Modeling and Integration of Peripheral Devices in Embedded Systems

Shaojie Wang[1] , Sharad Malik[2], Reinaldo A. Bergamaschi [3]

[1,2] Electrical Engineering Department, Princeton University, Princeton, NJ, USA
[3] IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

Abstract

This paper describes automation methods for device driver development in IP-based embedded systems in order to achieve high reliability, productivity, reusability and fast time to market. We formally specify device behaviors using event driven finite state machines, communication channels, declaratively described rules, constraints and synthesis patterns. A driver is synthesized from this specification for a virtual environment that is platform (processor, operating system and other hardware) independent. The virtual environment is mapped to a specific platform to complete the driver implementation. The illustrative application of our approach for a USB device driver in Linux demonstrates improved productivity and reusability.

1. Introduction

Device drivers provide a bridge between a peripheral device and the upper layers of the operating system and the application software. They are critical software elements that significantly affect design quality and productivity. Given the typical lifecycle of chipsets being only 12 to 24 months, system designers have to redesign the hardware and software regularly to keep up with the pace of new product releases. This requires constant updates of the device drivers. Design and verification of device drivers is very complicated due to necessity of thorough knowledge about chips and boards, processors, peripherals, operating systems, compilers, logic and timing requirements; each of which is considered to be tedious. For example, Motorola MPC860 PowerQUICC is an SoC micro-controller used in communications and networking applications. Its board support package (BSP) (essentially drivers) has 25000 lines of C code [6] – an indication of its complexity. With time-to-market requirements being pushed below one year, driver development is quickly becoming a bottleneck in IP-based embedded system design. Automation methods, software reusability and other approaches are badly needed to improve productivity and are the subject of this paper.

The design and implementation of reliable device drivers is notoriously difficult and constitutes the main portion of system failures. As an example, a recent report on Microsoft Windows XP crash data [7] shows that 61% of XP crashes are caused by driver problems. The proposed approach addresses reliability in two ways. Formal specification models provide for the ability to validate the specification using formal analysis techniques. For example, the event-driven state machine models used in our approach are amenable to model checking techniques. Correct by construction synthesis attempts to eliminate implementation bugs. Further, the formal specifications can be used as manuals for reusing this component, and inputs for automating the composition with other components.

Another key concern in driver development is portability. Device drivers are highly platform (processor, operating system and other hardware) dependent. This is especially a problem when design space exploration involves selecting from multiple platforms. Significant effort is required to port the drivers to different platforms. Universal specifications that can be rapidly mapped to a diverse range of platforms, such as provided by our approach, are required to shorten the design exploration time.

The approach presented in this paper addresses the complexity and portability issues raised above by proposing a methodology and a tool for driver development. This methodology is based on a careful analysis of devices, device drivers and best practices of expert device driver writers. Our approach codifies these by clearly defining a device behavior specification, as well as a driver development flow with an associated tool. We formally specify device behaviors by describing clearly demarcated behavior components and their interactions. This enables a designer to easily specify the relevant aspects of the behavior in a clearly specified manner. A driver is synthesized from this specification for a virtual environment that is platform (processor, operating system and other hardware) independent. The virtual environment is then mapped to a specific platform to complete the driver implementation.

The remainder of this paper is organized as follows: Section 2 reviews related work; Section 3 describes the framework for our methodology; Section 4 presents the formal specification of device behavior using the Universal Serial Bus (USB) as an example; Section 5 discusses driver synthesis; Section 6 describes our case study; and finally Section 7 discusses future work and directions.

2. Related Work

Recent years have seen some attention devoted to this issue in both academia and industry. Devil [3] defines an interface definition language (IDL) to abstract device register accesses, including complex bit-level operations. From the IDL specification, it generates a library of register access functions and supports partial analysis for these functions. While Devil provides some abstraction for the developer by hiding the low-level details of bit-level programming, its approach is limited to register accesses and it does not address the other issues in device driver development outlined above.

In the context of co-design automation, O'Nils and Jantsch [4] propose a regular language called ProGram to specify

hardware/software communication protocols, which can be compiled to software code. While there are some interesting features, this does not completely address the device driver problems as described here, particularly due to its inability to include an external OS. Other efforts in the co-design area [1,2] are limited to the mapping of the communications between hardware and software to interrupt routines that are a small fraction of a real device driver.

I2O (Intelligent Input Output) [8] defines a standard architecture for intelligent I/O that is independent of both the specific device being controlled and the host operating system. The device driver portability problem is handled by specifying a communication protocol between the host system and the device. Because of the large overhead of the implementation of the communication protocol, this approach is limited to high-performance markets. Like I2O, UDI [9] (Uniform Driver Interface) is an attempt to address portability. It defines a set of Application Programming Interfaces (APIs) between the driver and the platform. Divers and operating systems are developed independently. UDI API's are OS and platform neutral and thus source-code level reuse of driver code is achieved. Although UDI and our methodology share the common feature of platform and OS neutral service abstraction, our methodology is based on a formal model that enables verification and synthesis.

3. Methodology Framework

Devices are function extensions of processors. They exchange data with processors, respond to processor requests and actively interact with processors, typically through interrupts. Processors control and observe devices through the *device-programming interface*, which defines I/O registers and mapped memories.

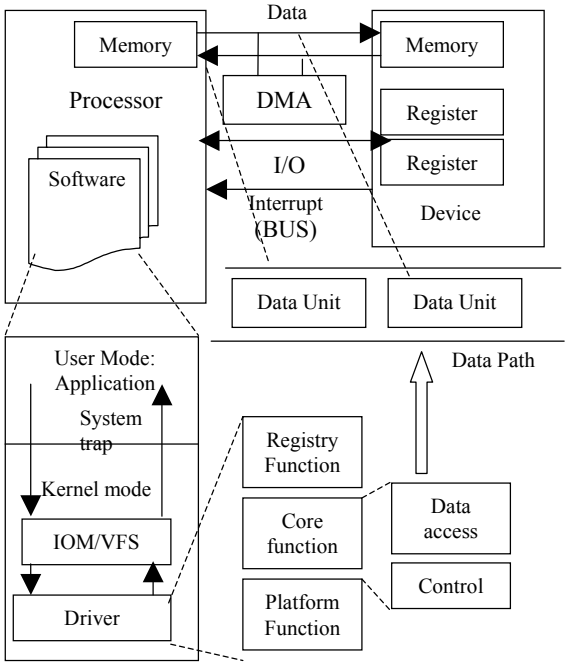

Figure 1: Driver Environment

Figure 1 sketches the relationship between devices, processors, the operating system and device drivers.

Processors access devices through memory mapped I/O, programmed I/O or DMA. A *data path* (or communication channel) is a specific path for exchanging data between the processor and the device as illustrated in Figure 1.

To hide the details of device accesses, device drivers are designed to be a layer between the high-level software and low-level device. In most cases, a device driver is part of the kernel. Application software, which resides in user space, uses system calls to access the kernel driver. System calls use traps to enter kernel mode and dispatch requests to a specific driver. Hence we can partition a device driver into three parts as illustrated in Figure 1 and explained below:

- *Core functions*, which trace the states of devices, enforce device state transitions required for certain operations, and operate data paths. Because such actions depend on the current states of the device, synchronization with the device is necessary. Common synchronization approaches are interrupts, polling and timer delay. In our approach, core functions are synthesized from a device specification. They interact with a platform independent framework called *virtual environment*. The device specification itself is explained in Section 4, and synthesis of core functions in Section 5.
- *Platform functions* that glue the core functions to the hardware and OS platform. The virtual environment abstracts architectural behaviors such as big or little endian, programmed I/O, memory mapped I/O. It also specifies OS services and OS requirements such as memory management, DMA/bus controllers, synchronization mechanisms etc. This virtual environment is then mapped to a particular platform (hardware and OS) by providing the platform functions that have platform specific code for the above details.
- *Registry functions* that export driver services into the kernel or application name space. For example, the interface can register a driver class to the NT I/O manager (IOM) or fill one entry of the VFS (Virtual File Switching) structure of the Unix kernel.

Figure 2 outlines our framework and illustrates how the three parts of the driver come together. It takes as input (1) the device

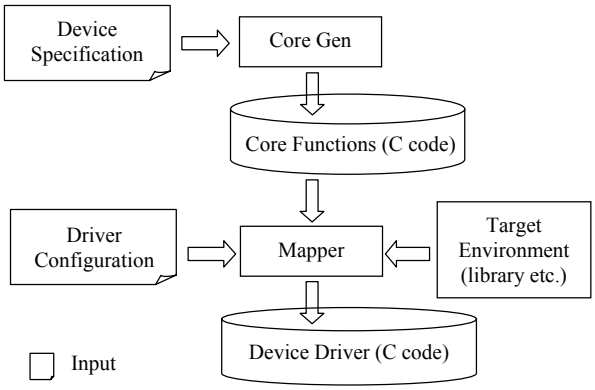

Figure 2: Framework Overview

specification, which is platform neutral and (2) the driver configuration, which specifies the device instance and environment; and outputs the device driver (C program) for a particular device in a particular environment.

The device specification provides all the information about the device required by the driver core function synthesis process (Core Gen). Core functions are implemented in the virtual environment. The *mapper* maps core functions in the virtual environment to the target environment. This mapping process does (1) platform mapping, by binding virtual OS service functions to the targeted OS, and virtual hardware functions to the targeted hardware; (2) registry mapping by translating core functions to the OS specific registry functions. It does this using the developer specified *driver configuration*. The driver configuration defines the target platform and necessary parameters. It includes the OS name, processor name, bus name, device instance parameters (such as interrupt vector number, base address), driver type (such as char device, network device), driver

```
.drv ep2_drv ( .config {
.os Linux 2.4.x  .processor SA1100
.irq 13          .io_base 0x80000000
.io_ext 0x30     .drvType Ethnet
… ## .drv ep2_drv
} ) { … }
```

Figure 3: SA1100 USB Device Controller (UDC) Driver Configuration

type specific parameters (such as maximum transfer unit for a network device) etc. These configurations are specified by keywords. Figure 3 shows the sketch of an example. This direct specification is sufficient for the mapper to target the core functions to a specific platform. While this requires the specification writer to have specific knowledge of the various elements of the configuration, information such as driver type and driver type specific parameters are encoded once and then reused across platform specifications.

4. Device Specification

In this section we describe the different parts of the device specification, which is the input to the generation of the core functions (see Figure 2). Based on our study of devices and device drivers, we partition this specification into the following parts: data path, device control, event handlers, core functions and device programming interface. Figure 1 provides some motivation for this division. The data path describes the transfer of data between the processor and the device. The device control describes the transitions between the states of the device using event driven finite state machines (EFSM) [5]. The processing of events in the event driven state machines is specified in event handlers. The core functions provide the device services to the upper layers of the software – the OS and the application. The device-programming interface provides the low-level access functions to the device registers. Eventually it is the core functions that need to be synthesized in C as the driver – however, this synthesis will need all the other parts of this specification to understand the complete device behavior. This partitioning of the specification allows for a separation of the various domains, which the device driver has to interact with, and is similar to the approach used by expert device driver writers.

Data Path

A *data unit* is a data block moving between the device and the software by a primitive hardware operation. For example, we can define a DMA access as a primitive hardware operation. A data unit is modeled as a tuple of data unit size, the event enabling a new data unit operation, starter function, the event indicating the end of a data unit operation, stopper function, the events indicating errors and access operations. Starter functions are the operations performed before data accesses. Stopper functions are the cleanup operations performed when the hardware data access operations end. Figure 4 defines the DMA data unit of the transmit

```
## DMA transmit FIFO
.dma udc_wt = dr_fifo@ep2_space .o
  .setup %{ IVAR CS2 TPC FLIP;
       IREG_IMP_WRITE(IDMA_SIZE-1); %}
  .cleanup    %{ IVAR_CS2_SST_FLIP; %}
  .abort  ep2_not_ready
              %{ IVAR_CS2_SST_FLIP; %}
  .abort  ep2_tpe_or_tur
              %{ IVAR_CS2_SST_FLIP; %};

## data unit
  .du ep2_du = udc_wt[64];

## data path
  .dp ep2 = ep2_du .o .interface
   .block %{ IVAR_CS2_FST_WRITE(1); %}
   .reset %{ IVAR_CS2_FST_WRITE(0); %};
```

Figure 4: SA1100 UDC Transmit Channel Data Unit and Data Path Specification

channel of the SA1100 USB device controller (UDC). While the details of the specification are beyond the scope of the paper, this illustrates the nature of the information required here.

A *data path* is defined as a stream of data units. The last part of Figure 4 provides an example of a data path specification. It specifies data transfer direction and the data unit of the data path.

Device Control

The *device control* specifies the state changes of a device. Because a device may be composed of several sub-devices operating in parallel, we use a concurrency model, event driven finite state machines [5], to formally model the device control. It has several *synchronization properties*: (1) the execution of event handlers and state transitions are free of race conditions, and (2) finite state machines do not share data and communicate through events if necessary. The synthesizer enforces the race condition free property by disabling the context switches appropriately. Figure 5 gives the sketch of an example of device control specification of a sub-component, USB setup protocol, of SA1100 UDC. Again, detailed syntax and semantics of the specification are beyond the scope of this paper, we will focus on illustrating the salient features of the specification.

There are four types of events: *hardware events*, *input events*, *output events* and *internal events*. Devices generate hardware events to indicate changes of hardware states. A typical hardware event is an interrupt. Higher layer modules send input events (also called service requests) to the driver. Events sent to higher-layer modules are called output events. As an example, USB host assigns a USB address to the UDC. The driver can emit an output event that carries the address information when the host assigns the address. The upper-layer software observes the USB address of the device through the output event. All other events are internal. Events can be considered as messages conveying information. In addition to the event name, an event

may convey information by carrying a parameter. Events are stored in a global event queue.

As shown in Figure 5, an event handler handles a particular

```
.fsm usb_protocol {
## The start state is Disabled. It accepts event start.
## Event handler is enclosed in %{ and %}. The
## destination state is ZombieSuspend.

Disabled start ZombieSuspend %{
     IVAR_CR_UDD_WRITE(0);
     iudelay(100);
     IEvtOut(ep1_reset);
     IEvtOut(ep2_reset);
   %};
     …
}

##  Control function. The request event is start. This
##  function moves the usb_protocol state machine to
## ZombieSuspend state, ep1 and ep2 to idle state.
##  ( Note: ep1 and ep2 are the implicit state
##  machines for data flows ep1 and ep2. )

.ctrl start (ep1, idle) (ep2, idle)
(usb_protocol, ZombieSuspend);
```

Figure 5: SA1100 UDC USB Setup Specification

event for a particular state of a state machine. The event handlers of a finite state machine may share data. Event handlers are non-blocking. As a result, *blocking behaviors are explicitly specified by states*. To remove a blocking operation from an event handler, the specification writer can restructure the state machine by splitting the handler at the operation and inserting a new state. Thus, we describe synchronization behaviors declaratively using states, rather than procedurally, which enables better checking. Specifically, interrupt processing is modeled as a couple of hardware event handlers.

Control Function Specification

A core function is responsible for providing the device services to the upper layers of software. As illustrated in Figure 1, core functions are responsible for managing data accesses and manipulating device states. Data accesses follow well-defined semantics, which specify, for example, that a block of data has to be transferred between the processor and device. Control functions are responsible for changing the control state of the device according to the state machines. As illustrated in the lower part of Figure 5, a *core control function* has an input event and a final state set. It accomplishes specific functions by triggering a sequence of transitions. Because a transition may emit multiple output events, multiple transitions can be enabled at one time. The core control function selectively fires eligible transitions, i.e. finds a transition path from the current state to the final state set. For example, the *start* function of SA1100 UDC is defined as the union of *start* event and a set of states, as shown in Figure 5. When the application calls the start function, it generates the input event (service request) *start*. It then finds a transition path to states (usb_protocol, ZombieSuspend), (ep1, idle), (ep2, idle) and completes. If the current state does not accept the *start* event, the function returns abnormally. Timeout is optionally defined to avoid infinite waiting.

Although a data flow does not have an explicit control state machine specified, a state machine is implicitly generated for it to manage the control states such as reset and blocking.

Device Programming Interface Specification

To model device register accesses, we use the concepts of *register* and *variable* (see Figure 6) - similar definitions can be found in Devil [3]. Event handlers and core functions access the

```
## Eight bit register cs2 is at offset 0x18. The 6th and
##    7th bits of udccs2 are reserved and read as 0.
.reg cs2 = base[0x18] .mask <00----->
        .slow : 8 bit;

## Variable definition for register cs2: except for
## the two reserved bits, each bit of register cs2
## is a variable. We generate functions such as
## "set", "clear" for them.
.vars cs2 [ *, *, fst, sst, tur, tpe,
            tpc, tfs];
```

Figure 6: SA1100 UDC Transmit Channel Register and Variable Specification Example

device registers through a set of APIs such as *IREG_name_READ* and *IREG_name_WRITE* that use these registers and variables. These API's are synthesized from the register and variable specifications, and extend the virtual environment. In addition to a basic programming interface, we provide additional features that enable common access mechanisms to be described succinctly. For example, FIFO is a common device-programming interface. It is always accessed through a register. Different devices use different mechanisms to indicate the number of data in the FIFO. To enable the synthesis of FIFO access routines, we have defined the concept of a hardware FIFO mapped register that is not illustrated in detail here because of limited space.

5. Synthesis

Given all the parts of the specification, the synthesis process synthesizes the C code for the entire driver. The functions that need to be provided are the core functions that provide the device services to the upper layers of software. In synthesizing these functions, all parts of the specification are used. This section outlines the synthesis process.

5.1 Platform Function and Registry Function Mapping

The platform interface includes fundamental virtual data types and a basic virtual API for the following categories:

(1) Synchronization functions,
(2) Timer management functions,
(3) Memory management functions,
(4) DMA and bus access functions,
(5) Interrupt handling (setup, enable, disable, etc.),
(6) Tracing functions,
(7) Register/memory access functions.

All virtual data types and API are mapped into platform specific types and API. This part is done manually based on an understanding of the platform. Note that while this is not synthesized, this approach provides for significant reuse as this needs to be done only once for each platform (or part of a platform).

Figure 7(a) illustrates a platform interface abstraction,

```
/*synchronization object */
Data Type: iblk_unit

Operations:
IDECLARE_BLK_UNIT(name)
/* block on bu */
inline void iblk(iblk_unit bu);
/* unblock */
inline void iunblock(iblk_unit bu);
/*check if any one is blocked on bu*/
inline int ichk_blk(iblk_unit bu);
```

Figure 7(a): synchronization primitive abstraction

synchronization primitive, in the platform independent virtual environment. Figure 7(b) shows the mapping of it on Linux kernel

```
typedef wait_queue_head_t iblk_unit;

#define IDECLARE_BLK_UNIT(name) \
     DECLARE_WAIT_QUEUE_HEAD(name)
inline void iblk(iblk_unit *bu)
 {  sleep_on(bu);   }

inline void iunblock(iblk_unit *bu)
 {  wake_up (bu);    }

inline int ichk_blk(iblk_unit *bu)
 { return(waitqueue_active(bu)); }
```

Figure 7(b): Linux 2.4 Mapping of the Primitive Specified in Figure 7(a)

2.4 that implements the interface defined in Figure 7(a) with Linux 2.4 kernel data type and functions. For lack of space, we do not show the mappings on other platforms.

We adopt a template-based approach to map the registry function interface to a specific platform by creating a library of platform specific templates. The synthesizer generates appropriate code to tailor the template for a particular device. Although the registry function generation is platform specific, it is reused for drivers for different devices.

5.2 Driver Core Function Synthesis

As illustrated in Figure 1, device driver core functions are either data access functions or control functions. We synthesize data access functions from the data path specification, and control functions from the device control specification.

Section 3 mentioned that driver core functions synchronize with the device through one of the 3 synchronization mechanisms: interrupt, poll and timer delay. As a result, our driver core function synthesizer synthesizes both the driver functions and the synchronization routines that are sufficient to completely define the platform independent part of the driver.

Synchronization Mechanisms Let us consider the synchronization mechanisms first. An interrupt is essentially an asynchronous communication mechanism whereby the interrupt handler is called when the interrupt occurs. Both polling and timer delay are either asynchronous or blocking. For example, a timer delay can be a busy wait that is blocking. On the other hand, we can register a routine that is executed by the system when the timer expires which is an asynchronous behavior. For brevity, we only describe the synthesis of asynchronous communication routines (e.g., interrupt handlers for interrupts), which are critical to synchronization mechanism synthesis.

Data Access Function A data access function in our framework is asynchronous: it does not wait for the completion of data access but returns right after the data access is enabled. This asynchronous behavior enables the overlap of data transfer and computation. A callback function can be registered to notify the completion of a data transfer. Synchronous communication is achieved by synchronizing the caller and the callback function.

A data path is modeled as a stream of data units. Hence, the data access function is implemented by iterating data unit accesses until the entire data block is transferred. If an error occurs, the whole data path access aborts. The device control can block or reset a data path.

Control Function A control function moves the device to a particular state. The execution of application software depends on such a state change. Hence, a control function is blocking (synchronous) in our framework, i.e., it will not return unless the final state set is reached. The control functions do not return values. The device is observed through output events, as illustrated in Section 4. We declare a variable for each output event. Device status is observed by reading such variables.

```
void ctrl(DEV_PRIV_DS  *priv,
          IEsmEvtVal val) {
  Enqueue the input event;
  while (true)  {
    while(firable transitions exist){
      Fire the transition;
      if(final state set is reached)
        return;
    }
    block(time_out_value);
  }
}
```

Figure8: Control Function

A control function involves a series of state transitions. It completes when the final state set is reached. Since our model is based on event driven finite state machines, a control function

```
void interrupt() {
 clear the interrupt;
 Enqueue the hardware event;
 if (event queue was not empty)
   return;
 if (a process/thread is blocked) {
   unblock it and return;
 }  else {
   while (firable transitions exist)
     Fire the transition;
 }
}
```

Figure 9: Interrupt Handler

essentially consists of a sequence of event firings. The state machines are checked to see if the final state set of the control

function is reached. If the control function is not time out and the final state set is reached, it returns successfully.

Execution of EFSM We schedule the finite state machines using a round-robin scheduling policy. The events are consumed in a First Come First Served manner. A multi-processor safe, non-preemptive queue is implemented. Driver functions are executed in the context of a process/thread. We view the execution of interrupt handlers as a special kernel thread. Hence, the essence of the core function synthesis is to distribute the event handler executions and state transitions to the interrupt handlers and driver functions. We dynamically make this decision by checking whether there is a blocked process/thread. If yes, the driver function does the work; otherwise, the interrupt handler does it. Figure 8 and 9 show the sketches of control functions and interrupt handlers respectively.

6. Case Study

The Intel StrongArm SA1100 [10] is one of the more popular microprocessors used in embedded systems. Its peripheral control module contains a universal serial bus (USB) endpoint controller. This USB device controller (UDC) operates at half-duplex with a baud rate of 12Mbps. It supports three endpoints: endpoint 0 (ep0) through which a USB host controls the UDC, endpoint 1 (ep1) which is the transmit FIFO and end point 2 (ep2) which is the receive FIFO.

The Linux UDC (USB device controller) driver (ported to Strong-Arm SA1100) implements USB device protocol [11] (data transfer and USB connection setup) through coordination with the UDC hardware. An Ethernet device driver interface has been implemented on top of the USB data flow.

To evaluate our methodology, we modeled UDC and synthesized a UDC Linux device driver that has an Ethernet driver interface and manages data transfer and USB connection setup. UDC ep1 is modeled as an OUT data path. Its data unit is a data packet of no more than 64 bytes transferred via DMA. Similarly UDC ep2 is modeled as an IN data path. The UDC control has 4 state machines: one state machine for each endpoint and one state machine managing the USB protocol state of the device. The state machines for data flow ep1 and ep2 are implicitly generated. Table 1 shows a comparison of code sizes for the original Linux driver, the specification in our framework and the final synthesized driver.

	Linux UDC Driver	*Specification*	*Driver Synthesized*
Line Count	2002	585	2157

Table 1: Comparison of Code Size

The reduction in code size is one measure of increased productivity. More importantly, the format and definition of the specification enables less experienced designers to relatively easily specify the device behavior. The declarative feature of our specification also enables easy synthesis of drivers of different styles. For example, the original UDC character device driver interface (without comments and blank lines) has 498 lines of code while our specification only requires a few extra lines. Furthermore, the code synthesized is only slightly bigger than the original code.

The correctness of the synthesized UDC driver is tested on a HP iPaq3600 handheld, a SA1100 based device. We setup a USB connection between the handheld and a 686 based desktop through a USB cradle. Familiar v0.5.3 of Linux kernel 2.4.17 is installed on the iPaq and RedHat 7.2 of Linux kernel 2.4.9 is installed on the desktop. We compiled the synthesized code to a loadable kernel module with a cross-compiler for ARM, loaded the newly created module on the iPaq, bound IP socket layer over our module and successfully tested the correctness with the standard TCP/IP command *ping*.

7. Conclusions and future work

Device driver development has traditionally been cumbersome and error prone. This trend is only worsening with IP based embedded system design where different devices need to be used with different platforms. The paper presents a methodology and a tool for the development of device drivers that addresses the complexity and portability issues. A platform independent specification is used to synthesize platform independent driver code, which is then mapped to a specific platform using platform specific library functions. The latter need to be developed only once for each component of the platform and can be heavily reused between different drivers. We believe this methodology greatly simplifies driver development as it makes it possible for developers to provide this specification and then leverage the synthesis procedures as well as the library code reuse to derive significant productivity benefits.

In the future, we plan to exploit the similarities between devices of the same type. With an appropriate inheritance mechanism, the reusability of device specification will be largely increased. We will also study optimization issues such as code size, performance and power consumption.

8. References

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, June 1997.

[2] I. Bolsen, H. J. De Man, B. Lin, K. van Rompaey, S. Vercauteren, and D. Verkest, "Hardware/software co-design of digital telecommunication systems", *Proceeding of the IEEE*, Vol. 85, No. 3, 1997, pp. 391-418.

[3] F. Merillon, L. Reveillere, C. Consel, R. Marlet and G. Muller, "Devil: An IDL for hardware programming", *4th symposium on operating systems design and implementation*, San Diego, October 2000, pp. 17-30.

[4] M. O'Bils, and A. Jantsch, "Device driver and DMA controller synthesis from HW/SW communication protocol specifications", *Design Automation for Embedded Systems*, Kluwer Academic Publishers, Vol. 6, No. 2, April 2001, pp. 177-205.

[5] E. A. Lee, "Embedded Software," to appear in *Advances in Computers* (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[6] http://www.aisysinc.com, November 2001

[7] www.microsoft.com/winhec/sessions2001/DriverDev.htm, March 2002.

[8] http://www.intelligent-io.com, July 2002

[9] http://www.projectudi.org/, July 2002

[10] http://www.intel.com/design/strong/manuals/27828806.pdf, October 2002

[11] http://www.usb.org/developers/docs.html, July 2002