Proc. of the 9th ISSS*, Nov. 1996, pp. 14-19.

[13] Boehm B. et al., “Cost models for future software life cycle processes: COCOMO 2.0”, *Annals-of-Software-Engineering*, vol. 1, 1995, pp.57-94.