Chapter 8

HIGH-LEVEL DEVELOPMENT, MODELING AND AUTOMATIC GENERATION OF HARDWARE-DEPENDENT SOFTWARE

Gunar Schirner, Rainer Dömer and Andreas Gerstlauer

Abstract With the increasing software content in modern embedded systems, software development clearly dominates the design cost. The development of Hardware-dependent Software (HdS) is especially challenging due to its tight coupling with the underlying hardware. Therefore, automatic generation of all embedded software including the HdS is highly desirable to meet today's shortened time-to-market demands.

In this chapter, we describe a system-level design approach that offers a seamless solution for generating embedded software, starting from an abstract specification and going to an implementation. In our high-level development environment, the application is developed in a platform-agnostic format that hides most implementation detail. The target platform and the mapping of the application to the platform are described separately. A system compiler then automatically generates a system model at the transaction level for performance analysis and development. The same system model later serves as an input to a software generation process, which generates the final binaries for all processors in the system. These binaries include the application, device drivers, and operating system code.

Using a design flow with automatic software generation offers significant productivity gains. At the same time, it allows the designer to focus on the algorithms without being burdened by implementation-level detail.

Keywords: System-level Design, Development Environment, Firmware, Software Generation

W. Ecker, W. Müller and R. Dömer (eds.), *Hardware-Dependent Software*,

8.1 Introduction

Software development starts dominating the design cost of modern complex Multi-Processor System-on-Chip (MPSoC). The software content is increasing since it allows to flexibly implement complex features and to quickly react to customer demands. In this context, Hardware-dependent Software (HdS) is especially challenging, due to its tight coupling with the underlying hardware (HW). Traditional approaches of manually implementing HdS become very time consuming. With a large amount of implementation detail, a manual implementation is tedious and error prone. Additionally, validating and debugging software executing on real hardware delays this important process until the availability of the final hardware platform. This hinders a parallel development of hardware and software and may result in missing the tight time-to-market constraints. On the other hand, a validation using low-level instruction set simulation suffers from a slow simulation, especially in a multi-processor context.

To increase productivity, we envision an integrated design flow that eliminates the need for low-level programming. In this chapter, we propose high-level HdS development that hides HW dependencies from designers and allows focusing on algorithms without being burdened by driver-level details.

In our high-level environment, as outlined in Fig. 8.1, the application is developed in a platform-agnostic specification written in a System-Level Design Language (SLDL). The specification model consists of a hierarchical process graph containing sequential C code in each process. In the hierarchy, processes are composed in a parallel-sequential fashion. Communication between processes is captured in abstract communication channels and shared variables, independent of their later implementation.

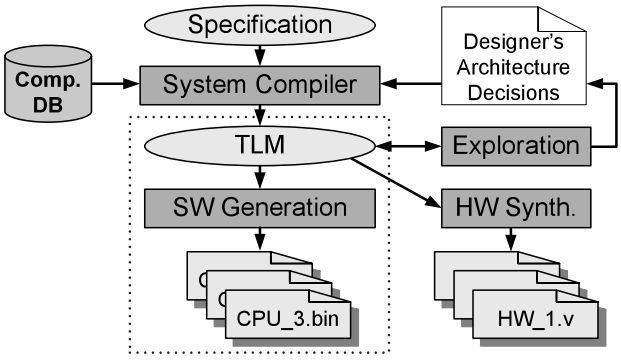

Figure 8.1. System design flow overview.

The targeted hardware platform is specified separately, containing processor and hardware allocation, mapping of processes to processors and hardware blocks, and the definition of the communication topology and its parameters. While mapping the specification to the platform, the designer also specifies important software aspects, such as task mapping, the definition of task priorities, and selection of the scheduling policy for each processor.

Based on application and platform specification, our system compiler automatically maps the application down to a set of processors and busses, creating a set of tasks for each processor, and generating the communication drivers between processes depending on their HW/SW mapping. The application-specific hardware-dependent code is generated by the system compiler. As one output, it generates a system model at selectable abstraction level (with different amount of detail).

The abstract system model is valuable for virtual prototyping, early performance estimation, and validation of the feasibility of the HW/SW mapping. It also enables functional validation of the application over the given platform. Furthermore, it exposes the effects of dynamic scheduling for each processor, allowing optimization of priority mapping and guiding static load balancing. Altogether, the system model is a convenient virtual debugging platform that is usable before HW availability.

Most importantly, the system model serves as an input to the back-end SW generation, which generates and cross-compiles the C code. In particular, it generates the firmware, drivers and interrupt handlers, which implement the external communication of the processor. It also adjusts the application code to execute on top of the selected Real-Time Operating System (RTOS). Finally, the linker creates the final software binary for each processor. For early validation of those binaries, a system model with integrated Instruction Set Simulators (ISSs) can be used.

We informally distinguish between software synthesis and software generation. Both produce an implementation out of an abstract input model by adding implementation level detail. In contrast to generation, synthesis includes in addition an automatic optimization for a given objective or cost function. In our work described in this chapter, we describe a pure generation-based approach that does not include an optimization.

The rest of this chapter is organized as follows. We first discuss the context of software generation and survey current approaches. Then, Sect. 8.2 describes in detail the envisioned HdS development based on a platform-agnostic input and abstract system models. Section 8.3 provides an overview on SW generation and Sect. 8.4 focuses on the generation of HdS. Section 8.5 discusses application examples and demonstrates the approach for six real-life applications. Section 8.6 summarizes and concludes the chapter.

8.1.1 Context and Related Work

Designing a modern complex MPSoC is challenging both in terms of hardware and software. The current manufacturing capabilities offer tremendous integration capabilities and a high degree of implementation freedom. For optimization, a vast exploration space has to be explored and analyzed in the design process. At the same time, the market demands a shorter time-to-market to yield competitive products. Hence, the challenge is to design increasingly complex embedded systems in a shorter period of time.

System-level design is accepted as the main approach to address the complexity challenges. It uses a unified approach to design hardware and software concurrently. System-level design uses higher levels of abstraction to describe a system. Ideally, this allows to describe a system solely as a composition of algorithms, so that the designer can maintain the system overview, while not being burdened by the vast amount of implementation details.

To capture systems jointly with hardware and software, System Level Design Languages (SLDLs) have been developed, such as UML, graphical input, Esterel and C-based languages. In this chapter, we focus on C-based SLDLs. Examples of C-based SLDLs are SystemC [GLMS02], which is widely used in academia and industry, and SpecC [GZD+00]. These languages are based on C++ and ANSI-C, respectively, and have been extended to also capture system and hardware aspects, such as parallelism, pipelining, signals, and bit-vectors to just name a few added concepts.

Abstract models for system-level design are often described as Transaction Level Models (TLMs) [GLMS02], which abstract away the details of pins and wires [CG03]. By omitting implementation-level detail, TLMs execute dramatically faster than bit-accurate models. Therefore, they are widely used for design space exploration and early development.

Today, TLMs are typically written manually [HYL+06] and are moreover rarely used for generation of a complete final implementation. Specialized partial solutions are already very successful, e.g. for generating the interface description between RTL hardware and software (see Chap. 5). To increase productivity, we envision a design flow that spans from an abstract, untimed, and platform-agnostic specification down to an actual implementation on real hardware, as we will describe in this chapter.

Traditionally, SW generation has been addressed from very specific input models and with a limited target architecture support. Some examples are POLIS [BCG+97], DESCARTES [RPZM93], and Cortadella et al. [CKL+00]. The POLIS [BCG+97] approach uses a Co-design Finite State Machine (CFSM) model, where each FSM represents a component in the system. Software generation is performed by transforming the input model into an S-Graph, and subsequent C code generation. This work focuses on reactive systems and

is not designed for general applications. DESCARTES [RPZM93] uses a data flow description (Asynchronous Data Flow (ADF) and an extended Synchronous Data Flow (SDF)) as an input and supports heterogeneous systems. With the specific input choice, these solutions favor a particular application type. In contrast, a flexible generic C-programming model is desirable over these specific input models to cater to the needs of a broader programming audience and to capture a wider range of application domains.

Abstract models, based on SLDLs with a generic C-programming model, have been used for modeling software (SW) and its execution in abstract form [KKW+06, GYNJ01]. Additionally, ISSs have been integrated into abstract system models to create system co-simulation environments [BBB+05, CoWa]. Such, virtual platforms allow for a detailed analysis of the system before availability of real hardware, often revealing details not available on the target [HYL+06]. While these approaches focus on simulation and validation, they do not offer an integrated solution to generate the final implementation.

Some early approaches show solutions to use an abstract model, which contains the common description of HW and SW, as a source for generating the embedded software. Herrara et al. [HPSV03] describe SW generation from a SystemC model. With SystemC being a library extension of C++, they propose to overload SystemC library elements for execution on the target system. This has the advantage of reusing the same model for specification and target execution. However, the approach partly replicates the simulation engine.

Krause et al. [KBR05] generate source code from SystemC and adjust the application to execute on top of an RTOS. To flexibly target different RTOS vendors, they capture the API in an XML format for a customized generation. This approach, however, does not describe in detail the generation of communication and synchronization code and the creation of the final target binary.

Gauthier et al. [GYJ01] describe a method for generating application-specific operating systems and the corresponding application SW. Their work focuses on the OS portion and does not address external HW. Our solution, on the other hand, explicitly includes heterogeneous external HW. Yu et al. [YDG04] show generation of application C code from an SLDL, however without showing the final target binary. Our approach includes generation of communication drivers, multi-task adaptation, and the generation of the final binary image.

The Phantom Serializing Compiler [NG05] translates multi-tasking POSIX C code input into flat C code by grouping blocks to Atomic Execution Blocks and custom scheduling them. This approach is oriented toward a pure SW solution. In contrast, we address SW generation in a system context, specifically taking HdS and external communication into account.

8.2 Software-enabled System Design Flow

Electronic System Level (ESL) design addresses the complexity challenges of designing a modern embedded system. One such flow is outlined in Fig. 8.1 and uses a two step design approach. This ESL flow, implemented in [DGP+08], generates first a system TLM for detailed performance estimation and early MPSoC development. In a second step, the TLM is used as an input to automatically generate SW binaries for the processors in the target platform.

The input to the system design flow is the *specification model*. It describes the algorithms of the system and their dependencies. The specification model is captured in an untimed and platform-agnostic form using a C-based SLDL. For the experiments reported in this chapter, we use the SpecC SLDL [GZD+00]. The concepts shown, however, are equally applicable to other C-based SLDLs, such as SystemC, as well.

Important for a flexible and analyzable input specification is the separation of computation and communication. This separation enables automatic refinement of communication and mapping of computation to separate processing elements. The computation is grouped in behaviors (or modules / processes), and communication is expressed in channels. The upper portion of Fig. 8.2 shows a graphical representation of a simple system specification. The boxes with rounded corners symbolize behaviors. The actual C code inside the behaviors (e.g. *B2* and *B3*) is omitted for brevity.

The behaviors communicate via direct point-to-point channels. For an easier generation, these channels are selected from a feature-rich set of standardized channel types. They allow for a wide range of communication types, such as synchronous and asynchronous communication, blocking and non-blocking communication (e.g. FIFO), as well as for synchronization only (e.g. semaphore, mutex, barrier). Basically, these channels are similar to standard communication primitives offered by middleware or an operating system.

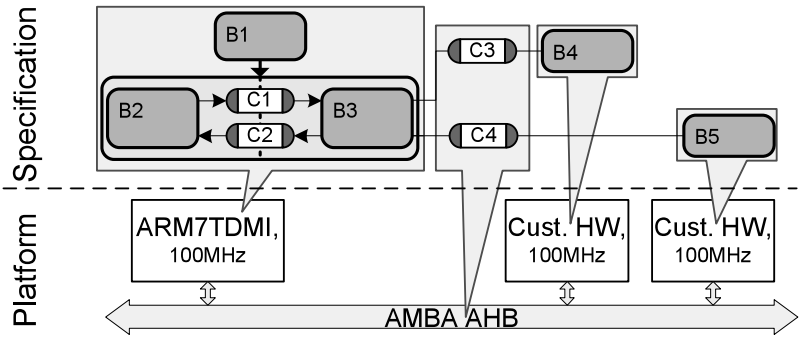

Figure 8.2. Example specification with architecture mapping.

Behaviors can be composed hierarchically to allow complex structures. They can be arranged to execute in any order, such as sequential, parallel, pipelined, or state machine controlled. In the example, behaviors *B2* and *B3* execute in parallel. They communicate through channels *C1* and *C2*. These channels are of type "double handshake", which implies blocking, synchronous communication that is not buffered. The channels *C3* and *C4*, for communication between *B3*, *B4* and *B5*, are finite depth FIFO channels. Using these standard channels allows for a very intuitive programming approach, that is independent of any hardware selection and application distribution.

A second input to our system design flow contains the architecture decisions which describe the platform, as visualized in the bottom portion of Fig. 8.2. The designer enters these decisions using an interactive Graphical User Interface (GUI).

Architecture decisions include the allocation of processing elements (PEs) (e.g. processors, HW components). In the example, an ARM7TMI processor and two custom hardware components are allocated. PE-specific parameters, such as clock frequency, are chosen during allocation. Additionally, the user defines the mapping of behaviors to PEs, deciding which PE will execute the computation inside each behavior. Behaviors, that are assigned to execute on a processor, are wrapped into tasks. The user can then define important task parameters, such as priority and stack size.

Besides dealing with the computation, the designer also controls the allocation and mapping of communication protocols. The example mapping decisions are illustrated in the bottom portion of Fig. 8.2. Here, a bus system of type AMBA AHB [AMBA] is allocated. The call-out boxes symbolize mapping the channels to that bus. For each channel, the user can also define essential communication parameters. For one, the user can select the synchronization scheme, such as polling or interrupt-based synchronization. Additionally, a bus address, that identifies the channel on the communication medium, can be selected.

Based on this these inputs, our system compiler [DGP+08] automatically generates a system TLM that reflects the architecture decisions. For this model refinement, components out of the component data base (compare Fig. 8.1) are instantiated and connected. The communication between processing elements is refined from the standardized abstract channels down to communication based on the selected medium (here the AMBA AHB). The TLM, see example in Fig. 8.4, allows for system exploration, performance analysis and debugging. The TLM simulates significantly faster than a traditional ISS-based model [SGD07].

Once the designer is satisfied with the performance and quality of the system, the same TLM serves then as input for the back-end HW synthesis and SW generation. The SW generation produces the final SW binaries that are

executable on a set of processors composing the platform. It generates the application code, and all drivers for communication in a heterogeneous system. The SW application executes on an off-the-shelf RTOS, or by using an interrupt-driven system for small applications.

8.3 Software Generation Overview

The SW generation, as shown in Fig. 8.3, uses the TLM as an input. As described before, the TLM reflects all architecture decisions. Computation is mapped to processing elements. Computation within each processor is grouped to tasks, all essential task parameters are captured, and the tasks are executed on top of an abstract RTOS (the concepts of RTOS modeling are also described in Chap. 9). The external communication has been refined according to an ISO/OSI layered approach. It is mapped to a set of busses and protocols using bus primitives. External synchronization is implemented (e.g. polling or interrupt) based on the designer's choice. Furthermore, the model contains all structural information to implement the communication decisions. Therefore, the input TLM contains all functional and structural information needed for the target implementation. Please see [DGP+08] for a more detailed description of the TLM generation.

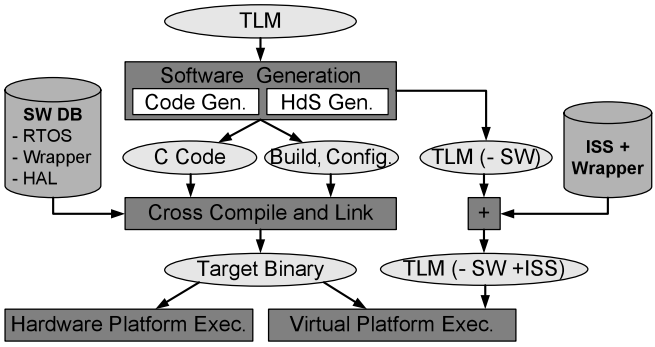

Figure 8.3. Software generation flow [SGD08].

Our *software generation* is divided into C code generation and HdS generation. The C code generation [YDG04], generates flat C code out of the hierarchical model captured in the SpecC SLDL. It converts behavior hierarchies into a set of C functions. Instance-specific variables are translated into a set of data structure instances. Additionally, the channel connectivity between behaviors is resolved into flat C code. In other words, the C code generation solves similar issues as early C++ to C compilers that translated a class hierarchy into flat C code.

The second portion, the HdS generation, generates code for processor internal and external communication, including drivers and synchronization (polling or interrupt). It also generates code to execute multiple tasks on the same processor. To create the complete binary SW image, it finally generates configuration and build files (e.g. Makefile) which select and configure database components. As such, a particular RTOS is chosen, properly adapted/ported to the selected processor. A hardware abstraction layer (HAL) is included based on the target platform, consisting of low-level drivers for the timer, the programmable interrupt controller (PIC), and the bus accesses.

Using a cross compiler, the final target binary (or binaries) is created, which can execute on the target processor(s), or alternatively on a virtual platform. A virtual platform allows validation and development of the final software binaries already before the availability of real hardware. To generate a virtual platform, our SW generation removes the model of the SW running on each processor from the TLM and replaces it with an ISS that is wrapped for integration into the system model. Each ISS instance then executes one SW binary.

8.4 Hardware-dependent Software Generation

The HdS generation uses the system TLM as an input (see example in Fig. 8.4), which was generated by the system compiler based on the designer's architecture decisions. Following the mapping definitions, illustrated in Fig. 8.2, the behaviors *B1*, *B2* and *B3* execute on the processor. The behaviors *B4* and *B5* are each mapped to an own HW accelerator. The TLM contains hierarchical behaviors, channels, and additional HW to properly reflect the platform characteristics. For example, it contains a model of a PIC that maps multiple external interrupts to the available CPU interrupts, and a timer module for periodic interrupts.

The HdS generation parses the input TLM into an abstract syntax tree and then operates on this tree for code generation. For explanation, we distinguish three generation aspects: communication generation, multi-task generation and generation of the final target image. The following sections describe each aspect individually.

8.4.1 Communication Generation

The communication generation deals with processor internal and external communication. In particular, it creates the driver code for communication between the software and external HW. It also generates code for synchronization, for which it inserts stubs into the application code, and generates interrupt handlers and/or polling code.

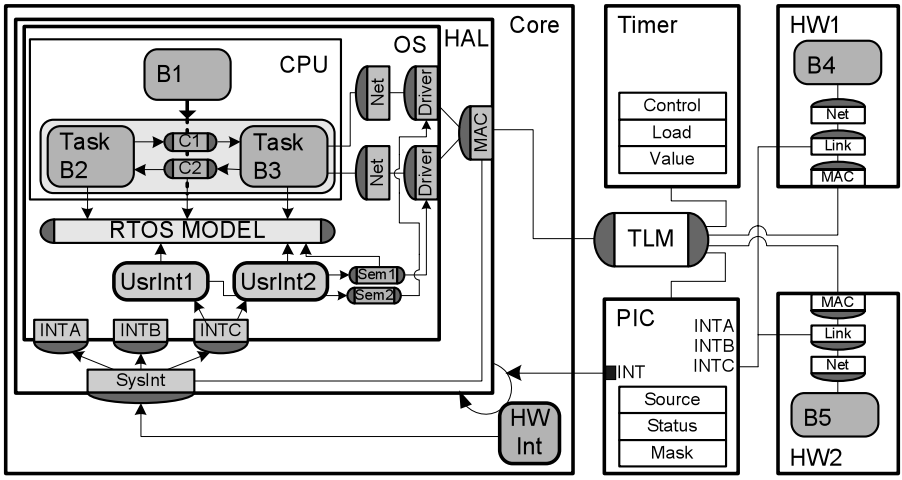

Figure 8.4. Processor and application TLM.

Internal Communication. Internal communication takes place between tasks on the same processor. In the example shown in Fig. 8.4, the channels *C1*, *C2*, *Sem1* and *Sem2* are used for internal communication. These are instances of our standard channels as also used in the specification. To provide the particular communication on the target system, the abstract standard channels are replaced with a target-specific implementation that uses the primitives of an underlying RTOS (or an emulation thereof, in case an RTOS is avoided). Note, that this implementation does not recreate the simulation environment on the target. Instead, a target-specific implementation is used that recreates the same interface and semantics as the abstract channels. For example, a blocking synchronous communication channel is implemented on an RTOS-based system with a semaphore, two events, and a *memcpy* using the services of our RTOS Abstraction Layer (RAL), which we insert for independence of the actual RTOS (for details, please refer to the later section about multi-task generation).

External Communication. To support heterogeneous systems, we follow the ISO/OSI layering model [ISO94] to implement external communication. Examples of external communication are the channels *C3* and *C4* of the initial specification (see Fig. 8.2). According to the mapping information, these channels capture communication between different processing elements (e.g. processor and custom hardware). These channels no longer appear directly in the system TLM in Fig. 8.4. Our system compiler has refined the abstract channels into stacks of half channels (namely *Net*, *Driver*, and *MAC*),

which are inserted into the processor model. A matching stack of half channels is inserted into each HW component (*HW1* and *HW2*) as well.

At the top of the stack, the typed user data is marshalled into a flat untyped data stream. This untyped stream provides a common representation that can be interpreted among different processing elements regardless of bitwidth, endianness and padding rules. This common representation for example allows that a little endian processor can read and interpret the data stream of a big endian processor.

The communication generation has access to the abstract syntax tree representing the application code. Therefore, it can extract the necessary type information from the application code and generate application-specific marshalling code that uses standard conversion functions to create the untyped data stream. For example, the user may define structure *tReq* that contains three elements *startTime*, *coeff1* and *base*, as shown in Listing 8.1.

Based on the information of the channel *Net* (see Fig. 8.4), the communication generation produces marshalling code that serializes the structure data into a flat byte stream as shown in Listing 8.2. Note that, in contrast to using fixed bitwidth types already in the specification, as discussed in Chap. 5 and Chap. 6, our system-level approach contains platform-agnostic types (e.g. plain *int*) in the initial specification model. The marshalling process here therefore is necessary in order to create the platform-specific types.

Data from the input structure (pointer *pD*) is converted into the buffer (pointer *This−>buf*). The marshalling code uses standard conversion functions for each basic data type (e.g. *uhonlong()*). Later in the generation process, a processor-optimized implementation of the marshaling function is selected from the database.

The next half channel, the *Driver*, contains information about the channel's system-wide addressing. It maps the end-to-end channel, which connects two behaviors, to a set of point-to-point links. In a platform with many busses, an end-to-end link may connect processing elements on different busses. Then, multiple point-to-point links create the connection across the busses, which are connected via communication elements (e.g. bridge or transducer). Note that, in comparison to the Chapter 5 and Chapter 6, our system-level approach

```
1 typedef struct stReq {
2   long           startTime;
3   short          coeff1;
4   unsigned short base;
5 } tReq;
```

Listing 8.1. User type definition in the specification model.

```
1  void c_pre_req_CPU_send(/* ...*/ *This, struct tReq *pD){
2    unsigned char *pB = This->buf;
3    htonlong(pB,   pD->startTime);
4    pB += 4;
5    htonshort(pB,  pD->coeff1);
6    pB += 2;
7    htonushort(pB, pD->base);
8    pB += 2;
9    c_link_CPU__CAN_CTRL_DLink_send(/*...*/This->buf, 8);
10 }
```

Listing 8.2. Generated code for marshalling of user data.

generates a custom register addressing here on-the-fly, based on an available system-wide view of the components and their address space.

The slave in our example is connected to the processor bus. Therefore, direct communication is possible and no additional communication elements are necessary. However, complex communication schemes spanning multiple bus hierarchies are possible. Then, user messages are packetized to minimize buffer requirements of intermediate communication partners. Depending on the information in the *Driver* channel, the corresponding source code is generated.

The driver also implements a channel-specific synchronization mechanism, which will be explained in the next section. Finally, the *Driver* transfers the data using the Media Access Control (MAC) layer, which implements the low-level access to the communication media. This layer provides services to transport an arbitrary sized contiguous block of bytes to an address in the system. According to the platform definition, the HdS generation selects later a processor-specific MAC implementation. In a simple case of a processor's primary bus, the MAC may use the processor's memory interface.

Synchronization. For a typical master/slave bus, external synchronization is required for a slave to indicate it being ready for a data transfer (e.g. required data being available). The designer choses the type of synchronization for each channel, selecting between polling or interrupt-based synchronization. Furthermore, the designer may choose to share interrupts between sources to reduce the overall number of interrupt pins. These choices are reflected in the generated system TLM.

If *polling* was chosen, polling code is generated as part of the driver code. An example is outlined in Listing 8.3. The CPU accesses the slave's polling flag to check whether the slave is ready for the communication. This access is performed using the MAC services analogous to the external communication (see the call to function *Ahb_masterMemRead()* in Line 5). If the slave is not

```
1  void c_link_CPU__HW_DLink_send(/* ... */ *This,
2    const void *pData, int len) {
3      unsigned char flag;
4      do { /* poll slave if ready */
5           Ahb_masterMemRead(/* ... */,
6           HW1_DLink_0_FLAG_ADDR, &flag, sizeof(flag));
7           if(flag) { /* break if ready */
8                break;
9           }
10          /* delay for poll. period */
11          TaskDelay(HW1_DLink_0_POLL_DELAY);
12     } while(1);
13     /* successfully synch'ed, transfer data now */
14     Ahb_masterMemWrite(/* ... */,
15          HW1_DLink_0_DATA_ADDR, pData, len);
16 }
```

Listing 8.3. Polling synchronization example.

ready, the polling code uses RTOS services to delay execution for the polling period (see function call *TaskDelay()* in Line 11), and repeats polling. Once determined that the slave is ready, the polling loop terminates (Line 8) and transfers the data (Line 14).

In case of *interrupt* synchronization, the TLM contains a model of the interrupt chain. In Fig. 8.4, for example, the chain consists of the *PIC*, the system interrupt handler *SysInt*, the application-specific interrupt handler *INTC*, the user interrupt handler *UsrInt1* and *UsrInt2*. Finally, semaphore channels (*Sem1*, *Sem2*) connect each interrupt handler with the driver code, so that the (short) interrupt handler can start the (long) driver to handle the communication. To implement interrupt-based synchronization, our HdS generation produces a chain of correlated code. The next paragraphs describe the interrupt-based synchronization code, following the event sequence when sending a message from *B5*, which is mapped to a hardware component, to *B2*, which is mapped to the processor. The event sequence is illustrated in Fig. 8.5.

At t_0, the behavior *B2* expects a message. With the message not being available, *B2* waits on the semaphore *Sem1* and yields execution to the next lower priority task *B3*. At t_1, behavior *B5*, that is mapped to *HW2*, reaches the code to send the expected message and signals via interrupt *INTC* the availability of the message to the processor core. On the way, the PIC sets the processor interrupt *Int*. This in turn triggers the interrupt chain on the processor, which we have labeled 1 through 4.

1. The low-level assembly interrupt handler preempts the currently running task *B3*. It stores the current context on the stack and then calls the sys-

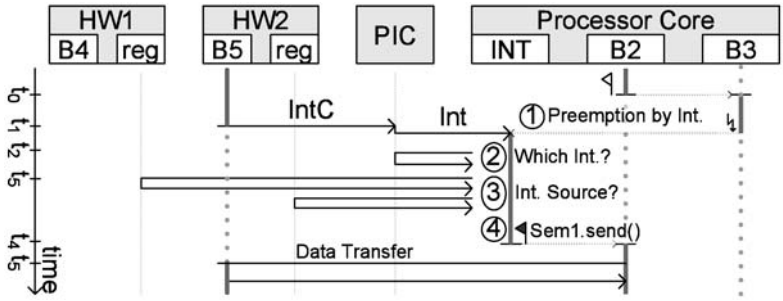

Figure 8.5. Events in external communication.

tem interrupt handler. The low-level assembly interrupt handler, which is part of the RTOS port is inserted from the software database.

2. The system interrupt handler (see half channel *SysInt* in Fig. 8.4) communicates with the PIC. It determines through memory mapped I/O the highest priority pending interrupt. It then invokes the application-specific interrupt handler (see half channel *INTC* in the TLM in Fig. 8.4). The *SysInt* code is one element of the Hardware Abstraction Layer (HAL) stored in the database.

3. Since the interrupt in this example is shared between *HW1* and *HW2*, the actual source of the interrupt is determined next. The application-specific interrupt handler *INTC* determines the source of the interrupt by reading the status registers in *HW1* and *HW2*. Subsequently, *INTC* then calls the corresponding User Interrupt Handler (in this case *UsrInt2* of Fig. 8.4).

4. Finally, *UsrInt2* calls the semaphore *Sem1* to release the driver code that executes in the behavior *B2*. The semaphore channel uses the earlier described internal communication services.

After releasing semaphore *Sem1*, the interrupt handler terminates. Subsequently, the task for *B2* becomes ready and is scheduled. Finally, after the context switch, *B2* reads the data from *HW2*.

For HdS generation, we implement this chain on the processor. The code falls into two distinct portions. The first part is application-independent, and therefore can be stored in the software database. The second portion is application-specific and has to be generated out of the system TLM. The code for steps 1 and 2 belongs to the first portion that is application-independent, and their code is taken from the database. The code for steps 3 and 4, on the other hand, is application-specific, and is generated (step 3 based on *INTC*, and step 4 based on *UsrInt2*).

```
1  void ARM7TDMI_INTC_body(/* ... */ *This) {
2      unsigned char flag;
3      Ahb_masterMemRead(/* ... */,
4          HW1_DLink_0_FLAG_ADDR, &flag, sizeof(flag));
5      if(flag) {
6          c_os_semaphore_release(/* ... */ This->sem1);
7      }
8      Ahb_masterMemRead(/* ... */,
9          HW2_DLink_1_FLAG_ADDR, &flag, sizeof(flag));
10     if(flag) {
11         c_os_semaphore_release(/* ... */ This->sem2);
12     }
13 }
14
15 void ARM7TDMI_OS_CPU_main(/* ... */ *This){
16     /* ... */
17     c_os_semaphore__init(/* ... */ This->sem1);
18     c_os_semaphore__init(/* ... */ This->sem2);
19     BSP_UserIrqRegister(INTNR_int1handler,
20         ARM7TDMI_INTC_body, /* ... */);
21     /* ... */
22 }
```

Listing 8.4. Interrupt handler outline for shared interrupt.

Listing 8.4 outlines the generated code for an application specific interrupt handler (as described for step 3) that is shared between two interrupt sources. The handler sequentially checks the interrupt sources using the MAC communication services (e.g. Line 3). Once the handler finds the interrupt initiating hardware, it releases the associated user task that executes the driver code (see call to *c_os_semaphore_release()* in Line 6).

In addition, startup code is necessary to setup the interrupt chain on the processor side. For one, the application-specific interrupt handler needs to be registered to the system interrupt handler, so that it executes upon receiving of the associated interrupt. In this example, our HdS generator produces startup code that registers application-specific interrupt handler *INTC* to the system interrupt handler for execution upon receiving *INTC* on the *PIC* (see Listing 8.4, Line 19). To gather the necessary information, it traverses the connectivity and architectural information stored in the TLM. It also generates code to instantiate the semaphore channel and inserts appropriate calls into the driver code.

8.4.2 Multi-Task Generation

When multiple tasks are mapped to the same processor, they have to be dynamically scheduled to alternate their execution. Our *multi-task genera-*

tion produces code that uses an underlying multi-task engine in order to control tasks and schedule them. We support two different approaches for multi-tasking. First, we mainly focus on a traditional execution on top of an off-the-shelf RTOS. Furthermore, we provide an alternative of interrupt-based multi-tasking that can execute on a "naked" processor without any operating system.

RTOS-based Multi-Tasking Our main focus rests on targeting an off-the-shelf RTOS. This ensures using a reliable, well-tested operating system that offers great flexibility and often comes with significant tool support from the RTOS vendor. Operating systems are available in a wide range and focus. Often, they are highly configurable to tailor the OS to the application needs. By configuration, the memory footprint can be minimized to fit the needs of the embedded system under design.

Our *multi-task generation* makes use of a canonical OS interface, which we call the RTOS Abstraction Layer (RAL), see Fig. 8.6 (left). The very thin RAL (few hundred lines of (mostly inlined) code), abstracts from the particular OS's function names and parameters. We have chosen the RAL approach to limit the interdependency between our generation and the actual target RTOS. To ensure a generic API, we investigated different RTOS APIs (uCOS-II, vxWorks, eCos, ITRON, POSIX) and chose common primitives for task scheduling, communication and synchronization.

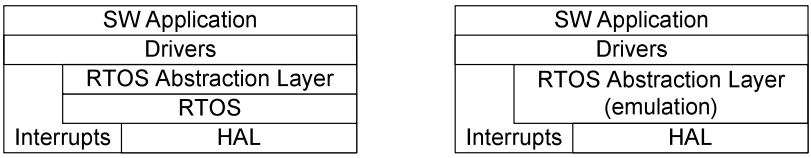

Figure 8.6. Software stack RTOS-based (*left*), interrupt-based (*right*).

Although the investigated RTOS APIs provided all necessary interfaces, this may not be the case for other RTOS APIs. In such cases, the RAL implements an emulation of the required functions that is constructed out of the available primitives. This approach guarantees that always an identical API, the RAL, is available to the generated SW generation, regardless of the particular RTOS implementation.

The input TLM contains mapping of behaviors to tasks (*Task B2*, *Task B3*) and their scheduling parameters. For RTOS-based multi-tasking, our HdS generation extracts the task control information from the TLM and generates task creation calls to the RAL. It also initializes the task's parameter set of the TLM (e.g. priority, stack size) on the target. From SLDL statements, which describe parallel execution of behaviors, our HdS generation produces code that calls

the RAL for task creation and release, and furthermore inserts code to join the multiple threads of execution after their completion.

To give an example, Listing 8.5 shows a partial specification following the system definition already shown in Fig. 8.2. It instantiates the three behaviors; *B1*, *B2* and *B3*. It executes first *B1* (Line 8) followed by a parallel execution of *B2* and *B3* (Lines 9 through 12).

Listing 8.6 outlines the generated C-code. The sequentially executing *B1* is directly called in the parent's main function (see call *TB1_main()* in Line 5). The parallel executing behaviors *B2* and *B3* are spawned using the RAL API function *TaskCreate()* (see Line 6 and Line 7). Note that *TaskCreate()* both creates a task and releases it for immediate execution. After spawning the tasks, the parent task waits until the created tasks have terminated (Lines 9 and 10).

In addition to the task control, processor internal communication is translated to RTOS-based communication. For that, the standardized communication channels (as described for the input) are implemented on top of the RAL. Our *multi-task generation* instantiates the target implementation and connects the channels according to the TLM connectivity information.

Interrupt-based Multi-Tasking In the second case, targeting a "naked" processor, concurrent software execution is performed without any RTOS. Instead, interrupts are utilized to provide multiple flows of execution. We support this alternative for systems where RTOS execution is not desirable. This may be the case, when the system consist of only very few tasks, the code is targeted to execute on a DSP, or when strict memory footprint limitations rule out utilizing an RTOS. We describe a motivating example for an interrupt-based

```
1  behavior B0(/*...*/) {
2      /* ... */
3      TB1  B1(/*...*/); /* instantiate behavior B1 */
4      TB2  B2(/*...*/); /* instantiate behavior B2 */
5      TB3  B3(/*...*/); /* instantiate behavior B3 */
6
7      void main(void) {
8          B1.main();
9          par {
10             B2.main();
11             B3.main();
12         }
13     }
14 };
```

Listing 8.5. Specification of behaviors.

```
void TB0_main(/* ... */){
    os_task_handle B2_thdl;
    os_task_handle B3_thdl;
    /* ... */
    TB1_main(/* ... */);
    B2_thdl = TaskCreate(TB2_main, /* ... */);
    B3_thdl = TaskCreate(TB3_main, /* ... */);

    TaskJoin(B2_thdl);
    TaskJoin(B3_thdl);
}
```

Listing 8.6. Generated RTOS-based multi-tasking code outline.

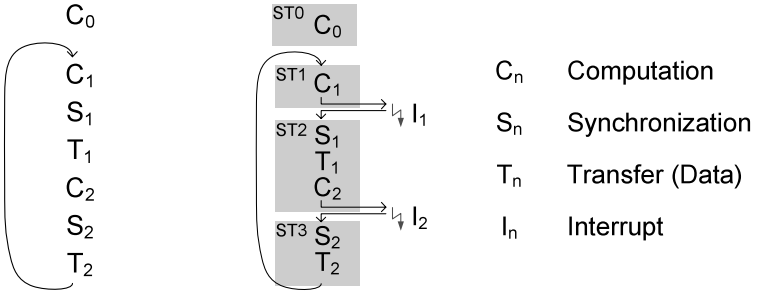

Figure 8.7. Reactive task template input (*left*) and output (*right*).

solution in Sect. 8.5. This case implements a GSM speech codec on a DSP with only two reactive tasks.

For our interrupt-based multi-tasking alternative, the RAL (see Fig. 8.6 (right)) implements a (very thin) RTOS emulation. It provides a subset of the RTOS services needed for software execution (e.g. events, processor suspension, and interrupt registration). To give an intuitive explanation, the *multi-task generation* converts the lowest priority task to execute in the processor main function, and all other tasks are converted to execute in a state machine fashion, in the context of their interrupt handlers.

More formally, we assume that each task is composed of a sequence of computation (C), synchronization (S), and data transfers (T). Figure 8.7 (left) shows an example sequence for one task. As described before, the driver code for communicating with external hardware contains both synchronization and communication. If only interrupts are used for synchronization, then the task main function can be transformed into a state machine, as shown in Fig. 8.7 (right).

In the state machine, each synchronization point starts a new state. For example, state *ST2* was created due to synchronization point S_1, and *ST3* due

```
1  void intHandler_I1() {
2      release(S1);       /* set S1 ready */
3      executeTask0();    /* task state machine */
4  }
5  void executeTask0() {
6    do {
7      switch(State) {
8                  /* ... */
9              case ST1: C1(...);
10                 State = ST2;
11             case ST2: if(attempt(S1)) {
12                   T1_receive(...);
13                 } else {
14                   break;
15                 }
16                 C2(...);
17                 State = ST3;
18             case ST3: /* ... */
19                 State = ST1;
20     }
21   } while (State == ST1);
22 }
```

Listing 8.7. Interrupt-based multi-tasking excerpt.

to S_2. The state machine transitions to the next state upon successful synchronization. For example, upon receiving of interrupt I_1, the state machine would transition from *ST1* to *ST2*. Additional states are inserted to implement conditional execution and loops. For example, the separation between the states *ST0* and *ST1* has been introduced to accommodate the one-time execution of the initialization code in C_0.

The created task's state machine is then executed in the interrupt handlers, which were initially chosen for synchronization of that task (in this example, the handlers of I_1 and I_2). In order to preserve the task priorities, the interrupts have to be chosen accordingly. A higher priority task has to exclusively use higher priority interrupts than a lower priority task. Consequently, the lowest priority task executes in the main task (T_{main}), the startup task of the processor.

Each local variable of a task's main function is integrated into a global data structure. Hence, the task execution no longer relies on an own stack, and may be executed in separate calls to the task's state machine.

Listing 8.7 outlines the generated C implementation. Please assume for explanation that the task's state machine is currently executing in the interrupt handler for I_1, *ST1* is the current state, and that computation C_1 has just finished. Next, the synchronization S_1 is checked (line 11). In case the synchronization has not yet occurred, the state machine terminates (line 14). Conse-

quently, the do-while loop, the function *executeTask0*, as well as the interrupt handler, all terminate. Thus, the processor can then serve a lower priority interrupt, or the main function.

Upon receiving the next interrupt I_1, the system interrupt handler calls the registered user interrupt handler *intHandler_I1* (see line 1). In line 2, the handler signals that S_1 is ready and then calls the state machine again (line 3). The current state is *ST2*, therefore the condition in line 11 is tested again. It now passes, since the synchronization has occurred, receives the data (line 12), and subsequently executes the computation C_2 in line 16.

The switch-case statement (lines 7 to 20) is surrounded by a do-while-loop, which is required to implement loops between states. In this example, the loop is necessary to transition from state *ST3* back to *ST1* without terminating the interrupt handler.

8.4.3 Binary Image Generation

The final aspect of HdS generation is the generation of a complete target binary. Our generation uses a cross-compiler tool chain (gcc) that is specific to the target processor and binary format. It generates configuration and makefiles for the binary image creation, which select components from the software database, configure these components, and in addition control the compilation and linking of generated code. This process is illustrated in Fig. 8.8.

An important aspect for establishing a flexible generation flow, with a wide variety of configurations with many processor and hardware combinations, is an effective design of the database. It is essential to identify the dependencies

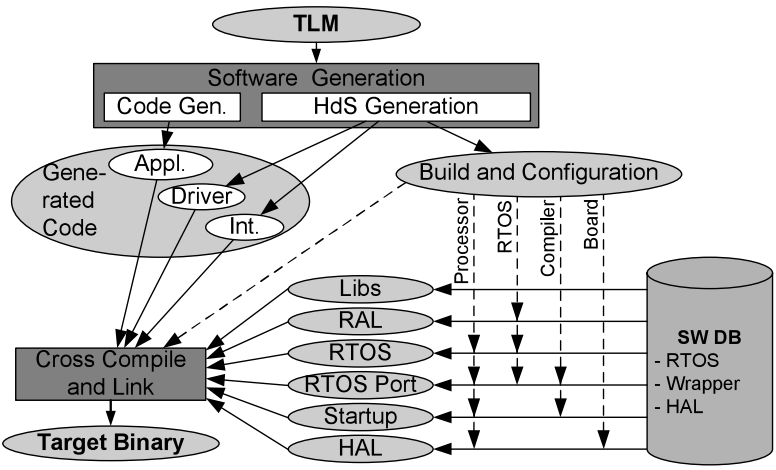

Figure 8.8. Generation of target binary.

of each database component with respect to the selected hardware/software configuration, e.g. the selected processor, RTOS, cross compiler, and board components. Capturing all dependencies is necessary for correctly selecting a component. On the other hand, overly specializing a component would lead to code duplication within the database, and yield a code bloat.

The matrix of arrows in Fig. 8.8 symbolizes the dependencies when selecting a component. Usually the most specific element is the RTOS port, since it depends on the RTOS type, the processor, and the cross-compiler (for example, for the call frame layout and the stack layout needed for the task creation). Our software generation also produces a customized Makefile, which selects the components according to the architecture information in the TLM, and then uses the cross-compiler to generate the target binary. Automating this step has the advantage, that the TLM serves as the sole input to the binary generation, avoids duplication of the system configuration (i.e. in the Makefile), and further minimizes the user effort.

8.5 Experimental Results

In this section, we describe some practical applications of your approach. We have applied it to a set of real-life examples. Two examples are covered in more detail. The first is a telecommunication example, the second uses an application from the automotive domain. Following that, we describe our generation results for several applications to more quantitatively compare the results.

8.5.1 Interrupt-based Implementation Example

We start by showing a specific example of an interrupt-based multi-tasking implementation. We implemented a GSM 06.60 [ETSI96] encoder and decoder on a Motorola DSP 56600 platform. As shown in Fig. 8.9, the DSP is assisted by a HW accelerator and four HW blocks that deal with input and output. The HW accelerator is dedicated to the computation-intensive codebook search of the encoding process.

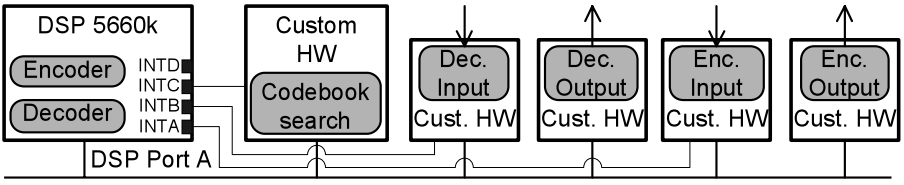

Figure 8.9. Media example of GSM transcoding.

In our application, the DSP only executes two reactive tasks (encoding and decoding). Also, an RTOS port for this particular DSP was not easily available. Therefore, we applied our interrupt-based multi-tasking approach to this example. Following a shortest-job-first scheduling policy, the longer executing encoder is assigned the lower priority of the two tasks. Hence, the encoder will execute in T_{main}. The higher priority (shorter) decoder task is transformed into a state machine. According to the architecture decisions, the decoder uses *IntB* for synchronization. Hence, the generated decoder's state machine will execute in the interrupt handler of *IntB*.

Figure 8.10 shows the state machine for the decoder task, which consists of 4 states. The states *ST1* and *ST2* have been created due to synchronization (S_1, S_2). The interrupt *IntB* is used for both synchronization points. A GSM speech frame consists of four sub-frames. Accordingly, *ST2* is repeated four times. The states *ST0* and *ST3*, respectively, are inserted to accommodate initialization, which executes only at the beginning, and post processing, which executes once per frame.

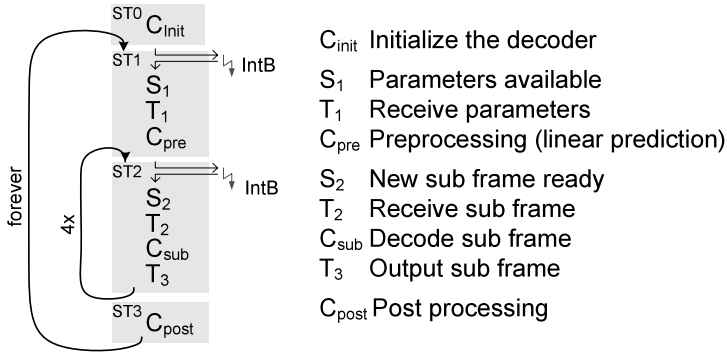

Figure 8.10. State machine for GSM decoder.

The input data is read by T_1 and T_2, which receive the initial parameters and the compressed sub-frame data, respectively. The decoded speech samples are transferred by T_3 without any additional synchronization into the output HW block. This particular transfer is performed without a preceding synchronization, since the receiving I/O HW is always ready.

Figure 8.11 shows the time line for transcoding one sub-frame after the initialization has already passed. The processor is suspended at the start of the time-line and waits for input data. At t_1, *IntA* signals availability of input data, and the registered interrupt handler is executed. The handler triggers event *e1* which the main task, T_{main} is waiting on. Hence, after termination of the interrupt handler T_{main} is resumed. After some processing, the encoder feeds

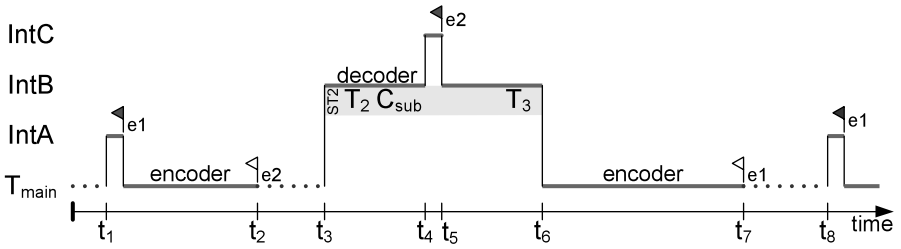

Figure 8.11. GSM transcoding execution.

the codebook accelerator. The encoder then suspends on event *e2* waiting for results from the accelerator. Again, the processor is suspended.

Later at t_3, *IntB* signals the availability of sub-frame data for decoding. The decoder state machine, which currently is in state *ST2*, is executed in the *IntB* handler. It reads the input data (T_2), decodes the sub-frame (C_{sub}), and transfers in T_3 the decoded speech samples to the output HW. Again, the latter needs no synchronization, since the output HW in the architecture is always available. At t_4, while decoding (in C_{sub}), the decoder is preempted by the higher priority *IntC*, which announces that the codebook search has finished. Subsequently, the interrupt handler releases the event *e2*. After the decoder interrupt handler has finished, the encoder resumes at t_6 and finishes at t_7. The same cycle repeats at t_8 with the next sub-frame. Throughout the execution of our testbench, 3451 interrupts are triggered. More results are later available in Table 8.2.

8.5.2 Exploration Example

We use an automotive example to illustrate the exploration capabilities with respect to comparing the two multi-tasking approaches. We model an Electronic Control Unit (ECU) containing an ARM7TDMI processor [ARM7]. The processor executes three tasks; anti-lock break control, RPM computation, and engine fan controller. Six sensors and actuators are connected to the ECU via two CAN busses (Fig. 8.12). Three further sensors are integrated in the ECU and are attached directly to the processor bus.

We have generated code for both approaches, first toward execution on top of the RTOS μCOS-II [Lab02], and second for interrupt-based execution. μCOS-II is a small, highly configurable RTOS that is mostly implemented in ANSI C. Ports of this RTOS are available for a wide range of processors, which dramatically simplified the integration.

Table 8.1 compares the generated RTOS-based and interrupt-based multi-tasking implementations. For the latter case, we mapped the lowest priority task, the fan control, to T_{main}, while the other two tasks were converted to state machines for execution in interrupt handlers.

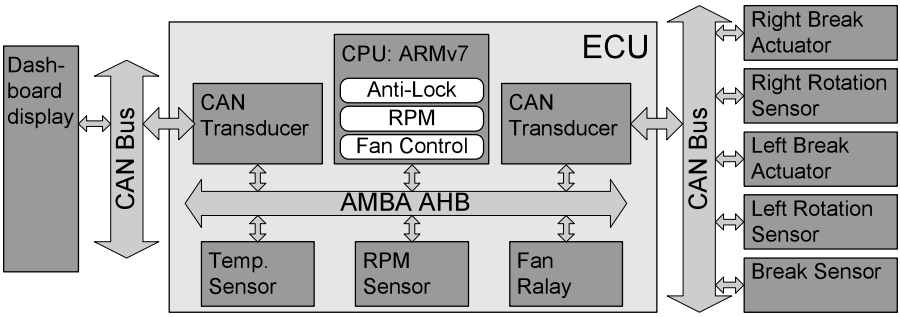

Figure 8.12. Automotive example application.

Multi-tasking	RTOS-based	Interrupt-based
Footprint	36224 Bytes	21052 Bytes
Alloc. Stacks	4096 Bytes	1024 Bytes
CPU Busy Cycles	6.706 MCycles	5.106 MCycles
# Interrupts	1478	1027

Table 8.1. Automotive example results.

As the results in Table 8.1 show, the automotive example profits from the interrupt-based solution. Avoiding the RTOS code yields a smaller memory footprint, since a simpler, more specific code is used instead. The footprint reflects the size of the ROM-able image and includes data, text and BSS segment. Neither solution uses dynamic memory allocation.

The interrupt-based multi-tasking results also in a smaller stack size, since all tasks share the same stack. Additionally, the interrupt-based solution shows a lower CPU consumption. The CPU busy cycles drop from 6.7 MCycles to 5.1 MCycles. This drop is due to the simpler implementation. The RTOS startup is avoided and fewer cycles are needed for the OS functionality (e.g. for event handling and context switching) due to simplicity.

To give an inside view of the system's performance, we analyze the interrupt latency. For the purpose of our measurements, we focus on the delay from the RPM sensor triggering the interrupt wire (to the PIC) to the first bus transaction appearing on the bus to read the RPM sensor.

In the interrupt-based approach, the latency until reading the RPM sensor is shorter (1001 cycles instead of 1794 cycles). This significant reduction is due to the execution in the interrupt handler itself. To compare, in the RTOS-based solution, the sensor is read in the task context, which results in an additional event communication and a context switch.

Also, we counted the number of occurring interrupts, which drops from 1478 to 1027. The interrupt-based solution does not use the timer for keeping

the system time, which explains the lower number of interrupts. On the other hand, the number of interrupts for data synchronization remains constant in both solutions.

Our automotive example clearly shows the benefits of the interrupt-based execution. We position it, where applicable, as an effective alternative in special cases (very few tasks, strict optimization requirements, or unavailability of an RTOS). Since either implementation can be generated automatically, a comparative exploration becomes easily possible.

8.5.3 Generation Results

To show the benefits of an automatic HdS generation, we have applied our HdS generation to a range of six target applications. The first two applications are the already described GSM transcoder and the car ECU. In addition, we examined a JPEG encoder, an MP3 decoder implemented in software, an MP3 decoder with 3 hardware accelerators, and a combined system with MP3 decoding and JPEG encoding.

Table 8.2 summarizes our generation results. The top section quantifies each target applications' complexity. It ranges from the simple JPEG with 2 I/O blocks to the combined application *Mp3 HW + JPEG*, which uses 6 I/O blocks, 3 HW accelerators, and 4 busses.

Example	GSM	Car	JPEG	Mp3 SW	Mp3 HW	Mp3 HW + JPEG
Complexity						
IO/ HW/ Bus	4/ 1/ 1	9/ 2/ 3	2/ 0 /1	2/ 0/ 1	2/ 3/ 4	6/ 3/ 4
SW Behaviors	112	10	34	55	54	90
Channels	18	23	11	10	26	47
Tasks/ ISRs	2/ 3	3/ 5	1/ 2	1/ 3	1/ 8	3/ 14
Lines of Code, RTOS-based						
Application	–	153	818	13914	12548	13480
HdS	–	649	210	299	763	1186
Lines of Code, Interrupt-based						
Application	5921	210	797	13558	12218	–
HdS	377	575	187	256	660	–
Execution, RTOS-based						
CPU Cycles	–	6.7 M	127.7 M	185.8 M	44.5 M	174.6 M
CPU Load	–	0.9%	100.0%	100.0%	30.9%	86.6%
Interrupts	–	1478	805	4195	1144	1914
Execution, Interrupt-based						
CPU Cycles	42.0 M	5.1 M	126.7 M	182.3 M	43.3 M	–
CPU Load	42.5%	0.7%	100.0%	100.0%	30.5%	–
Interrupts	3451	1027	726	4078	1054	–

Table 8.2. SW generation and execution results.

Next, the table shows the number of generated lines of code for application and HdS, each for the RTOS-based and the interrupt-based multi-tasking. As described earlier, we have not implemented the GSM in an RTOS-based solution, since we had no RTOS port available for the DSP. Also, we have not realized the *Mp3 HW + JPEG* example in the interrupt-based form, since it uses services we do not intend to replicate with interrupts. In the examples with HW acceleration, the HdS code is larger due to the extra effort in communication. Overall, a significant amount of code is generated (e.g. 1186 lines for *Mp3 HW + JPEG*).

Automatically generating the software binaries yields a significant gain in productivity. In all examples, our HdS generation completes in less than a second. On the other hand, manually writing the HdS would take days. Thus, the code generation in our approach has a significant impact on reducing the overall design time of embedded systems with HdS context.

To validate the correctness of the generated code, we executed each synthesized target binary on a virtual platform. For that, we integrated a Motorola proprietary instruction set simulator (ISS) for the DSP, and the SWARM ISS [Dal00] for the ARM7TMDI.

Each application executes functionally correct, yielding an output matching the specification. Table 8.2 shows the execution statistics of the ISS cosimulation. As in the car example, fewer CPU cycles (busy cycles only) are consumed in the interrupt-based solution. However, with an increasing computation complexity, the relative improvement becomes marginal. Similar to before, avoiding the OS timer tick reduces the number of processed interrupts.

8.6 Conclusions

Embedded software generation is an essential aspect of implementing todays SoC. It avoids the tedious and error prone manual implementation. In this chapter, we have presented a systematic approach for generating the final target binaries from an abstract specification model. We have shown software generation as an integral part of an ESL flow. Beginning from an abstract model containing the application specification, our flow automatically generates a system TLM based on the designer's architecture decisions. From the generated TLM, the software generation then automatically generates the binaries for each processor in the system. Together, this completes the ESL flow for the software, offering a seamless solution from an abstract system model down to an implementation on embedded processors.

The presented HdS generation addresses three parts: communication generation, multi-task generation, and binary image generation. It generates communication drivers, interrupt handlers, and adjusts for the target multi-tasking. Our approach supports targeting toward an existing RTOS. Furthermore, it of-

fers an alternative to use interrupts for multi-tasking if an RTOS-based execution is undesirable.

We have demonstrated automatic generation using six real-life target applications: different media applications and a control system. The ESL flow with integrated software generation addresses a wide range of target processors, platforms and applications.

Automating the tedious and error-prone process of manual firmware development results in significant gains in productivity. Not only is the automatic generation much faster than a manual implementation, it also allows the designer to focus on the essential algorithms, without the burden of implementation details. Further, with the automatic generation, alternative solutions can be quickly and easily generated. This allows for a rapid exploration of the embedded software design space, e.g. when investigating alternative mapping solutions.

Acknowledgments

The authors thank the SCE research team at the Center for Embedded Computer Systems at UC Irvine for their technical support. The authors also thank the editors and reviewers of this book for their valuable feedback in improving this chapter.

References

[AMBA] Advanced RISC Machines Ltd (ARM). AMBA Specification (Rev. 2.0), ARM IHI 0011A.

[ARM7] Advanced RISC Machines Ltd. (ARM). ARM7TDMI (Rev. 4) Technical Reference Manual, 2001.

[BBB+05] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Oliver. MPARM: exploring the multi-processor SoC design space with SystemC, *VLSI Signal Process.*, 41:169–182, 2005.

[BCG+97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic, Dordrecht, 1997.

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Newport Beach, CA, October 2003.

[CKL+00] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Marc Massot, Sandra Moral, Claudio Passerone, Yosinori Watanabe, and Alberto Sangiovanni-Vincentelli. Task generation and compile time scheduling for mixed data-control embedded software. In *Proceedings of the Design Automation Conference (DAC)*, Los Angeles, CA, June 2000.

[CoWa] CoWare. Virtual Platform Designer. www.coware.com.

[Dal00] Michael Dales. *SWARM 0.44 Documentation*. Department of Computer Science, University of Glasgow, November 2000. www.cl.cam.ac.uk/~mwd24/phd/swarm.html.

[DGP+08] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel Gajski. System-on-Chip Environment: A SpecC-based framework for heterogeneous MPSoC design. *EURASIP J. Embed. Syst.*, 2008.

[ETSI96] European Telecommunication Standards Institute (ETSI). *Digital cellular telecommunications system; Enhanced Full Rate (EFR) speech transcoding*, 1996. GSM 06.60.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic, Dordrecht, 2002.

[GYJ01] Lovic Gauthier, Sungjoo Yo, and Ahmed A. Jerraya. Automatic generation and targeting of application-specific operating systems and embedded systems software. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 20(11), November 2001.

[GYNJ01] Patrice Gerin, Sungjoo Yoo, Gabriela Nicolescu, and Ahmed A. Jerraya. Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2001.

[GZD+00] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic, Dordrecht, 2000.

[HPSV03] F. Herrera, H. Posadas, P. Sánchez, and E. Villar. Systematic embedded software generation from SystemC. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2003.

[HYL+06] Sungpack Hong, Sungjoo Yoo, Sheayun Lee, Sangwoo Lee, Hye-Jeong Nam, Bum-Seok Yoo, Jaehyung Hwang, Donghyun Song, Janghwan Kim, Jeongeun Kim, HoonSang Jin, Kyu-Myung Choi, Jeong-Taek Kong, and SooKwan Eo. Creation and utilization of

a virtual platform for embedded software optimization: an industrial case study. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Seoul, South Korea, October 2006.

[ISO94] International Organization for Standardization (ISO). *Reference Model of Open System Interconnection (OSI)*, second edition, 1994. ISO/IEC 7498 Standard.

[KBR05] Matthias Krause, Oliver Bringmann, and Wolfgang Rosenstiel. Target Software generation: An approach for automatic mapping of SystemC specifications onto real-time operating systems. *Des. Autom. Embed. Syst.*, 10(4):229–251, 2005.

[KKW+06] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW Performance estimation framework for early system-level-design using fine-grained instrumentation. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2006.

[Lab02] Jean J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, Gilroy, 2002.

[NG05] Andre Nacul and Tony Givargis. Lightweight multitasking support for embedded systems using the phantom serializing compiler. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2005.

[RPZM93] Sebastian Ritz, Matthias Pankert, Vojin Zivojnvic, and Heinrich Meyr. High-level software synthesis for the design of communication systems. *IEEE J. Select. Areas Commun.*, April 1993.

[SGD07] Gunar Schirner, Andreas Gerstlauer, and Rainer Dömer. Multi-faceted modeling of embedded processors for system level design, Abstract. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2007.

[SGD08] Gunar Schirner, Andreas Gerstlauer, and Rainer Dömer. Automatic generation of hardware dependent software for MPSoCs from abstract system specifications. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seoul, Korea, January 2008.

[YDG04] Haobo Yu, Rainer Dömer, and Daniel Gajski. Embedded software generation from system level design languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, Yokohama, Japan, January 2004.