Device Driver and DMA Controller Synthesis from HW/SW Communication Protocol Specifications

Abstract. We have separated the information required for HW/SW interface synthesis into three parts, the protocol specification, the operating system related information, and the processor related information. From these inputs a synthesis tool generates (a) device driver functions or (b) a combination of device driver functions and a DMA controller, depending on a designer's decision. The clean separation of information facilitates (1) efficient design space exploration with combinations of different processors, operating systems and protocols, and (2) maintaining a large number of different versions and variants of HW/SW interfaces by synthesising them on demand.

Protocols are specified as a grammar, which is fully independent of architecture and implementation. From this the synthesis tool generates device driver code in C and/or synthesizable RTL code in VHDL for DMA controllers. After the initial selection of implementation alternatives the presented methods are fully automated. Its computational complexity is quadratic in terms of the number of states. With real-life examples we show that the quality of the generated code is close to hand written quality in terms of performance, area and code size.

1. Introduction

Intellectual Property (IP) based design is emerging to close the gap between steadily increasing capacity, in terms of transistors on integrated devices, and the design productivity in terms of designed transistors per time unit. However, the integration of several IP blocks into a system on one chip makes specification and implementation of interfaces (e.g. bus interfaces and device drivers) a dominant design problem with respect to design time for embedded systems. For the extreme case where the entire system is composed of IP components, interfacing is the only design task. Consequently, we need effective ways to model, refine, and implement communication within an embedded system. To take full advantage of the reuse potential of IP blocks, be it software or hardware, we must efficiently support a large number of operating systems, processors, and communication protocols.

A hardware/software interface can consist of up to four parts: (1) device driver functions, (2) direct memory access (DMA) controller, (3) bus interface, and (4) the register file and handshake behaviour of the device. A device driver is the wrapper for a hardware device accessed from software. The device driver's behaviour essentially is a protocol defining how the device is accessed and synchronized with software. A device driver can perform all accesses to the device directly or it can use a DMA to transfer data between two places in the address space. In the latter case the device driver controls the DMA controller who in turn accesses memory and other devices. A DMA controller can move data between memory and memory, or memory and device.

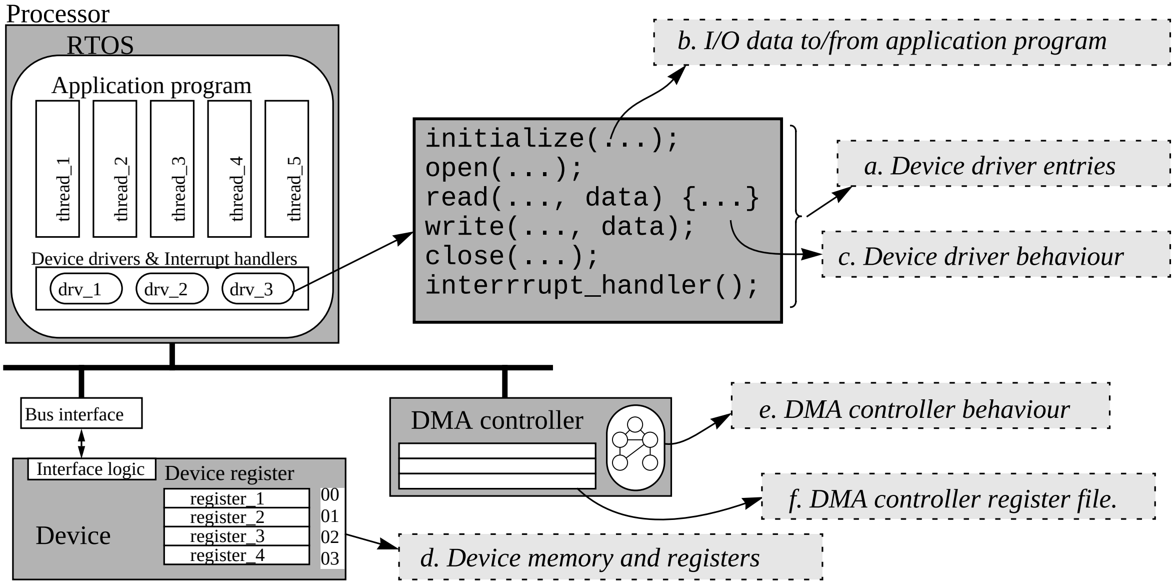


Figure 1. Break down of the HW/SW communication into different parts.

The behaviour of a controller can either be to move a block of data in one step, or move it word wise synchronized by e.g. an interrupt signal from a device.

Figure 1 illustrates the dependences of a device driver on various other parts, both hardware and software. We observe that a device driver contains information on: (a) processor architecture and behaviour, (b) real-time kernel behaviour, (c) device architecture and behaviour, (d) access protocol of the device, (e) application program interface (API) rules, and (f) DMA controller architecture and behaviour. Hence, device drivers accommodate a very high information density and are dependent on many parts which can appear in a large number of combinations. This fact is the reason for the low productivity for device driver modelling, four times lower than ordinary software code [3].

Días et al. [9] present a good example of design using IP components, in which more than 60% of the design is composed of IP components. We use this to illustrate how time-consuming the task of interfacing the IP components can be, i.e. design of the hardware/software communication parts of the system. Since this design uses a standard bus and IP components designed for this bus, the main design effort for the hardware/ software interface are the device drivers. Figure 2 shows a simplified schematic of the system, which is a pay phone controller. Consider the following design scenario: A system designer needs to evaluate what impact DMA controllers have on the system performance. This is done by comparing two alternative architectures of interfaces to the USARTs in Figure 2: (1) no DMA controller, and (2) DMA controllers to interface the USARTs. The assessment is difficult due to the many dependences and a simple calculation is not accurate enough. This is supported by the observation that in many practical situations the traffic on the bus is identified as performance bottleneck only after the system has been built. Ideally the comparison between the two

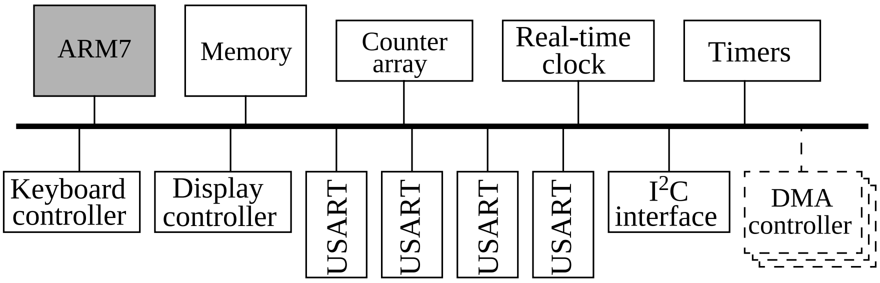


Figure 2. Simplified schematic of the pay phone controller system.

alternative architectures would be done by simulation with all the interface details included in the simulation models. However, without support tools, this is a formidable task because the designer must develop entirely new interface code for each configuration and component, due to the huge difference between implementations of a hardware/software communication protocol in pure software and with a DMA controller. This task is so time-consuming that system designers typically consider a very limited number of design alternatives and frequently only one. On the other hand, a tool is perfectly capable of generating efficient models for this purpose from a high level specification of the communication protocols, as will be shown in this paper.

In addition to the evaluation of different architectures there is the task of configuring a system for different product versions and product generations. For example, in a second version of the product it was decided to use a different processor and real-time kernel, but the functionality was the same. If the interface code was written with reuse in mind, i.e. macros for real-time kernel functions, and processor instructions are used in the code, most of the code can be reused. Only the APIs of the code have to be adapted to fit the new architecture. But a new processor will require another DMA controller. Thus, the interface code dependent of the DMA controller has to be redesigned.

Now consider the evaluation and design architecture upgrade tasks with a tool for hardware/software communication. Taking a protocol description of the interfaces the designer maps these onto an architecture, i.e. real-time kernel, processor, and DMA controller (see Figure 3). The communication synthesis procedure transforms the protocol descriptions into architecture specific device driver code and application specific DMA controllers. This enables IP providers to supply a wrapper for IP components. Thus, the designer that uses these IP components only has to use the tool to generate the interface code for the specific architecture.

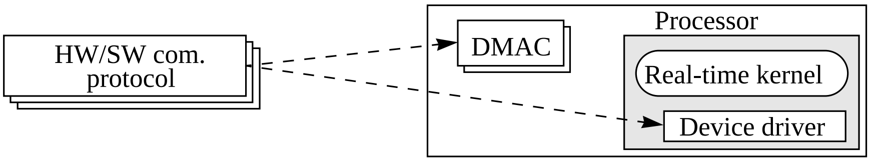


Figure 3. Mapping of hardware/software communication protocols onto selected architecture.

The next section presents the related work. Section 3 describes the design flow, target architecture and protocol modelling in ProGram. Section 4 analyses the architecture dependent parts of device drivers and proposes a way to represent them. Section 5 and 6 present the proposed synthesis methods for software and for a software/DMA solution, respectively. Section 7 contains descriptions and results from the examples used to evaluate the methods.

2. Related work

Recently, much attention has focused on the problem of communication synthesis for embedded real-time systems [28]. But none of them has fully addressed the problem described above. CoWare [4], Polis [2] and Cool [23] concentrate on the case where the whole design functionality is captured within their environment and then communication is refined during system synthesis, i.e. the device drivers are generated together with the custom HW and the operating system. But if the user wants to use IP blocks and off-the-shelf real-time operating systems (RTOS), he/she will face the same problems as with manual design [31].

CoWare [4] is a design environment for heterogeneous hardware/software systems on a chip. CoWare accepts heterogeneous communicating processes, which are specified in DFL, VHDL or C. CoWare's main focus is system integration and handling of communication in embedded systems. The heterogeneous specification is mapped onto different processors (DSP, micro controller or hardware). Inter process communication is done with point-to-point communication channels. The inter-process communication semantic is based on remote procedure call. Hardware/software communication channels are mapped onto a fixed architecture. This architecture is based on several library models implementing different I/O scenarios. For software, the communication procedures are captured as parameterized C functions that are mapped onto a software model, i.e. they adapt to processor specific I/O handling, interrupt handling, etc. For hardware, a hardware interface cell is generated to connect with a handshake protocol to an I/O control unit. This I/O control unit is a link between the processor and the handshake protocol.

Polis is focused on control-dominated applications with system architectures composed of a single processor surrounded by custom or library hardware. Polis uses Codesign Finite State Machines (CFSM) as the internal representation for a system description, separating communication, behaviour and timing of the system. The communication model is globally asynchronous, locally synchronous, with non-blocking finite buffers between CFSMs. C code and HDL code are generated from the CFSMs mapped to software and hardware, respectively. Except for the I/O drivers and code generated from the CFSMs, the software code consists of a generated application-specific operating system for the selected processor. All communication within software or between software and hardware occurs through shared memory, I/O ports or memory-mapped I/O. The synthesized hardware includes the address decoders, multiplexers, latches and glue logic. Special-purpose hardware must follow a simple, data/strobe based protocol in order to be interfaceable with other CFSMs.

In contrast to both CoWare and Polis, our approach uses abstract, implementation independent protocol specifications, and is targeted specifically towards gluing IP blocks and third party RTOS together.

In Chinook [5,6], the definition of a device driver also includes the bus interface. The driver/interface is described in a timing diagram description (SEQ), this captures both the behaviour and timing constraints for the interface. The description is synthesized into low-level software code accessing the device via the ports of a micro-controller together with the required glue-logic. The synthesis procedure tries to find the cheapest implementation (smallest amount of hardware) with regard to both timing constraints and resource constraints. Ortega et al. have in [27] expanded this approach to communication synthesis for distributed systems, i.e. from a behaviour description and an architectural mapping they synthesize interfaces for different bus structures, e.g. CAN and I2C.

MakeApp [14] 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 [31], since they do not support DMA and generate code only for a specific real-time kernel (Chinook) or no kernel (MakeApp).

Jerraya et al. use SOLAR [17] as an intermediate language, it supports system level modelling and synthesis. The communication semantics are based on the concept of remote procedure call communication via send/receive operations, similar to CoWare. Types of protocols supported include blocking and non-blocking communication. A channel is implemented by allocating communication units from a library [7]. These library units are composed to fulfil a set of constraints put on the channel. The library consists of a set of communication services and protocols along with their implementations (a mixture of hardware and software). Interface synthesis techniques are used to permit communication between the processor and the chosen communication units.

Eisenring et al. [11] present a method for modelling the target architecture by means of object oriented methods. The architectural model is used to generate hardware/software interfaces from data flow descriptions. LYCOS [16], COSYMA [12] and Vahid et al. [32] propose to solve the hardware/software communication by supplying a communication library. However, this only moves the problems of designing and maintaining device drivers from the designer to the provider of the library. The only approach that handles the mapping of hardware/software communication on to DMA controllers is described by Eisenring and Teich [11], but their approach is limited to a set of standard APIs.

Öberg et al. present a communication protocol synthesis tool that specifies the design in a special purpose language ProGram [34], which is based on context free grammar, the synthesizable subset is limited to regular grammar with attributed conditions. The specified protocols are synthesized into hardware. Seawright et al. [29] have a similar approach but instead of grammar they use a graphical version of regular expressions to specify the behaviour. This work has resulted in the commercial tool DALI. It uses a graphical interface for entering the hierarchical production rules, called frames, together with their actions. Actions are described using a host language, VHDL or Verilog. All inputs and outputs in the design entity are exactly described at clock cycle level. The output is an FSM described in VHDL.

We adopt Öberg's approach to specify communication protocols in ProGram and apply it to the specification and synthesis of the software part and DMA controllers in hardware/software interfaces. The synthesis tool requires a behavioural description of the protocol and a mapping to a particular architecture. The relevant information about architecture elements, HW or SW, are captured in two libraries. All the implementation details of the device driver and DMA controller are automatically synthesized. The communication synthesis tool is also capable of generating simulation code for commercial hardware/software co-simulators [26].

3. Methodology

In a system design methodology based on the reuse of IP-components, our method generates the implementation for a given access protocol (i.e. hardware/software interface) and maps this onto a defined architecture. The access protocol is implemented either as a pure software device driver or as a device driver with a DMA controller. This decision is made by the designer for each concurrent ProGram production rule. Inputs to the hardware/software interface synthesis procedure are the access protocol (ProGram) and libraries capturing processor and kernel specific parts. ProGram specifications are either delivered by the IP provider or the designer writes them. The ProGram specification defines the ports to the application, the ports on the device and the protocol behaviour. In this paper we concentrate on the device driver and DMA controller part of the access protocol and we do not discuss the synthesis of the bus interface.

We specify device drivers independently of the architecture using ProGram [25]. ProGram is a grammar based notation for protocol applications, which is inspired by YACC [18]. Specifications in ProGram deal with sequences of allowed events as opposed to states and state transitions in an FSM model. In contrast to parsers for compilers ProGram allows for several concurrent input and output streams. The ProGram description is synthesized into a set of untimed extended state machines [25,34], which is the input to the architecture mapping procedure described in sections 5 and 6.

Figure 4 shows that the mapping procedure uses data from two libraries to generate the architecture specific code. The first library captures the information on the operating system architecture (OSLib) and the second library captures the processor specific characteristics (ProcLib). This organization carefully separates protocol specification from operating system and processor dependences. With this input the tool can generate three alternative implementations: (I) pure software implementation, (II) code for mixed hardware/software solutions with a DMA controller, and (III) multi-level simulation models. With multi-level simulation models we mean the generation of device drivers for simulation. That is, device drivers for emulation of the hardware behaviour used in software simulation or device driver for hardware/software cosimulation of the system. In this paper we discuss only alternative I and II, and refer the reader for further details about the multi-level simulation methodology to [26].

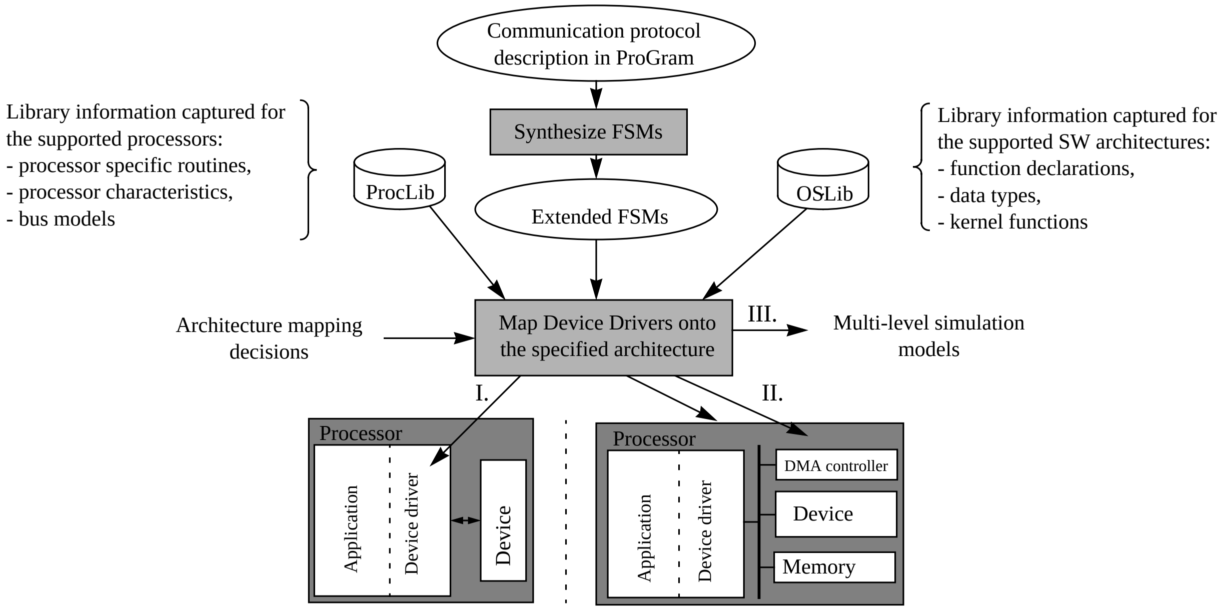

Communication protocol description in ProGram
Synthesize FSMs
Extended FSMs
Library information captured for the supported processors:
- processor specific routines,
- processor characteristics,
- bus models
ProcLib
OSLib
Library information captured for the supported SW architectures:
- function declarations,
- data types,
- kernel functions
Architecture mapping decisions
Map Device Drivers onto the specified architecture
III.
Multi-level simulation models
I.
II.
Processor
Application
Device driver
Device
Processor
Application
Device driver
DMA controller
Device
Memory

Figure 4. Overview of the device driver synthesis system.

3.1. Target architecture

Our approach to device driver synthesis supports two alternative target architectures: (1) pure software implementation and (2) software together with a generated DMA controller, see Figure 4. For both target architectures we assume that the device has already been interfaced to the system bus with a bus interface.

Software implementation: For pure software implementation, access functions and interrupt routines are implemented as software functions in C, which are mapped to fit the selected real-time kernel and processor. For the appropriate call of the interrupt routine an interrupt handler is generated.

With DMA controller: The combined DMA and software based architecture is used when the designer has selected one of the state machines to be implemented as a DMA controller which controls the data transfer between a hardware device and memory. This is the preferred solution if high peripheral data rates must be achieved, and performance and bus utilization are critical.

3.2. Protocol modelling in ProGram

A ProGram model defines the possible event sequences on several parallel inputs in terms of a grammar [34]. Actions, which are associated with grammar rules, define the reaction to input events and the event sequence on the output ports. A ProGram description consists of three parts: (1) interface declaration, (2) tokens and memories, (3) grammar rules and actions. ProGram handles only bit vector types.

Interface declaration: The interface declaration describes the interface to the application and the device. There are four types of ports to describe the device driver interface: (1) ports to the application (sw), (2) registers of the interfaced device (hw), (3) internal signals (internal) and (4) interrupt signals from the device (interrupt). In addition to the type a port declaration contains name, direction and bit width. A port declaration of a device register requires also the register's relative address.

Tokens and memories: A Token is a pattern of bits read from an input stream or written to an output stream and can also be viewed as a constant. Reading and writing tokens are the primary events. Memories and variables can be defined to maintain a state and to communicate between concurrent activities of the protocol implementation. The size and layout of memory fields can be defined.

Grammar rules and actions: The grammar description begins with one or several start rules. They define concurrent activities and work in practice as process declarations that define the signals used for the start condition, `read` in Figure 5 is such a start rule. The synthesis process generates for each start rule an access function or an interrupt routine. The read rule in Figure 5 would result in an access function.

Actions in the grammar specify assignment of values to signals, i.e. the parts enclosed by curly brackets in Figure 5. Expressions to compute these values may be put directly in the assignments or may be associated with some symbols in the action value section. The assignments can then simply refer to these symbols. The expressions allow concatenation and conditionals in addition to the usual arithmetic and logic operations. The operands can be constants, signals, other action value symbols or bit patterns recognized by grammar symbols. A grammar rule consists of a grammar symbol that serves as a rule identifier and a list of alternatives; e.g., the alternatives for the `waitReady` rule in Figure 5 are zero and one. Each alternative is a sequence of non-terminal symbols, terminals and actions. Passing the new signal stream as a parameter to the subtree of productions redirects the input stream.

```
read: bit { tmpCNTR(2..0) = channelNo; } bit {CONTROL = tmpCNTR;} waitReady;
waitReady(int):   1    waitReady
                | 0    { readData = HIGH LOW; };
```

Figure 5. Example of grammar rule with actions.

4. Architecture characteristics and modelling

To describe why different parts of the device driver are modelled in libraries we first analyse the architecture dependent parts of a device driver. Then we describe the modelling of operating system and processor characteristics for synchronization, execution delays, interrupt handling, mutual exclusion, direct memory access and the device driver interface.

4.1. Analysis of device drivers

To identify what parts of the hardware/software interface that need to be mappable during interface synthesis, we analyse the architecture dependent parts of the device driver.

Synchronization with external events: Synchronization of device drivers with external events can be implemented in three ways, as illustrated in Figure 6: (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. Polling is the only one of these three synchronization schemes that can be implemented independent of the software kernel.

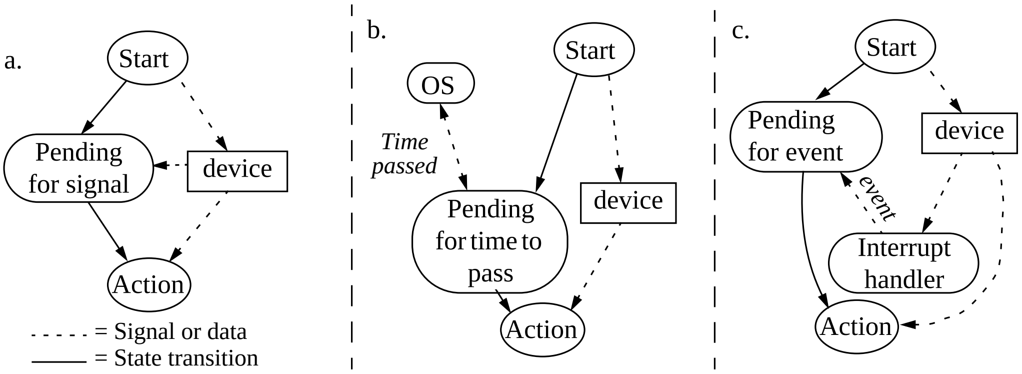


Figure 6. Synchronizing with external device.

Synchronization with internal events: Internal synchronization, i.e. synchronization between the device driver parts, is implemented using the semaphore facilities of the kernel. To prevent deadlocks the sender and receiver of the internal synchronization events have to be located in different execution flows. Hence, a sender has to be located in an interrupt routine and the receiver in an ordinary access function, since wait operations are not allowed in interrupt routines.

Delay execution: 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 a kernel, hence this function is architecture dependent.

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 performs the action itself. Interrupt handling is dependent on both the processor and the kernel. Since there is no standard way to write interrupt routines in high-level languages we use an assembler entry. There is also a kernel dependence in the case of a preemptive kernel; for this case the interrupt handler has to notify the kernel when entering an interrupt routine using kernel specific routines.

Mutual exclusion: There are two possible ways to prevent several application threads accessing the same device simultaneously: (1) by means of semaphores and (2) by disabling the interrupt mechanism. Both have their advantages and disadvantages. Disabling interrupts is fast and does not add overhead to the execution but it disables the normal behaviour of the kernel. That is, the system will be executed as a single thread and no scheduling will occur. Semaphores do not affect the kernel behaviour but introduce communication overhead. Hence, one should use semaphores for complex device drivers and disable interrupts for simple ones.

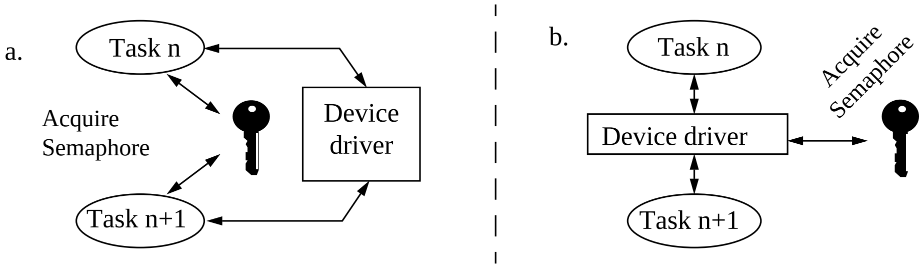


Figure 7. Handling mutual exclusion. (a) Mutual exclusion control outside device driver. (b) Mutual exclusion control inside device driver.

Both methods for controlling mutual exclusion are architecture dependent. To prevent this the control can be placed outside the device driver, as in Figure 7a. The cost of this is less structured code compared to the solution in Figure 7b, where the control is placed inside the device driver.

Device driver interface: The device driver interface defines the driver component naming style and how in and out parameters to the device driver are handled. For device drivers developed for single threaded software with no operating system or systems with small real-time kernels like µC/OS or RT-kernel, the device driver interface is determined by the designer, project or company coding style. 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 [31].

DMA controllers: DMA controllers are used to decrease the processor load, and to speed-up the transfer of large amounts of data. This is achieved at the cost of additional hardware and reusability of the device driver code, since the code not only depends on the kernel and processor but also on the DMA controller, since each DMA controller has its own unique architecture and behaviour. A DMA controller can move data between memory-memory or memory-device. From the software view, the controller is a set of registers for setting address pointers and controlling the behaviour. The behaviour can either be to move a block of data in one step, or move it word wise synchronized by e.g. an interrupt signal from a device. When a DMA transfer is finished, the controller synchronizes with the software by triggering an interrupt.

4.2. Modelling the software environment

To map architecture dependent parts of a device driver to a specific architecture, the characteristics and behaviour of the architecture are captured in two libraries, OSLib and ProcLib. They capture the operating system specific and processor specific parts, respectively (Figure 4).

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

Type	Library entry	Comment
Characteristics	name	Name of the real-time kernel
	kernel_type	{preemptive, not-preemptive, single thread}
	tick_time_period	Time between two clock ticks
	interrupt_supported	{true, false}
	header_files	A list of header files needed by the kernel
API generation	interrupt_routine_def	Script for generating interrupt routine declaration
	access_routine_def	Script for generating access driver routine declaration
	task_def	Script for generating task declaration
	type_conversion_table	Table for converting bit vector to C data types
Services	enable_interrupts	Function for enabling interrupts
	disable_interrupts	Function for disabling interrupts
	create_binary_semaphore	Function that generates a binary semaphore
	delete_binary_semaphore	Function that delete a binary semaphore
	pending_semaphore	Function for pending a semaphore
	set_semaphore	Function for sending an event with a semaphore
	pending_time	Function for waiting for a specified time
	enter_interrupt	Function for informing kernel of interrupt service start
	exit_interrupt	Function for informing kernel of interrupt service end

Table 1. Library structure used in OSLib to capture a real-time kernel.

SW environment characteristics: This part contains the environment name, type, timing, and include files needed for this kernel. The type is captured since it affects the behaviour of the interrupt routines. Characteristics captured by OSLib are defined in Table 1.

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 scripts for generation of function declarations: (1) interrupt service routine, (2) device driver function, and (3) task function. These scripts generate the function names and parameter declarations. There is an additional translation table for type conversion, i.e. for translation of bit vector types to C data types.

SW environment services: SW environment services are kernel functions for semaphore and interrupt handling. Modelled services are discussed in Section 4.1 and the system functions to provide these services are enumerated in Table 1.

4.3. Modelling of processor features

The model of the processor (i.e. the processors's characteristics and functionality) used in ProcLib is divided into three parts: (1) the processor characteristics, (2) processor specific routines, and (3) DMA components.

Type	Library entry	Comment
Characteristics	name	Name of the processor
	device_access_type	{memory mapped, port mapped}
	internal_data__bus_width	{8,16,32}
	external_data_bus_width	{8,16,32}
Routines	push_proc_register	Instruction to store processor register on stack
	pop_proc_register	Instruction for restoring processor register from stack
	enable_interrupts	Instruction for enabling interrupts
	disable_interrupts	Instruction for disabling interrupts
	return_from_interrupt	Instruction for returning from interrupt
	function_call	Instruction for calling functions
	read_port_mapped	Instruction for reading port mapped
	write_port_mapped	Instruction for writing port mapped
DMA components	master_interface	Interface for the accessing processor bus
	slave_interface	Memory interface for accessing a register file

Table 2. Library structure used in ProcLib to captured the processor specific parts.

Processor characteristics: The captured processor characteristics are the (a) processor name, (b) the internal and external data bus width, since these will affect the address calculation of the device addresses, and (c) the device access type, i.e. if it uses a port mapped or memory mapped device access method (Table 2).

Processor specific routines: Processor specific code necessary for device driver generation is code for saving and restoring of registers, code for enabling and disabling of interrupts. The last is used if the kernel does not support this feature. Table 2 defines the processor specific code captured in ProcLib.

Master interface: This is an interface between a fixed protocol (defined in Figure 8) and the captured processor's bus protocol. The port with fixed protocol on the interface is the one connected to the generated part of the DMA controller. A data access is initiated from the DMA controller who activates the master interface, which then obtains control over the processor bus and performs the data access.

On one hand it implements the processor specific bus interface, and on the other hand it implements the master interface protocol to communicate with the generated DMA controller. The master interface communicates with the generated parts with a fixed protocol, which is defined in Figure 8. Hence, this approach is independent of bus organization and arbitration controller, since arbitration handshake and bus specific behaviour is implemented by the master interface, not by the generated parts of the DMA controller.

The master interface component in Figure 8 uses a 2-phase request/acknowledge protocol. The protocol is synchronized with the bus clock used for the processor bus. A read/write cycle starts with the request (*req*)

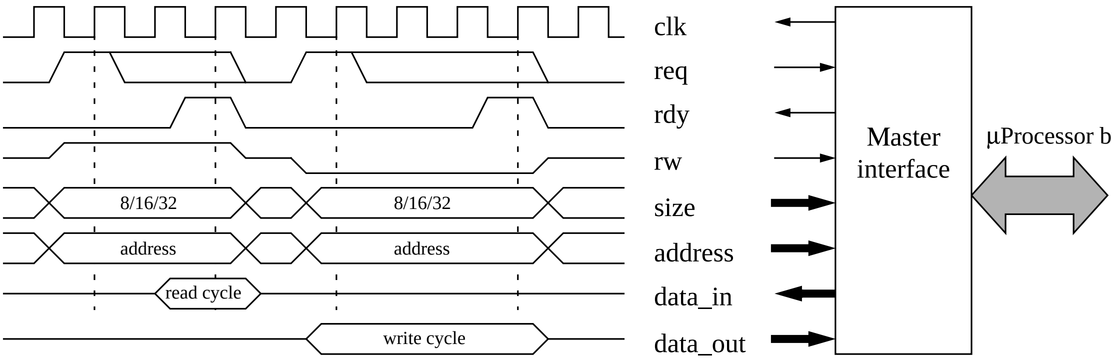


Figure 8. Master interface component and its access protocol.

signal driven high, the *rw*, *size* and *address* signals are driven to the appropriate values. A cycle is terminated by detection of the ready (*rdy*) signal at the positive clock (*clk*) edge. Read data (*data_in*) is valid only at the positive clock edges and when the *rdy* signal is high. Write data (*data_out*), *address*, *size* and *rw* signals should be valid before the positive clock edge when *req* is high and until after the positive clock edge when *rdy* is high. The *size* signal indicates the size of the transfer, i.e. it is up to the master interface to implement the reading and writing of data.

Slave interface: The slave interface is a simple memory interface that is used to enable access to the generated DMA register file (described in Section 6) from the software parts of the device driver. The register file is accessed through an ordinary asynchronous memory interface (similar to a static RAM with *read*, *write* and *chip_select* signals). In this way the set-up functions can initialize the base address and the access functions can write and read back the internal variables and pointers.

5. Software synthesis

From the ProGram description of the device driver several untimed FSMs are synthesized using the same technique used for generating HW interfaces [34]. From there the implementation is created in four steps: (1) translation of the bit vector data types used in the ProGram and FSM representations into C/C++ data types; (2) transformation of the state machines into state machines suitable for software implementation; (3) optimization of the state machines; (4) mapping of the state machines onto the selected processor/real-time kernel, generation of the application program interface, and generation of C code. These steps are indicated in the synthesis procedure depicted in Figure 9.

Map data types: Data types in the ProGram description and the generated FSM descriptions use a bit vector data type. A table in OSLib defines the translation of bit vectors into C types. This table describes how the translation from a bit vector of a specific width is represented in C.

Transform: The synthesis procedure has two transformation rules for identifying synchronization: (1) one

```
procedure generate_sw_implementation
for all Signals
   signal.map_bit_vector_types(OSLib)                                 } ① Map data types
end for
for all FSMs
   for all FSM.States
      state.apply_transformation_rules(OSLib, ProcLib)                } ② Transform
   end for
   FSM.minimize_no_states()
   FSM.minimize_no_goto()                                             } ③ Optimize
   case (type of FSM)
      ACCESS : FSM.gen_function(OSLib)
      INTERRUPT : FSM.gen_int_routine(OSLib)                          } ④ Map and generate code
      TASK : FSM.gen_task_function(OSLib)
   end case
end for
end procedure
```

Figure 9. Synthesis procedure for transforming the FSMs to FSMs for SW implementation.

for external synchronization and (2) one for internal synchronization. The objective of these rules is to transform the FSM behaviour to fit a kernel based software architecture, i.e. use the semaphore capability of the kernel for synchronization. As seen in Figure 9, both these transformation rules are applied to all states in synthesized state machines.

Synchronization points in a ProGram description are translated into self-loop states in the generated FSM, which is waiting for a synchronizing event. If a self-loop is found in one of the FSMs and its state transition conditions (*cond* and $c_1 ... c_m$) fulfil the constraints defined in Table 3, the state with a self-loop is transformed according to Figure 10 and Table 3. That is, the self-loop state transition is removed and replaced with a wait (semaphore event) statement. Depending on the condition constraints, the state is transformed into an internal or external synchronization point. With external synchronization we mean synchronization with an interrupt signal generated by the interfaced device; with internal synchronization we mean synchronization between an access function and an interrupt routine.

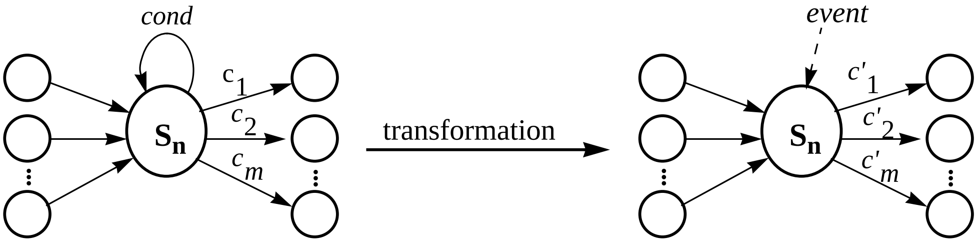


Figure 10. Template for transformation rules, where m is the number of transitions from state S_n.

A common constraint for both external and internal synchronization is that the *cond* condition (the condition for self transition) should be formed by sensing if one signal (of bit width one) is zero or one. All other conditions ($c_1 ... c_m$) should be complementary to this condition. The type of synchronization depends on the type of the signal. If the signal in the *cond* condition is an interrupt signal, then an external synchronization point is detected; if it is an internal signal, then an internal synchronization point is detected. These constraints are formally described in Table 3.

Transformation rule	Constraints	Transformations
External synchronization	• `width`(sig) = 1 • sig is an interrupt signal • cond ∈ [(sig=0) | (sig=1)] • cond = ¬[c1 ∨ c2 ∨ ... ∨ cm]	• Remove self-loop transition • Insert wait on semaphore in code of S_n • Add an interrupt routine that sends an event to wait statement
Internal synchronization	• `width`(sig) = 1 • sig is an internal signal • cond ∈ [(sig=0) | (sig=1)] • cond = ¬[c1 ∨ c2 ∨ ... ∨ cm]	• Remove self-loop transition • Insert wait on semaphore in code of S_n • Replace assignments to sig with semaphore manipulations

Table 3. Transformation rules used in the software synthesis process.

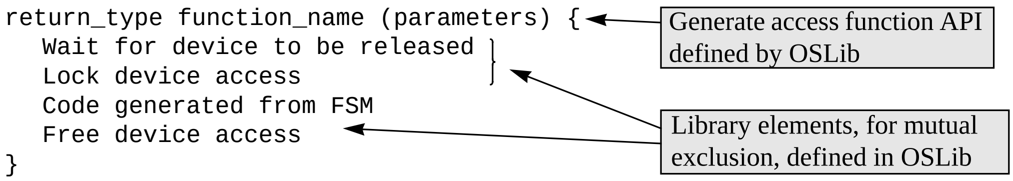
For all synchronization points the self-loop transition of the state, fitting the transformation rule, is removed and replaced with a wait for software event routine (semaphore) in the code part of the state. For the external synchronization point the event is sent by a generated interrupt routine and for the internal synchronization point all assignments (from a specified interrupt routine) to the event are replaced by semaphore manipulation routines. Figure 17 shows an example of a transformation for an external event.

Optimize: Apart from transforming the state machines to fit software architectures, the synthesis procedure optimizes the state machines for software implementation with respect to size and speed. Optimizations done are minimization of the number of states and *goto* operations. Both these optimization methods aim to minimize the code for the state transitions and, thus, decrease the code size and typically increase the performance. Minimization of the number of states in an FSM is done when the state machine is generated from the ProGram description. In the “optimize” step, state minimization goes further by removing states not necessary for software implementation, e.g. delay states. A delay state is a state whose only function is to preserve execution order i.e. it has one entry and one exit transition. The execution order in software is ensured by merging the code of the removed state to the beginning or end of the code block of the succeeding or preceding state, respectively. The optimization is partly illustrated in Figure 17. The optimizations effect on size depends on the protocol, if the state machine has many actions the code implementing the jumps will be smaller relatively to the code size. Thus, the effect of the optimization will be smaller. This is also true for the performance, but jump instructions often cost more in execution time relative to other instructions and they are executed several times. That is, the optimization is more likely to be effective with respect to performance optimization.

The synthesis procedure generates C code and the state machines are implemented with *if* and *goto* statements. By ordering the states in the code that implements the state machine the code for state transitions is minimized. By ordering states properly according to their execution sequence unnecessary *goto* statements can be avoided and code size and performance will improve. The *goto* minimization is performed such, that transitions that can be implemented by placing the states adjacent to each other, are placed first and remaining states are placed randomly.

Generate and map code: When the state machines are transformed and optimized for software implementation, the protocol description has to be mapped onto the processor and real-time kernel. This is done at the same time as the code is generated. The code generation has two different components to generate code for; (1) access functions and (2) interrupt routines. To activate the interrupt routines there is an interrupt handler that also has to be handled by the code generation. The application program interface to a device driver is a set of functions used to access the device. The layout of these APIs are determined either by the design rules or by the kernel. In the synthesis procedure the API is generated from the rules that determine how data should be passed between the application and device driver together with the scripts that generates the function name.

The function prototype of the access functions, i.e. function name and parameters, is generated as described above by using scripts in OSLib. The body code of the access function consists of routines for mutual exclusion taken from the OSLib and the C code for the FSM. The code implementing the FSM uses kernel specific routines for synchronization handling from the OSLib library.



```
return_type function_name (parameters) {
   Wait for device to be released
   Lock device access
   Code generated from FSM
   Free device access
}
```

Generate access function API defined by OSLib

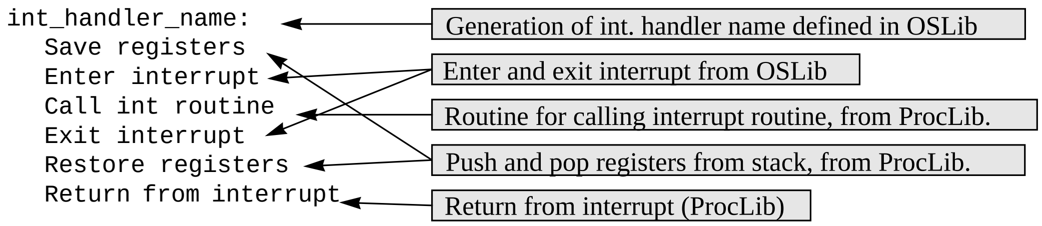
Library elements, for mutual exclusion, defined in OSLib

Figure 11. Code structure for an access function

The interrupt handler that responds to the interrupting device activates the interrupt routine. Interrupt type state machines are translated into interrupt routines. An interrupt routine has no parameters since the interrupt handler activates it without parameters. Thus, code for the interrupt handler is a parameterless function prototype and the code which implements the state machine.

Entry and exit of an interrupt handler must be in assembly language, since there is no support in C for the necessary operations. The generated code for an interrupt handler 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.

The code generation ends with the generation of an initialization function. This function creates and initializes the semaphores used by the other components. It also sets the addresses of the different device regis-



```
int_handler_name:
   Save registers
   Enter interrupt
   Call int routine
   Exit interrupt
   Restore registers
   Return from interrupt
```

Generation of int. handler name defined in OSLib
Enter and exit interrupt from OSLib
Routine for calling interrupt routine, from ProcLib.
Push and pop registers from stack, from ProcLib.
Return from interrupt (ProcLib)

Figure 12. Code structure for an interrupt handler.

ters. The initialization function takes the device base address as a parameter. The code generation also generates the declarations of semaphores and device register pointers.

A complexity analysis of the implementation of pseudo code for the synthesis procedure in Figure 9 shows a run time complexity of $O(n^2)$, where the n is the number of states for the whole description. This has also been verified by exercising the synthesis procedure with a set of random examples of different complexity [24].

6. DMA controller synthesis

Synthesis of DMA controllers from parts of a device driver description is a complement to the software synthesis in Section 5. The API for a device driver implemented in software and with a DMA controller is the same as for a pure software implementation, i.e. the only difference is the implementation of the behaviour. The DMA implementation has apart from the software code a generated DMA controller. The DMA controller consists of four blocks (Figure 13): a master interface, a slave interface, a register file and a controller. The controller and register file are both generated from the ProGram description. The register file contains pointers and variables used by the device driver. The controller is a transformed state machine generated from the ProGram description. Before the synthesis of the controller part, the designer has to select which of the device driver state machines should be implemented in hardware as a DMA controller. Both interrupt and access type state machines are candidates for DMA implementation. When this is done the synthesis procedure generates the DMA controller in three steps; FSM transformation for synchronization, FSM transformation for data accesses and generation of register file.

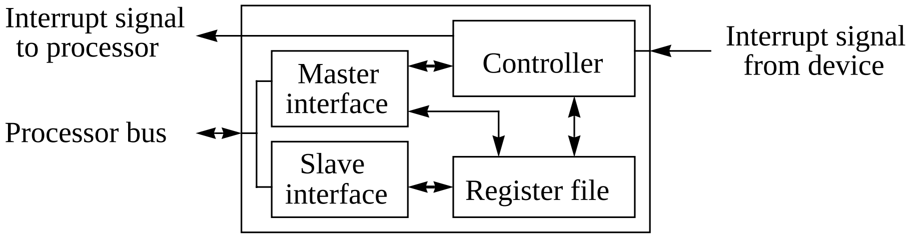
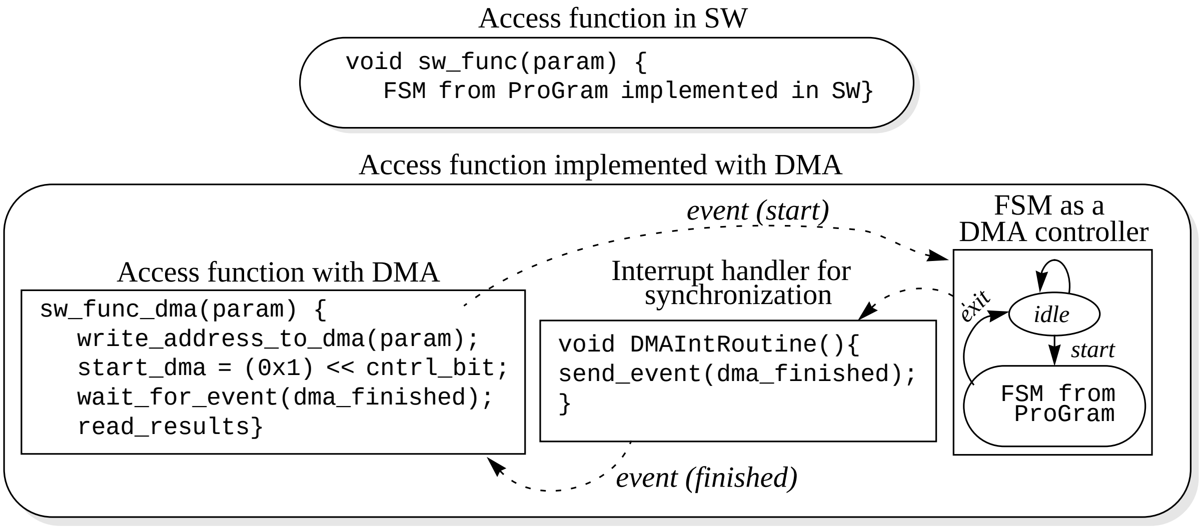


Figure 13. Architecture for the generated DMA controller.



```
void sw_func(param) {
   FSM from ProGram implemented in SW}
```

```
sw_func_dma(param) {
   write_address_to_dma(param);
   start_dma = (0x1) << cntrl_bit;
   wait_for_event(dma_finished);
   read_results}
```

```
void DMAIntRoutine(){
send_event(dma_finished);
}
```

```
FSM from
 ProGram
```

Figure 14. Access function implementation in software and with SW/DMA.

FSM transformation for synchronization: There are two types of synchronization considered here; start and stop of the whole DMA state machine and synchronization within state machines. The transformation rules are formally described in Table 4. Synchronization of access type state machines is done in four stages (see Figure 14): (1) in the generated software part of the access function the parameters are downloaded to the register file of the DMA controller. (2) send start signal to the DMA controller, sensed by an inserted wait-state. (3) when the DMA controller state machine reaches the exit transition, exit transition is redirected to the idle state, sends an interrupt to a generated added interrupt routine. (4) The generated interrupt routine sends an event to the waiting software part of the access function. Control of start and stop of interrupt type state machines is handled by inserting a wait-state before entering the state machine. The wait-state senses the interrupt signal from the device and makes a transition to the original part of the state machine when the interrupt signal is asserted.

Transformation rule	Constraints	Transformations
Access function control	• FSM of access type	• Generate SW function and interrupt handler • Add idle state before entry state of the state machine. • See Figure 14.
Interrupt routine control	• FSM interrupt routine type.	• Insert an idle state before the entry state in state machine.
External synchronization	• same as in Table 3	• Interrupt signal from device is used as synchronization signal.
Internal synchronization	• same as in Table 3 • sender is a DMA controller	• Set interrupt signal to processor. • Add an interrupt routine that receives interrupt and sends the software event.
Internal synchronization	• same as in Table 3 • receiver is a DMA controller	• Wait for control signal to be asserted • Assert control signal instead of send event

Table 4. Synchronization transformation rules used in DMA controller synthesis.

Implementation of synchronization points within and between state machines, i.e. external synchronization and internal synchronization, is handled differently for state machines implemented as DMA controllers. External synchronization, i.e. synchronizing with an event via an interrupt routine, is implemented by sensing the interrupt signal from the device directly in the state machine. Two different cases of internal synchronization exist: the event sender is a DMA controller and the event receiver is a DMA controller. For the first case, instead of semaphore manipulation, the interrupt signals to the processor are asserted. The interrupt is received by an interrupt routine that sends the event to the receiver as a software event. For the second case, the receiver of the event will wait for a control bit to change instead of waiting for a semaphore event. The sender, i.e. a software function, of the event asserts the control bit.

Transformation of data transfers: Data in memory and device registers have to be accessed through the processor bus. Thus, the state machine has to communicate with the master interface to access the memory and device. This is achieved by inserting a state for reading or writing data, see Figure 15.

Data read	• External value (memory location device register) used in expression.	• Insert read state (see Figure 15a) before the state that fit the constraint.
Data write	• Data assigned to memory or device register.	• Insert write state (see Figure 15b) after the state that fit the constraint.

Table 5. Data access transformation rules used in DMA controller synthesis.

For a data-read, data is read into a register by inserting a read state that reads the data via the master interface. The state is inserted before the state where the data is used. Assignment of variable/register is done by inserting a write state after the state where the data is written. Thus, data is written via the master interface These two transformation rules are formally described in Table 5.

a.

```
when state_n1 =>
   req <= ‘1’; rw <= ‘1’; size <= “00”
   address <= pointer + offset;
   if (rdy = ‘1’) then
      req <= ‘0’;
      next_state <= state_n2;
      tmp <= data_in;
   end if;
```

b.

```
when state_n1 =>
   req <= ‘1’; rw <= ‘0’; size <= “00”;
   address <= pointer + offset;
   data_out <= tmp;
   if (rdy = ‘1’) then
      req <= ‘0’;
      next_state <= state_n2;
   end if;
```

Figure 15. States for accessing external data.

Generating the register file: A register file for accessing the device registers and memory data is generated. The register file contains: device addresses (Figure 16a), internal variables, external variables and vector addresses (Figure 16b). A control register for monitoring the state of the DMA controller and for synchronization is also generated (Figure 16c).

The device access registers are used to generate the device register addresses in the DMA controller during execution. There is one base register that stores the device base address and several offset registers (one for each device register). The address of a device register is the sum of the base register and the offset address for a specific device register. The base register has the same bit width as the processor address bus, and the register offset is set to $\lceil \log 2(\text{size}(deviceRegFile)) \rceil$. Internal signals are either internal variables or pointers to vectors. The internal variables have the same bit width as specified in the protocol description and the pointers have the same width as the processor address bus. Addresses to vector elements are generated in the same way as device register addresses, but as a sum of the vector pointer and the index variable. The control register's least significant bit indicates if the base register address is written or not. If this address is not written, the controller can not serve any interrupts from the device since the controller does not have the address to the device. The remaining bits are used for synchronization between different parts of the device driver.

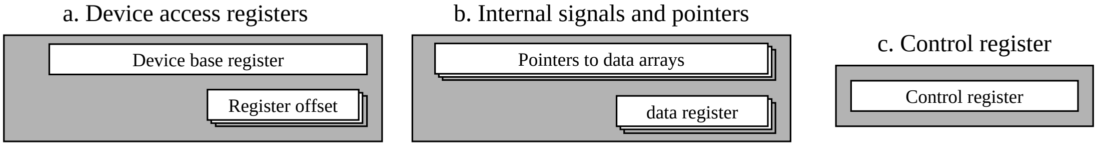


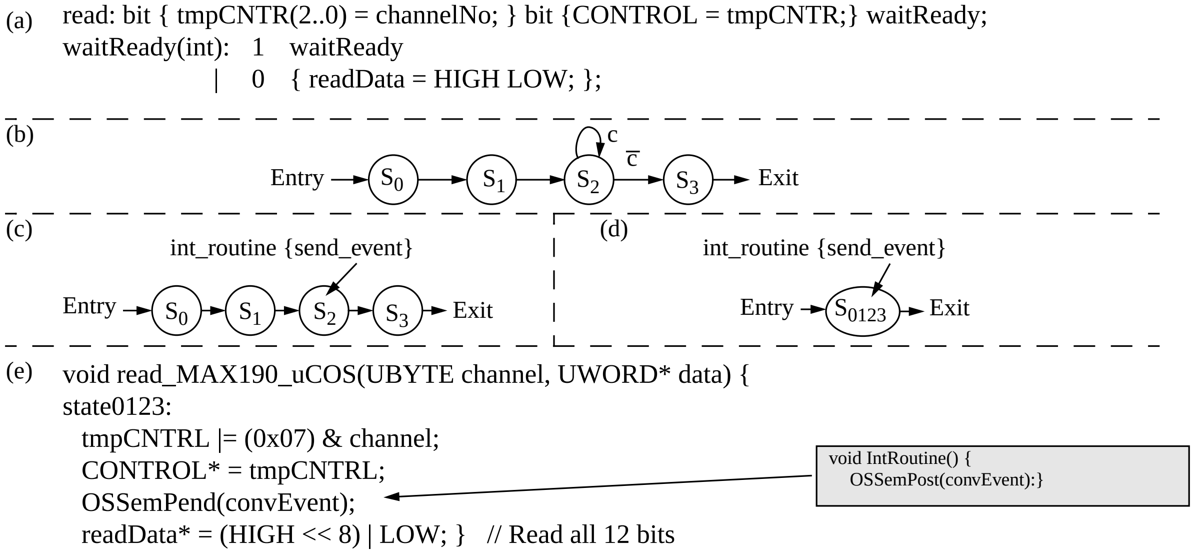
Figure 16. Register file for the DMA controller.

7. Examples

We present results from hardware/software interface protocols to five different components: three standard components (MAX197 [20] a 8 channel 12-bit ADC, MAX530 a 12-bit DAC [21] and TL16552 [30] a 2 channel UART) and two system design case studies, i.e. channel decoder in a D-AMPS base station [10] and a operation and maintenance unit (OAM) of an ATM switch on the F4 level [8].

7.1. An illustrative example

In Figure 17, a design example is transformed from a ProGram description into a software implementation. Part (a) of Figure 17 shows the ProGram protocol description, this is further described in section 3.2 and figure 5 where the very same example is used. The ProGram description starts with an assignment and it then waits for an event to occur (`waitReady`). When the event is received, data from the component is read (`readData`). Part (b) shows the result from the translation from ProGram to an FSM. The first bit statement and assignment are together represented by S0. S1 covers the second bit statement and assigment. The wait for an event statement is translated into the self-loop in state S2. Finally, state S3 is the exit-state. Parts (c) and (d) show the transformed and optimized state machines, respectively. In the transformed state machine the self-loop is replaced by a wait for a semaphore event, this is sent by an interrupt routine that is activated by the `int` signal. Part (e) of Figure 5 shows the generated code.



(a)

```
read: bit { tmpCNTR(2..0) = channelNo; } bit {CONTROL = tmpCNTR;} waitReady;
waitReady(int):   1   waitReady
                | 0   { readData = HIGH LOW; };
```

```
void read_MAX190_uCOS(UBYTE channel, UWORD* data) {
state0123:
  tmpCNTRL |= (0x07) & channel;
  CONTROL* = tmpCNTRL;
  OSSemPend(convEvent);
  readData* = (HIGH << 8) | LOW; }    // Read all 12 bits
```

```
void IntRoutine() {
   OSSemPost(convEvent):}
```

Figure 17. Example of a simple device driver function transformed from ProGram specification to C implementation. (a) ProGram specification, (b) FSM generated from ProGram, (c) FSM after applying transformation rules, (d) FSM after eliminating delay states, (e) Generated C code for the read function and generated interrupt routine.

7.2. HW/SW communication of a IS-54B base station channel decoder

D-AMPS [10] (IS-54B) is a North American digital cellular standard developed in the late 80's by the Cellular Telecommunications Industry Association. Each radio channel in D-AMPS has 30KHz bandwidth and is segmented in time into six slots using time division multiple access (TDMA), whereas 2 slots make up one voice circuit. The channel decoder, see Figure 18, receives 272 bit frames that should be processed in 6.67 ms. 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.

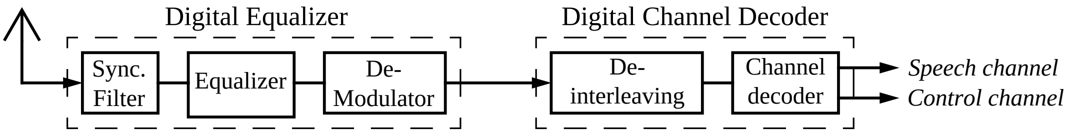


Figure 18. Receiver in IS-54B base station.

We have selected a hardware/software partition from an industrial implementation, i.e. a Viterbi decoder is implemented in hardware and the rest of the functionality is implemented in software. From a known partition we can extract the software interface (interface to the application code, Figure 19a) and a hardware interface (device register file in Figure 19b).

Figure 19 describes the interfaces of the protocol; the behaviour of the protocol is defined as follows. When the protocol is called from software the inputs from the application (*data*, *facch_polyn*, *uch_polyn*, *data_type*) are read. Then the data and polynomial signals are formatted according to what is indicated by the *data_type*

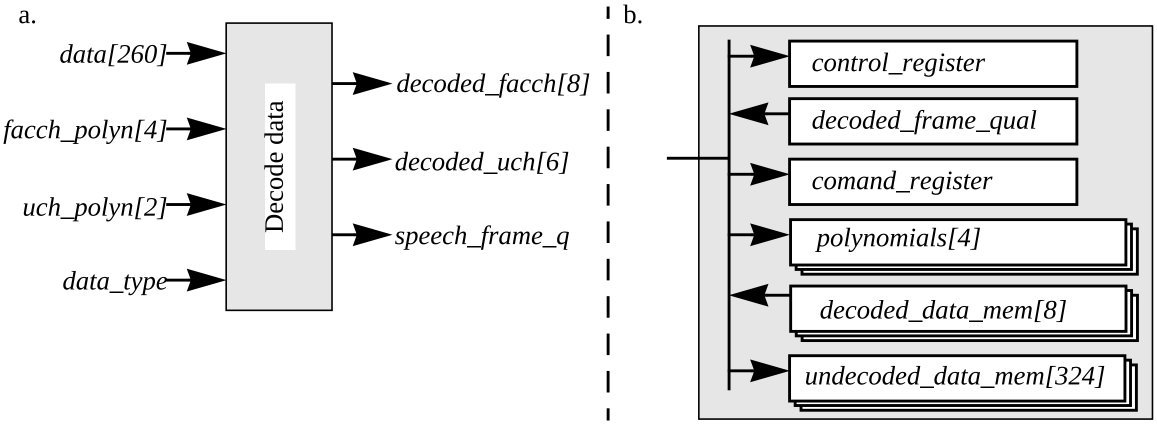


Figure 19. (a) Application (SW) interface for the decode data functionality (all signals are 16-bits wide). (b) Device (HW) interface to the decode circuit, registers and memory organized as 16-bit blocks.

signal and then loaded into the device registers. The device is given a command and starts executing while the software waits for a ready event from the device. The device can be used on the data multiple times for each access depending on the *data_type* signal. When the ready event is received by the software, the decoded data (*decoded_data_mem*) is read and formatted to the right output signal (*decoded_facch, decoded_uch*) and the *decoded_frame_qual* register is copied to the *speech_frame_q* signal.

7.3. HW/SW communication of an ATM (F4) OAM block

Our second example implements the operation and maintenance (OAM) functions corresponding to the F4 level of the Asynchronous Transfer Mode (ATM) protocol layer [8], see Figure 20. This level handles OAM functionality concerning fault management, performance monitoring, fault localization, and activation/deactivation of functions. ATM is based on a fixed-size virtual circuit-oriented packet switching methodology. All ATM traffic is broken into a succession of cells. A cell consists of five bytes of header information and a 48 byte information field. The header field contains control information of the cell (identification, cell loss priority, routing and switching information).

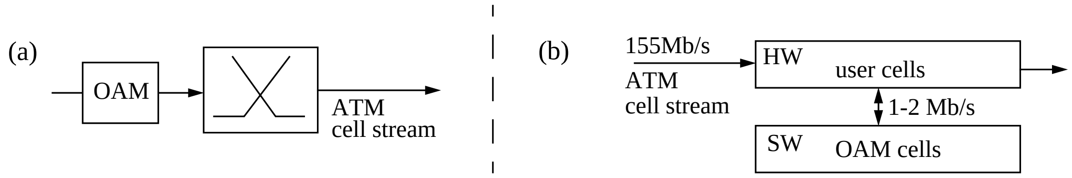


Figure 20. OAM block in an ATM network (a) and selected HW/SW partitioning (b).

Figure 21 describes the interfaces of the protocol, the behaviour of the protocol is defined as follows. The protocol consist of two parts; one for reading OAM cells and one for writing OAM cells from/to the OAM unit. The read part inputs a *gen_oam_cell* from software. Then it waits for the device to be ready to receive the frame. The data from software is then reformatted and written to the *oam_cell_in* device memory of the

OAM unit. The write part reads the *oam_cell_out* device memory and reformats it into the *oam_cell* which is delivered to the software.

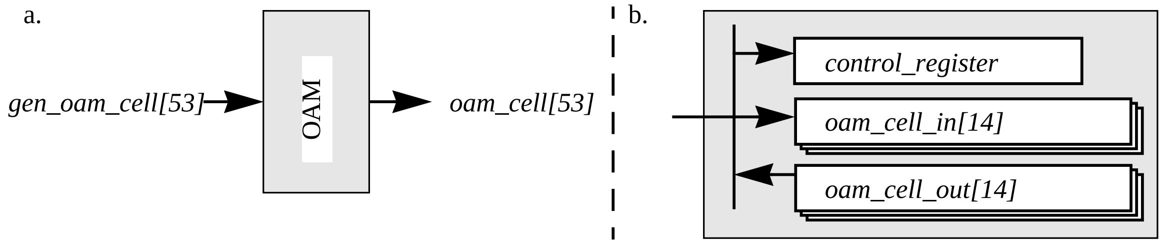


Figure 21. (a) Application (SW) interface for OAM unit (signals are 8 bits wide), (b) Device (HW) interface for the OAM unit (registers are 32 bit wide).

7.4. Results from software synthesis

The examples described above have been used to evaluate the system by comparing results generated from hand written C code (written by us) and by using our approach starting from ProGram. Several experiments have been conducted with different combinations of the examples, real-time kernels and embedded processors. Table 6 lists the seven selected configurations. They contain two processors, MC68000 [22] and ARM7 [13], and two real-time kernels, µCOS [19] and CREX [15].

#	behaviour	kernel	processor	Modelling [LoC]		Code generation time [sec.]		
				C	ProGram	ProGram to FSM	FSM to C	Total
1	MAX197	µCOS	ARM	75	20	0.09	0.90	0.99
2	MAX 530	µCOS	ARM	41	9	0.06	0.83	0.89
3	TL16552	µCOS	ARM	180	118	0.13	1.05	1.18
4	OAM unit	CREX	MC68000	75	45	0.12	1.00	1.12
5	OAM unit	µCOS	ARM					
6	Channel decoder	CREX	MC68000	254	75	0.11	1.02	0.13
7	Channel decoder	µCOS	ARM					

Table 6. Code size and code generation times for implementation examples used in Figure 22 and 23.

The first column in Table 6 indicates the implementation number used in figures 22 and 23. The second through fourth columns indicate the configuration. A comparison of the lines of code (LoC) needed to model the behaviour in C and ProGram code is shown in column 5 and 6. The difference in code size comparing Program and C code comes from the different levels of abstraction. Finally, the last three columns show the time in seconds to generate C code from ProGram on an HP 735 workstation. The code generation times (less than three seconds) show that time to generate new implementations is not an obstacle for an extensive exploration of different architectures.

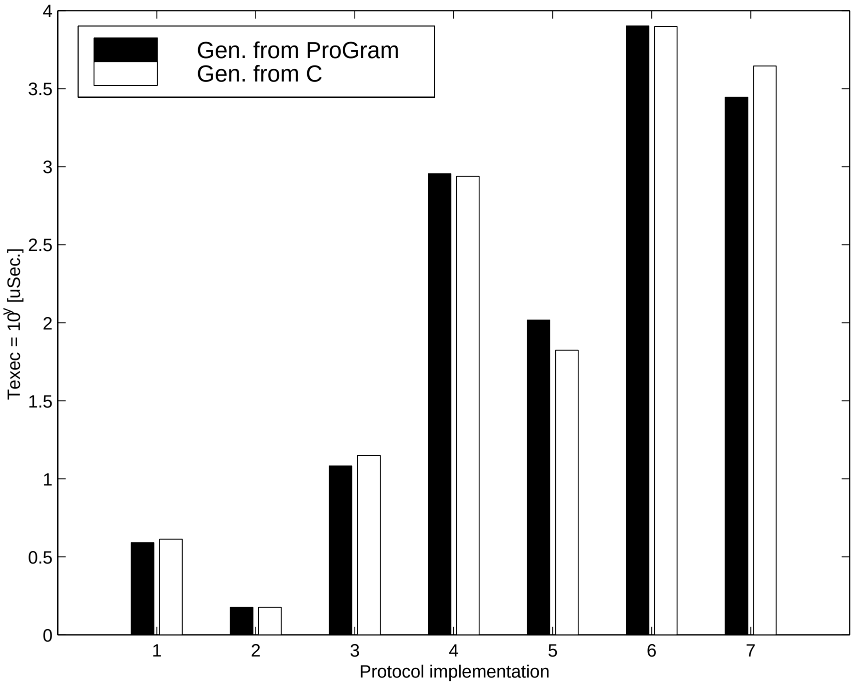


Figure 22. Performance for the different implementations described in Table 6.

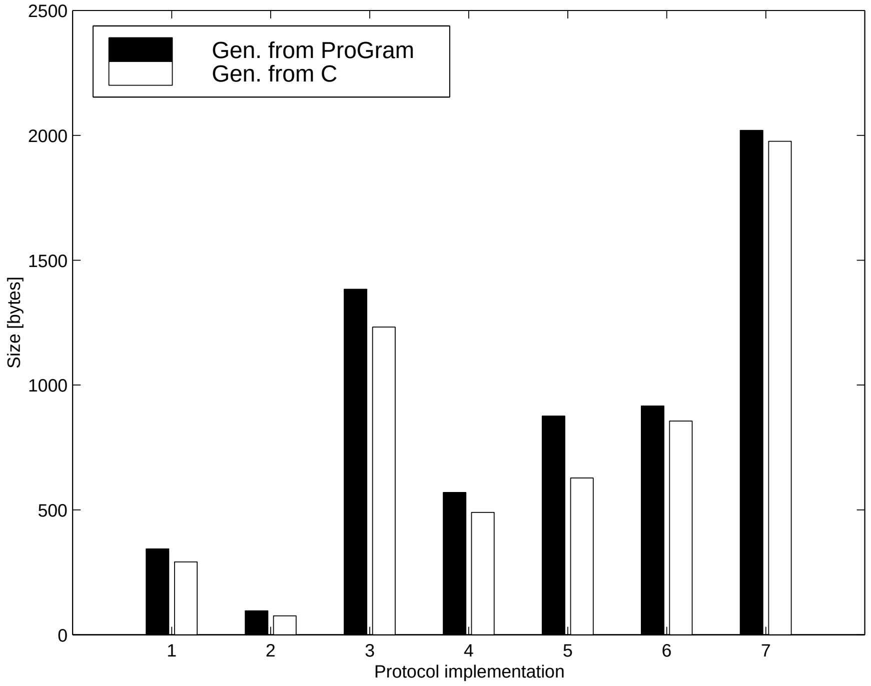


Figure 23. Code size in bytes for the different implementations described in Table 6.

Figures 22 and 23 show a comparison of the performance and code size between implementations generated from hand written C code and from ProGram. These values are captured using the configurations described in Table 7. Figures 22 and 23 show that code size and performance of device drivers generated from ProGram is similar to those manually written in C. Thus, with our tool we get similar code size and performance as for hand written code but we get the code automatically from the protocol specification. The code size numbers are generated by the cross-compilers used to generate the executable [1,15]. The code performance numbers are generated by taking a mean value from the execution of the device driver code (all functions except the initialization functions) with varying inputs. Presented performance values do not include the execution time for the real-time kernel.

Processor	Speed	Memory access time	Compiler/Simulator	Compiler directive
ARM	33 MHz	30 ns	ARM [1]	Optimize for speed
MC68000	16 MHz	125 ns	INTROL/CODE [15]	Optimize

Table 7. Compiler and simulator settings used to generate results.

7.5. Results from DMA synthesis

The components described above have been used to compose a set of protocols to evaluate the DMA controller synthesis by comparing results from the protocols modelled in hand written C code and ProGram. From the ProGram specification the tool generates C code and a DMA controller. The C code is implemented in software. The DMA controller is generated for an Z80 [33] micro processor.

Device	Synthesized behaviour	Critical path	Controller gate count	Total gate count
MAX197	1. Read buffer (x values at channel ch)^1	10 ns	1141	2100
	2. Read buffer (x values at channel ch)^2	10 ns	1156	2120
	3. Read all channels and store in buffer^1	10 ns	999	1970
	4. Read all channels and store in buffer^2	8 ns	1018	1980
MAX530	5. Write x values from a buffer^1	12 ns	1086	2150
	6. Write x values from a buffer^2	12 ns	1101	2170
16552	7. Ring buffers (Rx buffer & Tx buffer)	14 ns	1997	2960

Table 8. Synthesis results for the generated DMA controller.

Table 8 shows the synthesis results for the generated DMA controllers. The first column contains the name of the interfaced device; the second column describes the behaviour of the communication protocols. These protocol behaviours are composed of an access function and an interrupt routine. Tables 8 and 9 present results regarding the interrupt routine part of the protocol, i.e. each DMA controller corresponds to an interrupt routine. The examples are all of the type read or write blocks of data, either of blocking type (i.e. wait for data transfer to finish) or non-blocking type (no wait for the transfer to finish). Columns three through five present the synthesis results from an RTL synthesis tool for the LSI 10k technology. The third column presents the critical path for the generated controller. Columns four and five give the gate count for the generated part of the DMA controller and the whole DMA controller (i.e. the master interface from the library + generated controller), respectively.

Table 9 compares the performance and bus usage between a pure software solution and the hardware solution with an application specific DMA controller. The first column contains the name of the interfaced device; the second column gives a reference to the behaviour description in Table 8. Columns four and six give the performance (calculated as $performance = 1/\mathrm{t}_{\mathrm{exec}}$) numbers for two different configurations, software and DMA controller, respectively. Columns three and five present the numbers for system bus usage normalized to the all-software solution, this is calculated as $usage_{HW} = \mathrm{mem}_{\mathrm{access}}(HW)/\mathrm{mem}_{\mathrm{access}}(SW)$. The sixth column shows the speed-up for a hardware solution compared to a software solution, calculated as $Speedup = \mathrm{t}_{\mathrm{exec}}(SW)/\mathrm{t}_{\mathrm{exec}}(HW)$. This illustrates that the hardware solution results in 8 to 20 times higher performance. But in many cases more important is the reduced bus utilization by a factor between 10 to 20, because the processor bus is very often a scarce and highly utilized resource. The performance of the generated DMA controller is not higher than that of a standard DMA controller. The difference is, that ours is automatically generated from the same protocol description as the all-software device driver. Thus, less effort to implement a DMA based interface is required with our method.

Device	Behaviour	Pure software		With generated DMA controller		Speedup
		Bus usage[3]	Performance[4]	Bus usage[3]	Performance[4]	
MAX197	1	100 %	$2.9 \cdot 10^4$ s^{-1}	4.7 %	$5.6 \cdot 10^5$ s^{-1}	19.4
	2	100 %	$2.9 \cdot 10^4$ s^{-1}	4.9 %	$5.3 \cdot 10^5$ s^{-1}	18.4
	3	100 %	$3.0 \cdot 10^4$ s^{-1}	4.9 %	$5.6 \cdot 10^5$ s^{-1}	18.7
	4	100 %	$2.9 \cdot 10^4$ s^{-1}	8.5 %	$2.5 \cdot 10^5$ s^{-1}	8.6
MAX530	5	100 %	$4.2 \cdot 10^4$ s^{-1}	5.1 %	$6.7 \cdot 10^5$ s^{-1}	16.0
	6	100 %	$4.2 \cdot 10^4$ s^{-1}	5.4 %	$6.1 \cdot 10^5$ s^{-1}	14.5
TL16552	7	100 %	$2.9 \cdot 10^3$ s^{-1}	8.6 %	$4.4 \cdot 10^4$ s^{-1}	15.2

Table 9. Performance comparison between pure software implementation and an implementation with software generated from ProGram combined with a DMA controller generated from ProGram.

From these results we can see that the controller is quite fast (above 70 MHz) even for such an old technology as LSI 10k. The most likely application area for this is a system on a chip (SoC), but the DMA controller will work as good as a separate device on a board. With our approach, we achieve a 20 times speed-up of the communication performance. Considering the size of a system on chip can be > 1M gate, the communication speedup is achieved at a small cost of the total system area. This makes the application specific DMA controller solution an appropriate choice if communication performance is a bottleneck.

8. Conclusion

We have presented an approach to hardware/software communication synthesis. Synthesis is done from an architecture and implementation independent description of a device access protocol. The protocol description can be mapped, according to a designer decision, to a pure software device driver implementation or a mixed hardware/software device driver using a DMA controller to speed-up the data transfer. The synthesis procedure maps the generated code to fit the selected real-time kernel and processor. For parts selected to be implemented as a DMA controller the synthesis procedure generates RTL VHDL code. The generated DMA controller is connected to the processor bus via a library interface. The synthesis procedures have been tested with several realistic examples.

Our approach enables a designer to *plug-and-play* with IP components without having to go through the tedious work of writing the interface code for the different components. Instead, the designer just lets the tool generate the interface code for the selected architecture and configuration, which takes less than 3 seconds for the presented examples. The components are accessed through the generated API. This way the designer can evaluate different architectures and configurations, without spending time on rewriting the interface code. Another major benefit is the ease of design maintenance, i.e. the designer does not need to redo the interface parts for new design generations. All implementations in our approach are generated from one protocol description, regardless of kernel, processor and DMA controller. Thus, the probability of design errors is decreased since a protocol is only captured once as opposed to other approaches where the same protocol needs to be captured several times. Hence, implementations for different architectures will be consistent with each other.

Acknowledgements

This work has been supported by Mid Sweden University, the Swedish National Board for Industrial and Technical Development - NUTEK, and the Swedish Foundation for Strategic Research SSF.

References

1. ARM Software Development Toolkit - Reference Manual, Version 2.0, Advanced RISC Machines Ltd., 1995.
2. F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A, Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara, Hardware-Software Co-Design of Embedded Systems: The Polis Ap-

proach., Kluwer Academic Press, 1997.

3. B. Bohem, B. Clark. E. Horowitz, C. Westland, R. Madachy, R. Selby, “Cost models for future software life cycle processes: COCOMO 2.0”, Annals of Software Engineering, Vol. 1, pp. 57-94, 1995.
4. I. Bolsens, H. J. De Man, B. Linn, K. van Rompaey, S. Vercauteren, D. Verkest, “Hardware/Software Co-Design of Digital Telecommunication Systems”, Proceedings of the IEEE, Vol. 85, No. 3, pp. 391-418, 1997.
5. P. H. Chou, R. B. Ortega, G. Borriello, “The Chinook Hardware/Software Co-Synthesis System”, Proceedings of the International Symposium on System Synthesis, 1995.
6. P. H. Chou, R. B. Ortega, G. Borriello, “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems”, Proceedings of the International Conference on Computer Aided Design, pp. 488-495, 1992.
7. J-M Daveau, G. F. Marchioro, T. Ben-Ismail, A. A. Jerraya, “Protocol selection and interface generation for hw-sw codesign”, IEEE Transaction on Very Large Scale Integration, Vol. 5, No. 1, pp. 136-144, 1997.
8. M. De Prycker, Asynchronous Transfer Mode, Prentice Hall, 1995.
9. J. C. Días, J. Riesco, P. Plaza, “Design of an ARM based System-on-a-Chip for Pay Phones”, Proceedings of the International Workshop on IP Based Synthesis and System Design, pp. 101-105, 1998.
10. EIA/TIA Interim Standard, Cellular System Dual-Mode Mobile Station - Base Station Compatibility Standard, IS-54-B, April, 1992.
11. M. Eisenring, J. Teich, “Domain-Specific Interface Generation from Dataflow Specifications”, Proceedings of the 6th International Workshop on Hardware/Software Codesign, pp. 43-47, 1998.
12. R. Ernst, Th. Benner, “Communication, Constraints and User Directives in COSYMA”, Technical Report CY-94-2, Technische Universität Braunschweig, June 1994
13. S. Furber, ARM System Architecture, Addison Wesley Longman, ISBN 0-201-40352-8, 1996.
14. R. Grehan, “Driver Assistance”, Computer design, vol. 36, no. 1, pp. 75-80, 1997.
15. INTROL/CODE - Reference Manual, Introl Inc., Milwaukee, WI 53202 USA, 1999.
16. D. C. R. Jensen, J. Madsen, S. Pedersen, “The Importance of Interfaces: A HW/SW Codesign Case Study”, Proceedings of 5th International Workshop on Codesign, 1997.
17. A. A. Jerraya, K. O'Brien, “SOLAR: An Intermediate Format for System-Level Modeling and Synthesis”, Computer Aided Software/Hardware Engineering, Ed. J. Rozenblit, IEEE Publisher, chap. 10, 1994.
18. S. C. Johnsson, Yet another compiler compiler, Computing Science Tech. Rep. 32, AT&T Bell Lab. Murray Hill, 1975.
19. J. J. Labrosse, mC/OS - The Real-Time Kernel, R&D Publications, Lawrence, Kansas 66046, 1992.
20. MAX 197 - Multi-Range 12-bit ADC, Data sheet, Maxim Integrated Products, CA.
21. MAX 530 - Multi-Range 12-bit DAC, Data sheet, Maxim Integrated Products, CA.
22. MC68000 Family Reference Manual. Motorola Inc., 1990.
23. R. Niemann, P. Marwedel, “Synthesis of Communicating Controllers for Concurrent Hardware/Software Systems”, Proceedings of Design automation and Test in Europe, pp. 912-913, 1998.
24. M. O’Nils, “Specification, Synthesis and Validation of Hardware/Software Interfaces”, PhD Thesis, TRITA-ESD-1999-04, Kungliga Tekniska Högskolan (KTH), Stockholm, Sweden, June 1999.
25. M. O’Nils, J. Öberg, A. Jantsch, “Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces“, Proceedings of Euromicro Conference, pp. 55-58, 1998.
26. M. O’Nils, A. Jantsch, “Multi-phase Validation of Hardware/Software Interfaces based on Generated Simulation Models“, Proceedings of IEEE Workshop on High Level Design, Validation and Test, pp. 32-40, 1998.
27. R. B. Ortega, G. Borriello, “Communication Synthesis for Distributed Embedded Systems”, Proceedings of the International Conference on Computer Aided Design, 1998.
28. R. B. Ortega, L. Lavagno, G. Borriello, “Models and methods for hw/sw intellectual property interfacing”, NATO ASI on System-level Synthesis, 1998.
29. A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, J. Buck, “A System for Compiling and Debugging Structured Data Processing Controllers”, Proceedings of the European Design Automation Conference, 1996.
30. TL16552 - Dual Asynchronous Communications Element with FIFO, Data sheet, Texas Instruments Inc., 1996.
31. E. Tuggle, “Writing Device Drivers”, Embedded Systems Programming, Jan. 1993, pp. 42-65.
32. F. Vahid, L. Tauro, “An Object-Oriented Communication Library for Hardware-Software CoDesign”, Proceedings of 5th International Workshop on Hardware/Software Codesign, 1997.
33. Z80 - Microprocessor Family User’s Manual, Zilog Inc., 1994.
34. J. Öberg, A. Kumar, A. Hemani, “Grammar-based Hardware Synthesis of Data Communication Protocols”, Pro-

ceedings of the 9th International Symposium on System Synthesis, pp. 14-19, 1996.

Affiliation of Authors

MATTIAS O’NILS mattias@ite.mh.se

Department of Information Technology, Mid Sweden University, SE-851 70 Sundsvall, Sweden

AXEL JANTSCH axel@ele.kth.se

Department of Electronics, Electronic System Design Laboratory, Royal Institute of Technology, Electrum 229, SE-164 40 Kista, Sweden

Footnotes

[1] Non-blocking communication.

[2] Blocking communication.

[3] Bus usage normalized the pure software implementation.

[4] Maximum number of interrupts served per second.