Chapter 5

HW/SW INTERFACE

Implementation and Modeling

Wolfgang Ecker, Volkan Esen, Thomas Steininger and Michael Velten

Abstract This chapter addresses HW/SW interface implementation and modeling. As introduction, basic concepts regarding HW/SW interfaces on both HW and SW side are presented in detail. The focus is on several aspects of register and bit field read/write access, address mismatch, synchronization, and data alignment. The HW micro-architecture is outlined in block diagrams, the SW code is listed in C-code snippets. As new contributions, data flow abstraction for HW/SW models and consistently derived RTL models, TLM models, and C code by using a template approach are presented.

Keywords: Address Offset, Base Address, Bit Field, C, Low/High Level Driver, Endianness, Interrupt, Register, SystemC, Template, Volatile, XML

5.1 Introduction

HW/SW interfaces at the lowest level deal primarily with the transfer of data from one storage element to another. These storage elements are either registers in the CPU, registers in the HW peripheral to be accessed, or cells in a memory array. By execution of data move operations—primarily by a processor but also by specific blocks as DMA (direct memory access) units—the data is transferred from one address to another. These data move operations are either coded as assembler instructions or compiled from a higher level programming language. This seemingly simple mechanism gains high complexity in today's embedded systems:

- Thousands, even tens of thousands of registers can be found in complex SoCs. Flat or hierarchical bus systems, potentially each with an own protocol, are used to access all these registers from one or more CPUs. Alignment of data is only one issue that has to be considered in this context.
- Data transfer can be initiated by polling. Alternatively, an interrupt can be executed by a CPU or a DMA (direct memory access) device. It requires careful implementation to avoid side effects and to fulfill all time constraints of all registers to be accessed.
- The number of different kinds of registers invented by designers is almost unlimited. The simple read/write of a cell can be associated with a variety of additional functionality that either constrains the read/write access or causes side effects. The driving force here is either the limitation of available addresses or performance optimization of the HW/SW interface. To give two examples: A register is cleared after it has been read in order to show that data has been consumed already or a trigger impulse is generated and passed to the hardware core of a component in order to request the execution of some algorithm.

At the software side also a simple mechanism is executed: Data is moved from one object or address to another object or address. Complexity arises here from the wide range of interpretation of the values, the effects hidden behind these data transfers, and the interaction of the data transfer with the control flow of the rest of the software. Especially the interrupt signaling mechanism shall be mentioned here, which is another HW/SW interface to signal some request from a hardware component to the software.

In order to cope with the complexities of the HW/SW interface, formal models and specifications have been developed uniquely describing the structure and semantics of the interface. Based on the formal description, parts of the hardware side of the interface and the software side of the interface can be generated.

In this chapter, we discuss first implementation issues of the HW/SW interface from the ground up. A simplified serial interface peripheral device, in the following referred to as SIF, is used throughout the chapter as an example for the various alternatives and options of the HW/SW interface. Though being simplified, this serial interface device contains all important use cases related to a general industrial HW/SW interface. The serial interface is continuously extended and the final version is described in a data sheet like—data book oriented—style (see Sect. 5.5).

Based on the serial interface device, various aspects of the HW/SW interfaces are discussed. They include reading and writing complete data words to registers, access to single bits, synchronization between HW and SW, and register address mismatch.

Finally, modeling aspects including models and meta models are discussed. An outline of further aspects concludes the chapter.

5.2 Reading and Writing Data Words

As a first step towards the HW/SW interface, full data word read and write is introduced. For that a SIFv1 is introduced having only the `TXD_REG` and `RXD_REG`. Afterwards, the flags `data_transmitted` and `data_received` are experimentally implemented, each flag as an own register.

5.2.1 General Approach

Today's most often used HW/SW interface is a so called memory mapped HW/SW interface. Here, memory elements of hardware devices are mapped into the address space of the CPU executing the software. An address decoder takes (mostly the upper) bits of the address and converts them to select signals of the memory and the peripherals to be accessed. Potentially, additional address signals are passed to the memory and the hardware devices in order to select internal memory elements. An example is depicted in Fig. 5.1.

When the select signal is active, the memory elements are read when the read signal is active, or written when the write signal is active. In modern bus systems, address, data, enable, and read/write signals may be applied synchronously in different time windows in order to enable, for instance, pipelined access. They might also be encoded differently, for instance a read-not-write signal `R_Wbar` may replace the read `RD` and write `WR` signals in presence of a bus enable signal.

When the CPU executes a read or write operation, the signal values are set in an appropriate way in order to move one word from the memory cell or a peripheral register to the CPU register, and vice versa. This read or write CPU operation may be part of instructions that move data from and to variables of a higher level programming language. In this way, a memory-mapped HW/SW

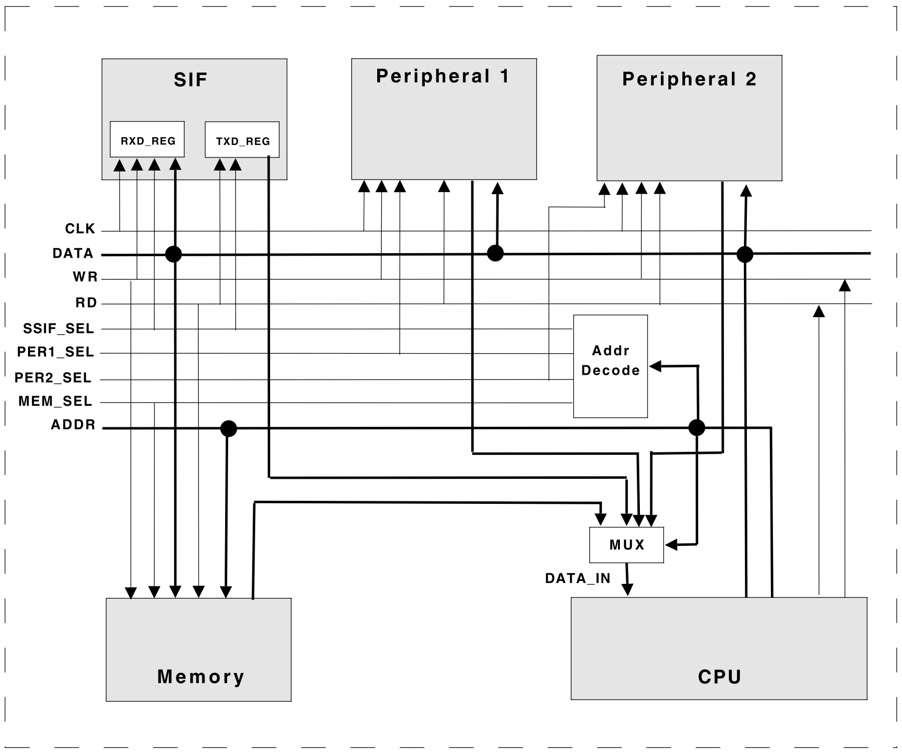

Figure 5.1. Simplified signal level connection of the SIF.

interface allows programing of a peripheral register interface using a higher level language.

As an alternative to the memory mapped HW/SW interface option, special instructions and hardware infrastructure may be provided as well. This option is often used for 8-bit CPUs, as Zilog's Z80 (see [Wik]), since the memory space is limited and shall be reserved for data memory / program memory purpose (i.e. no registers and no memory mapped I/O). As a drawback, this kind of interface has no correspondence in higher level languages, so that only assembler instructions can be used to transfer values over the interface. Higher level languages require some assembler code in-lining to support register access in this case.

In the future, also register mapped HW/SW interfaces may be introduced as a part of an I/O co-processor strategy. A first step in this direction is implemented in MIT's raw architecture (see [MIT]). Here, data can be transferred from one CPU to another by moving data to and from specific registers. Also, this kind of interface can only be programmed in assembler or with inlined assembler, since selected register access is not possible in higher level languages.

In the future, concurrent descriptions of the HW/SW interface may arise that can be used to automatically compile code for such concepts as well.

In the examples, a 32-bit CPU is assumed to be used supporting both 32 bit address size and 32 bit data size. The CPU is able to read and write bytes as well.

5.2.2 Full CPU Word Registers

As a first step, we introduce in the following two registers of the serial interface device—one can be written and one can be read. The memory space allocated by this first version is shown in Table 5.1.

To transmit data via the serial channel, this data must first be written to the `TXD` register. Data received from the serial channel can be read from the `RXD` register. This register has an offset of 0 to the base address of the peripheral as well.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32

Table 5.1. SIFv1 register overview.

Shown in Fig. 5.1 is an address decoder that computes the enable signals for all memory and peripheral devices communicating with the CPU. In the serial interface example, the address decoder takes the upper 8 bits of the address to compute 4 out of 256 possible enable signals. The byte address support of the CPU requires additional 2 address bits. So, each enabled device has available $32-8-2=22$ addresses for internal registers or memory space. So, a 16 MB memory ($4 \cdot 2^{20}$ addresses of 4 bytes each), for instance, can be enabled with one of these signals.

Assuming for the rest of this chapter, that the serial interface enable is active, when the upper 8 address bits take the value `0xFF`, the base address of the serial interface is `0xFF000000`. The memory address space reserved for the serial interface now ranges from this base address `0xFF000000` to address `0xFFFFFFFF`. In turn, the serial interface only provides memory cells for the base address `0xFF000000` and leaves the other addresses unused.

Accessing these registers from a C program can be done generally in two ways, object-based or function-based. In the first way, types and objects are declared and initialized to read and write the registers via those objects. In the second way, a function layer is introduced to allow access to the registers.

In the object-based alternative, first a pointer is declared for each register as shown in Listing 5.1.

The `volatile`-keyword in the listing gives the compiler the hint that the object may change without CPU interaction. So, an access to that object is not

```
#include <stdint.h>

volatile uint32_t *rxd_reg_ptr, *txd_reg_ptr;
```

Listing 5.1. Type and object declaration for direct register access.

removed by the optimizer of the C compiler. The type `uint32_t` is declared in `stdint.h` and specifies an unsigned 32 bit type independent from the target CPU.

The pointers are then initialized with the base address of the serial interface as shown in Listing 5.2. The distinction between the rxd-register and the txd-register is done in hardware via the read and write signal. Options herefore are discussed later in this chapter.

```
rxd_reg_ptr = (volatile uint32_t *) 0xFF000000;
txd_reg_ptr = (volatile uint32_t *) 0xFF000000;
```

Listing 5.2. Pointer initialization for direct object access.

The base address in these functions can be replaced by symbolic names—either constants or macros—, which will be shown in Listing 5.3. By doing so, the addresses can also be set via compile options.

```
#define SIF_BASE_ADDRESS 0xFF000000

rxd_reg_ptr = (uint32_t *) SIF_BASE_ADDRESS;
txd_reg_ptr = (uint32_t *) SIF_BASE_ADDRESS;
```

Listing 5.3. Pointer initialization for direct object access with symbolic values.

To preserve type consistency, the integer literal is cast to the register pointer type. Finally, transmitted data can then be accessed by dereferencing the pointers as shown in Listing 5.4. The type of the variable `rx` is assumed to be `uint32_t`. The integer literal is cast to that value.

```
/* reading a value from serial stream */
rx = *rxd_reg_ptr;

/* writing a value to the serial stream */
*txd_reg_ptr = (uint32_t) 0x12345678;
```

Listing 5.4. Accessing the SIF register.

In the function-based alternative, first types and access functions are declared, as shown in Listing 5.5. In this simple case, these functions contain exactly the statements of the object-based access alternative. These functions can then be called—which is not shown in the code snippet—to transmit data via the SIFv1.

```
void transmit(uint32_t data) {
   *(0xFF000000) = data;
}

uint32_t receive() {
   return (uint32_t) *(0xFF000000)
}
```

Listing 5.5. Type and function declaration for data transmission.

Two further coding options are in use for accessing registers. The first one uses macros instead of functions. This is more efficient, if the compiler does not support function inlining optimizations. The other option uses classes, class variables, and class methods to access the peripheral registers. Here, the access to the registers of the serial peripheral device can be controlled more efficiently (e.g., via private and public access rights, or via additional checks), but the C++ compiler is mostly not able to produce as efficient a code as the C-compiler, since the overhead caused by the class-based approach is often not eliminated.

5.2.3 Registers Storing One Bit Each: A First Approach to Bit Fields

It is quite obvious that a two register interface so far only works correctly if the hardware read/write protocol blocks the read and write transactions until they have been successfully finished. This means in the serial interface device case, the read is blocked until a piece of data is successfully received from the serial stream, and a write is blocked until a piece of data has been correctly transmitted to the serial stream. This also blocks the CPU and prevents it of from performing other activities. This is no ideal solution!

To avoid blocking the execution of other parts of SW, the SW must be able to check if the serial interface can transmit further data, or if the serial interface has received new data that can be read. Two additional registers, each storing only zero or one, can offer this information to the software. The device is now called SIFv2.

The interface of the SIFv1 extends as shown in Fig. 5.2 and Table 5.2.

Two additional, readable registers are introduced that also require two additional address lines for distinction. A multiplexer is used to internally select the appropriate register value. For bus accesses, the SIFv2 now has one writable register and three readable registers. The ready-for-transmission register has an offset of 1, and the data-available register has an offset of 2.

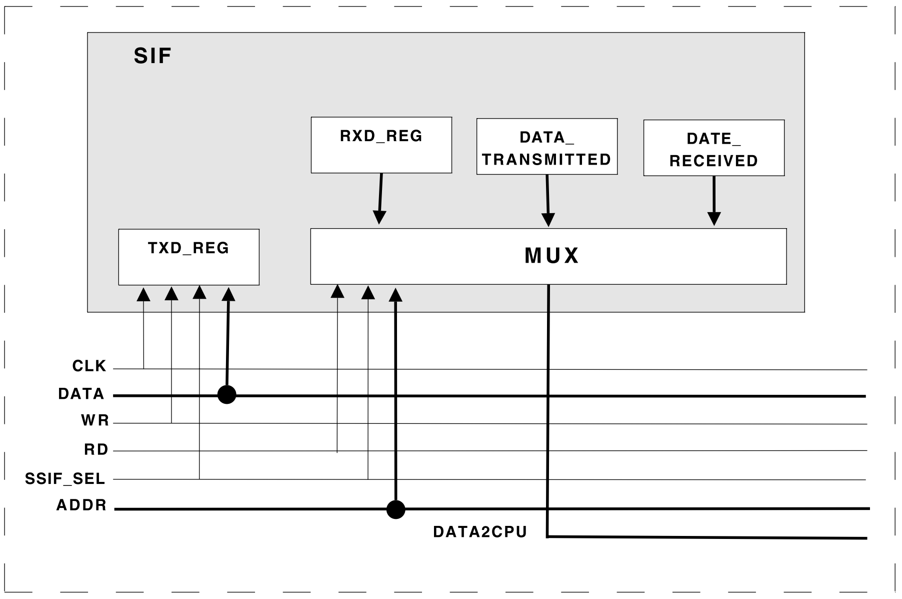

Figure 5.2. Simplified signal level register access.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_TRANSMITTED	R	1	32	32
SIF_RECEIVED	R	2	32	32

Table 5.2. SIFv2 register overview.

For this purpose, the SW interface for readable registers must be extended as well. As an initial solution, two more register pointers are declared and initialized, as shown in Listing 5.6.

The offset of the new registers is hard-coded in each pointer initialization and added here to the base address. Since the CPU supports byte access, the increment between two registers is 4.

The read access to the new status registers is similar to the received data as shown in Listing 5.4.

Listing 5.7 shows a more elegant and more frequently used option. Here, a C struct is used to describe all registers. Since each of the entries in the C struct has the size of the CPU word, the address of each entry automatically increases by 4. The explicit increment of addresses, as used in Listing 5.6, is not necessary here.

```
volatile uint32_t *rxd_reg_ptr,
                  *txd_reg_ptr,
                  *data_transmitted_ptr,
                  *data_received_ptr;

/* Initialize references to readable registers */
rxd_reg_ptr          = (uint32_t *) 0xFF000000;
data_transmitted_ptr = (uint32_t *) 0xFF000004;
data_received_ptr    = (uint32_t *) 0xFF000008;

/* Initialize references to writable registers */
txd_reg_ptr          = (uint32_t *) 0xFF000000;
```

Listing 5.6. Type and object declaration for additional flags.

```
volatile uint32_t *txd_reg_ptr;

struct r_reg_t {
  volatile uint32_t rxd_reg;
  volatile uint32_t data_transmitted;
  volatile uint32_t data_received;
} *r_reg_ptr;
```

Listing 5.7. Type and pointer declaration for data transmission.

In order to access the registers, the dereferencing mechanism of C is used, as shown in Listing 5.8. Since a pointer is used to refer to the registers, the dereferencing operator −> is used.

```
/* reading a value from serial stream */
rx = r_reg_ptr -> rxd_reg;

/* read flags */
ready_for_transmission = r_reg_ptr -> data_transmitted;
data_available         = r_reg_ptr -> data_received;
```

Listing 5.8. Object access to flag registers.

```
uint32_t is_ready_for_transmission() {
  return (uint32_t) *(0xFF000000)[1];
}

uint32_t is_data_ready() {
  return (uint32_t) *(0xFF000000)[2];
}
```

Listing 5.9. Function access to flag registers.

The function-based approach—to show an alternative access in Listing 5.9—makes use of the C index operator. Beginning with the base address of the serial interface peripheral, the `rxd_reg` and the transmission flags can be accessed by indices 0, 1, and 2, respectively. The compiler converts this to an address increment of the size of referenced elements in byte. In our case, the size is 32 bits or 4 bytes. So, the index of the transmission flags are addressed under `0xFF000004` or `0xFF000008`.

Also a mix of both approaches—object-based register access and method-based register access—is possible. Doing so, the data structures of the object-based access are used by the functions to implement the access. As benefit over pure object-based access, the functions provide an additional layer, which hides future HW changes from higher level software. Also additional sanity checks are possible to be embedded here.

Registers holding transmission status and other things often use just one bit, or a few bits. This is waste of address space but potentially acceptable if a sufficient number of address lines, that means a sufficiently large address space, is available. However, peripheral register accesses cause performance penalties. Merging several of those bits efficiently in one register can reduce the number of register accesses (e.g. for peripheral configuration, or check) and thus improve performance.

Further on, it is quite error prone that readable and writable registers have a different address size and information located at different addresses. It is more clear and safe to have one linear address space only, which in turn would allow having one C struct specifying all registers—the readable registers, the writeable registers and the read/writable registers—at the SW side. Both of these issues are discussed in the next section in more detail.

5.3 Bit Fields

The need for bit fields sharing one register has been introduced in the previous section. Now, different aspects of the bit field based interface are discussed. It is shown that in this application registers lose their role as memory element and take the role of a shell accessed under a specific address. In other words, registers take the role of an alias.

Bit fields—now representing the memory elements—are associated with that shell, which specifies their word access from the CPU. Internal bit offsets, as shown for the serial interface in Table 5.9, are used for bit addressing in the data word. Besides the data registers, txd-register and rxd-register, and the bit fields `data_transmitted` and `data_received` also the bit fields controlling the parity bit (`tx_enable_parity`, `tx_odd_parity`, `rx_enable_parity`, and `rx_odd_parity`) are introduced.

5.3.1 Introducing Bit Fields

In order to introduce bit fields, a third version of the serial interface SIFv3, is introduced. The registers are shown in Table 5.3.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_FLAG_REG	R	1	32	32

Table 5.3. SIFv3 register overview.

Readable Bit Fields Instead of having two separate registers storing the two flags `data_transmitted` and `data_received`, one common flag register is used. This flag register stores the `data_transmitted` flag at bit number 0, and the `data_ received` flag at bit number 1.

All other bits with numbers from 2 to 31 are unused. Since no hardware resources need to be spent to store the unused bits, the bit fields (and not the registers) specify the hardware size and properties of the required flipflops or registers. Reading from those bits cannot be avoided, since the CPU always reads a full 32 bit word. Depending on the specification, an undefined value or a constant, for instance 0 as used in the example, is returned for each of those bits. The updated diagram of the serial interface is shown in Fig. 5.3.

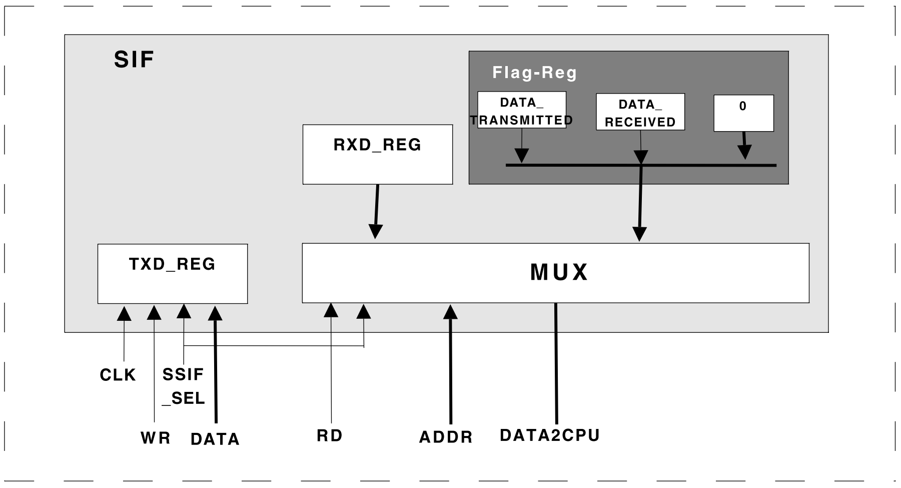

Figure 5.3. Registers with readable bit fields.

To access these flags, so-called bit fields in C are used. First, as shown in Listing 5.10, a C struct specifies the flags and unused bits of one register. Next, a struct is defined that specifies the readable registers: the `rxd_reg_data` register and the `flag_reg` flag register.

```
volatile uint32_t *txd_reg_ptr;

struct flag_register_t {
   volatile uint32_t data_transmitted : 1;
   volatile uint32_t data_received :1;
   const    uint32_t unused :30;
}

struct r_reg_t {
   volatile uint32_t          rxd_reg;
   volatile flag_register_t   flag_reg;
} *r_reg_ptr;
```

Listing 5.10. Type and object declaration for bit field based register access.

The bit fields are then accessed with the dereferencing operator −>, as shown in Listing 5.11. The bit fields holding the flags are selected as struct elements in the corresponding register using the dot operator. The way of reading a full data word from a register remains unchanged.

```
/* reading a value from serial stream */
rx = r_reg_ptr -> rxd_reg;

/* read flags */
ready_for_transmission = r_reg_ptr -> flag_reg.data_transmitted
    ;
data_available         = r_reg_ptr -> flag_reg.data_received;
```

Listing 5.11. Register flag access via C bit fields.

A function-based coding option is shown in Listing 5.12. Instead of using a pre-initialized pointer, which would have been possible here as well, hard-coded addresses are used. The data register is read without offset, the flag register is read via base address and index offset 1. The compiler, knowing the address scheme of the CPU, translates this to the address `0xFF000004`. In case of bits not residing at bit position 0, the bit fields themselves are then extracted via masking the unused bits by performing a bitwise `and` operation with a corresponding bit mask. Potentially—as needed for the `data_ready_flag`—the bits have to be right aligned upfront by using the C shift right operation.

Writable Bit Fields In order to write configuration information to the serial interface, an additional writable register with bit fields is introduced as SIFv4 in Fig. 5.4 and Table 5.4.

```
uint32_t is_ready_for_transmission() {
   return (uint32_t) (*(0xFF000000)[1]) & 0x00000001;
}

uint32_t is_data_ready() {
   return (uint32_t) ((*(0xFF000000)[1]) >> 1) & 0x00000001;
}
```

Listing 5.12. Register flag access via C shift and logical operators.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_FLAG_REG	R	1	32	32
SIF_CFG_REG	W	1	32	32

Table 5.4. SIFv4 register overview.

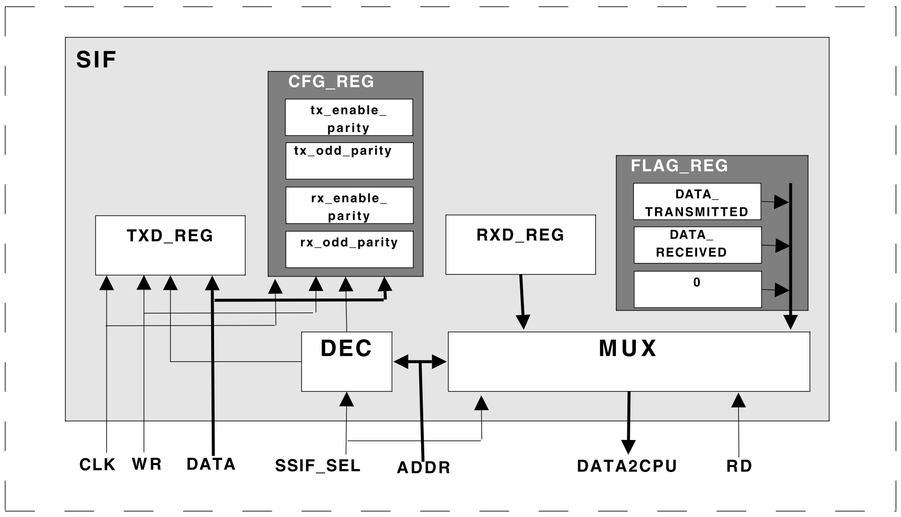

Figure 5.4. Registers with writable bit fields.

A decoder is inserted here in order to decode address values to register select signals.

None of these bit fields can be written individually, so writing to one bit field automatically modifies the other bit fields. However, the configuration register—and so the bit fields—cannot be read, since reading address 1 gets the value of the flag register.

For this purpose, as a first solution, a shadow variable is introduced that holds the last written value to the configuration register. Setting or clearing a bit field is then first performed on the shadow variable, and afterwards the shadow variable is written to the register.

```
struct confi_reg_t {
   volatile uint32_t tx_enable_parity   : 1;
   volatile uint32_t tx_odd_paroity     : 1;
   volatile uint32_t tx_enable_parity   : 1;
   volatile uint32_t tx_odd_paroity     : 1;
   volatile uint32_t                    : 28;
}

struct w_reg_t {
   volatile uint32_t          txd_reg;
   volatile config_reg_t      config_reg;
} *w_reg_ptr;
```

Listing 5.13. Type and object declaration for writing configuration bit fields.

```
/* shadow variable for configuration register with bitfields */
static config_reg_t config_reg_shadow;

/* write flags via shadow variable */
config_reg_shadow.tx_enable_parity = 1;
w_reg_ptr -> config_reg = config_reg_shadow;

/* writing a value to serial stream */
w_reg_ptr -> txd_reg = 0x12345678 ;
```

Listing 5.14. Flag configuration via C bit fields.

As shown in Listing 5.13, structs for written bit fields are declared in the same way as for read bit fields. The use of the above mentioned shadow variable is shown in Listing 5.14. Since setting a bit field with one statement is in this case no longer possible, the use of functions for setting and clearing the bits, as shown in Listing 5.15, is the preferable solution. Besides encapsulation of the shadow register handling, additional checks may be included as well.

Listing 5.15 shows several alternatives how access functions for registers and bit fields can be built. The function `write_cfg_tx_enable_parity` sets the bit fields via bit wise `or` operation. As an alternative, the C bit fields and the structs defined for object based register access can be used as shown in function `write_cfg_ tx_odd_parity`. Without them, a bit wise `and` operation with `0xFFFFFFFE` would have been used (bit masking of the least significant bit). Finally, the function `write_cfg_tx_odd_ parity` combines two flag accesses on the shadow variable before writing it to the registers. This improves performance, since the shadow variable needs to be written to

```
/* shadow variable for configuration register with bitfields */
static config_reg_t config_reg_shadow;

void write_cfg_tx_enable_parity() {
   config_reg_shadow |=  0x00000001;
   *(0xFF000000)[1] = config_reg_shadow;
}

void write_cfg_tx_odd_parity() {
   config_reg_shadow.tx_odd_parity =  1;
   w_reg_ptr -> config_reg = config_reg_shadow;
}

void set_rx_parity(bool odd_parity) {
   config_reg_shadow.rx_odd_parity =  odd_parity;
   config_reg_shadow.rx_enable_parity = 1;
   w_reg_ptr -> config_reg = config_reg_shadow;
}

void clear_rx_parity() {
   config_reg_shadow.rx_enable_parity = 0;
   w_reg_ptr -> config_reg = config_reg_shadow;
}
```

Listing 5.15. Register flag access via C shift and logical operators.

the register only once, but semantic information about the content of the bit fields is required in this case.

Mixed Readable and Writable Bit Fields Conceptually, redundancy as existent in the configuration register and its shadow variable is a source of bugs. Furthermore, shadowing provides no appropriate solution if a bit field is also modified by peripheral internal actions. Hence, in a new version SIFv5, the writable bit fields shall be readable as well.

Therefore, the readable bit fields in the flag register are extended by the writable bit fields as shown in Listing 5.16 and Table 5.5. In order to make them read/writable, the write must be triggered by CPU write to the appropriate address, and the value must be multiplexed to the output by a CPU read.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_FLAG_REG	RW	1	32	32

Table 5.5. SIFv5 register overview.

```
struct flag_reg_t {
    volatile uint32_t data_transmitted  : 1;
    volatile uint32_t data_received     : 1;
    volatile uint32_t tx_enable_parity  : 1;
    volatile uint32_t tx_odd_paroity    : 1;
    volatile uint32_t tx_enable_parity  : 1;
    volatile uint32_t tx_odd_paroity    : 1;
    volatile uint32_t                   : 26;
}

struct r_reg_t {
    volatile uint32_t     rxd_reg;
    volatile flag_reg_t   flag_reg;
} *r_reg_ptr;

struct w_reg_t {
    volatile uint32_t           txd_reg;
} *w_reg_ptr;
```

Listing 5.16. Readable and writable register with flags.

Having this structure available, the flags can be changed consistently, as shown in Listing 5.17.

In function `write_cfg_tx_enable_parity`, first the current values of the flags are read from the status register, then aligned, and next assigned to the temporary shadow variable `config_reg_shadow`. Afterwards, the flag is set in the temporary variable. Finally, the updated flags stored in the temporary variable are assigned to the bit fields in the register. The temporary variable `config_reg_tmp` is not necessary, the flag update might have been done in one expression alone.

```
void write_cfg_tx_enable_parity() {
    config_reg_t config_reg_tmp = (config_reg_t)
        (((*0xFF000000)[1]) >> 2) & 0x0000000F;
    config_reg_tmp |=  0x00000001;
    *(0xFF000000)[1] = config_reg_tmp;
}
```

Listing 5.17. Bit field update by reading and writing a complete register.

Access control for bit fields Another alternative, SIFv6, to overcome shadow registers is the introduction of additional bits in combination with new coding as it is shown in Table 5.6.

For one flag, generally three options are possible when adding an additional bit for control:

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	32
SIF_RXD_REG	W	0	32	32
SIF_CONFIG_REG	W	1	32	32

Table 5.6. SIFv6 register overview.

- The two bits in the register act as RS signals to the flipflop storing the flag. If “`00`” is assigned, then nothing changes, if “`01`” is assigned, then the flipflop, i.e. the flag, is set, and if “`10`” is assigned, then the flipflop is cleared. The assignment “`11`” is illegal.
- The two bits in the register act as JK signals to the flipflop. As expected from the JK-flipflop behavior, this has the same behavior as the RS option, except the case “`11`” is legal and forces the toggle of the flag.
- One bit in the register acts as new value for the flag and the other as enable. This alternative can also be used to enable several bits in parallel, for instance, to write a configuration value that has more than two alternatives.

It is obvious to say that control in the options above may be coded in a different way, for instance in a low active way. Having enable and data in different registers is also an option but requires three register accesses: enable–write–disable.

Listing 5.18 shows the application of the third alternative to the parity configuration bits. Each bit, marked by the name suffix `_val` is guarded by an enable bit with the name suffix `_en`.

```
struct config_reg_t {
   volatile uint32_t tx_enable_parity_val : 1;
   volatile uint32_t tx_enable_parity_en  : 1;
   volatile uint32_t tx_odd_parity_val    : 1;
   volatile uint32_t tx_odd_parity_en     : 1;
   volatile uint32_t rx_enable_parity_val : 1;
   volatile uint32_t rx_enable_parity_en  : 1;
   volatile uint32_t rx_odd_parity_val    : 1;
   volatile uint32_t rx_odd_parity_en     : 1;
   volatile uint32_t unused               : 24;
}

struct w_reg_t {
   volatile uint32_t       txd_reg;
   volatile config_reg_t   config_reg;
} *w_reg_ptr;
```

Listing 5.18. Bit field access via access control.

The flags can now be easily set or reset, as shown in Listing 5.19. Here, the `tx_ enable_bit` is set and the `tx_odd_parity` is cleared. The bit fields configuring the RX path are not changed, since the corresponding enable bits are not set.

```
/* enable parity in the transmission path */
w_reg_ptr -> config_reg = 0X00000003;

/* set odd parity in the transmission path */
w_reg_ptr -> config_reg = 0X00000008;
```

Listing 5.19. Register bit field access with guarded value access.

Unfortunately, the bit fields in C cannot be used since two of them cannot be assigned in one statement. For this reason, it is better to define the two bits—one for the value and one for the enable—in one entry of the struct. Making use of such a type definition, as shown in Listing 5.20, finally allows to set or clear the bit with one statement, as shown in Listing 5.21. Here, also the symbolic names `set_bit`, `clear_bit`, and `keep_bit` are defined as constants. Macros are also an option here.

```
constant uint32_t set_bit   : 2 = 3;
constant uint32_t clear_bit : 2 = 1;
constant uint32_t keep_bit  : 2 = 0;

struct config_reg_t {
   volatile uint32_t tx_enable_parity : 2;
   volatile uint32_t tx_odd_parity    : 2;
   volatile uint32_t rx_enable_parity : 2;
   volatile uint32_t rx_odd_parity    : 2;
   volatile uint32_t unused           : 24;
}

struct w_reg_t {
   volatile uint32_t        txd_reg;
   volatile config_reg_t    config_reg;
} *w_reg_ptr;
```

Listing 5.20. Bit field via dual-bit access control.

```
/* enable parity in the transmission path */
w_reg_ptr -> config_reg.tx_enable_parity = set_bit;

/* set odd parity in the transmission path */
w_reg_ptr -> config_reg.tx_odd_parity = clear_bit;
```

Listing 5.21. Register flag access via enabled setting of bit fields.

If the bit fields are modified by the peripheral core as well and new values have an influence on the new settings of the bit fields, then they must be made readable as well.

A structured approach to Bit Fields. All options shown above have both advantages and disadvantages. Avoiding side effects partially is not possible, if bit fields are changed by the peripheral core; bit fields have to be specified twice in the read registers and write registers, in order to allow reading and writing them; reading and writing bit fields might have to be done at different addresses or bit offsets; re-assignment of bit sets might be needed.

For this reason, we propose to reserve one address for each register, independently if it is written, read, or read and written. The read and write signal is no longer used to distinguish bit fields. Furthermore, we request, that registers that are written by software, can also be read by software.

This implies that multiplexers and decoders must be implemented here in such a way that values and bit fields can be accessed under the same address and bit position for read and write access.

The application of this methodology to an industrial style peripheral is shown after discussion of impact of bus infrastructure and a textual specification of the serial interface peripheral later on.

5.4 Register Address and Data Mismatch

In this section, advanced topics on the HW/SW interface are discussed. They deal mostly with not having a bijective mapping between CPU address space and peripheral address space. To give examples, the IP may have holes in the data and address space, multiple registers may be accessed under one address, or one register may be accessed under multiple addresses.

5.4.1 Hierarchical Bus

In order to relate the advanced topics to real architecture structures, our bus is extended in direction of a hierarchical bus (see Fig. 5.5). The bus is split in a CPU bus—mostly a high speed bus—and a peripheral bus—mostly a slower bus with potentially less address signals and data signals.

As interface between these buses, a so called bridge is introduced. In the example, the bridge is selected by the same signals as formerly the serial interface unit. Since the bridge introduces a new hierarchy in the global address map, the addressing scheme so far needs to be updated slightly. Thus, the first 8 address bits are now used to select devices connected to the CPU bus. The bridge to the peripheral bus can be considered as such a device. The remaining 24 address lines are passed to the bridge. The upper 8 lines (lines 24 to 16)

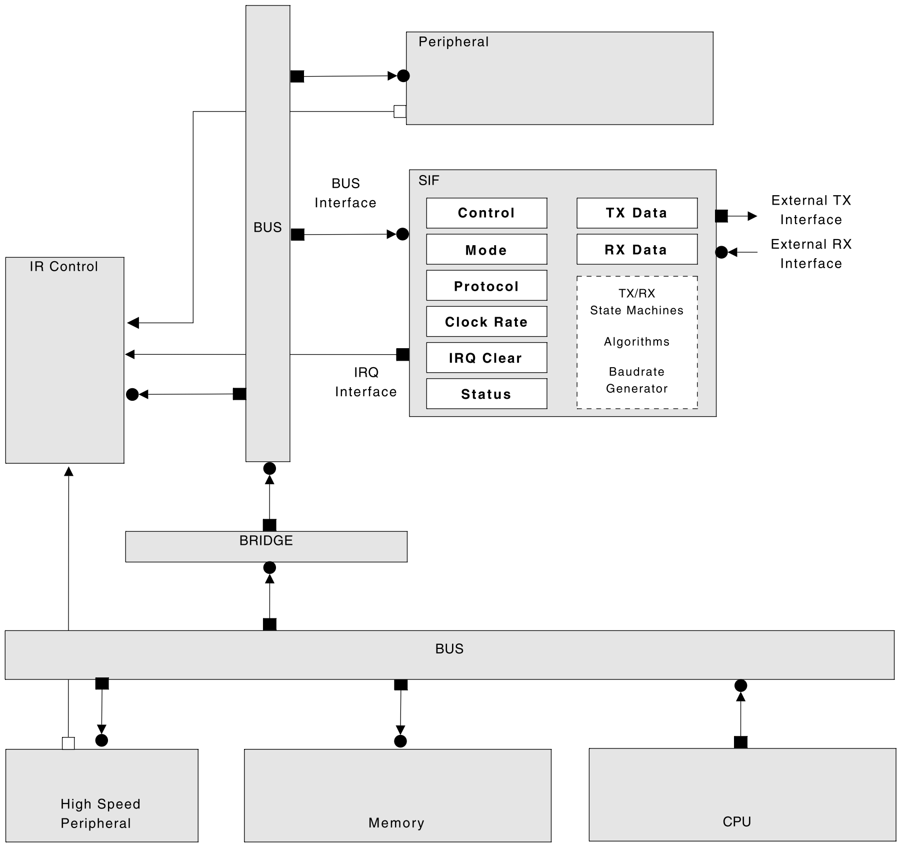

Figure 5.5. Simple hierarchical bus system.

are decoded within the bridge in order to select a specific device (i.e., peripheral) connected to the peripheral bus. With regard to these 8 address bits, the address of the serial interface unit is now assumed to be `0xFF`. Due to this adapted address map, the new base address of the serial interface unit from the CPU perspective is `0xFFFF0000`. This value is the compound of the base address of the bus bridge (i.e., `0xFF000000`) and the base address of the serial interface unit within the peripheral bus (i.e., `0xFF0000`). This yields 16 bits for addressing resources within the serial interface peripheral. Taking into account that the lower two bits are reserved for byte addressing, 14 bits remain for addressing internal registers of the serial interface which restricts the maximum number of addressable 32-bit registers of the serial interface unit to $16 \cdot 2^{10}$—still a more than sufficient number.

5.4.2 Byte Addressing

Most embedded 32 bit CPUs also support half-word and byte addressing. In this case, not only 32 bits can be accessed at once, but also just 16 bits and 8 bits. Byte addressing with 32 bit wide RXD registers and TXD registers does not provide any benefit, in fact, it makes the access worse. To show this case, a SIFv7 is introduced in Table 5.7. Here, each register has its own address. The addressable unit has changed from 32 bit to 8 bit.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	32	8
SIF_RXD_REG	W	1	32	8
SIF_CONFIG_REG	W	2	32	8
SIF_FLAG_REG	R	3	32	8

Table 5.7. SIFv7 register overview.

```
struct flag_register_t {
    volatile unsigned data_transmitted : 1;
    volatile unsigned data_received :1;
    volatile unsigned :6;
}
struct config_reg_t {
    volatile unsigned tx_enable_parity : 1;
    volatile unsigned tx_odd_parity    : 1;
    volatile unsigned rx_enable_parity : 1;
    volatile unsigned rx_odd_parity    : 1;
    volatile unsigned unused           : 4;
}

struct reg_t {
    volatile uint32_t         rxd_reg;
    volatile uint8_t          txd_reg[4];
    volatile config_reg_t     config_reg;
    volatile flag_register_t  flag_reg;
} *reg_ptr;
```

Listing 5.22. Struct for byte addressing.

Listing 5.22 shows a data structure definition that matches with byte addressing. The flag register and configuration register is filled up to 8 bits only. The 32 bit data is now represented by a 4 element array of the data type `uint8_t`. This `uint8_t` is also taken from the `stdint.h` include file. If 4 byte alignment of the structs is ensured—as is the case in the example—also the type `uint32_t` can be used instead of the 4 element array. In both cases, the data word can be written at a glance using CPU word access—but only by using ugly casting.

If 4 byte alignment of the structs cannot be guaranteed—as would be if the 8 bit flag register preceded the 4x 8 bit data register—the data word must be assigned byte by byte, which would cause a 4x overhead in writing the data. Both, flag register and config register can be accessed in any case by a byte access without any penalty.

If only 8 bit data registers were assumed—now in SIFv8—the access would become simpler. This is shown in Table 5.8 and Listing 5.23.

Register	AccessExt	Offset	Width	AddrUnit
SIF_TXD_REG	R	0	8	8
SIF_RXD_REG	W	1	8	8
SIF_CONFIG_REG	W	2	8	8
SIF_FLAG_REG	R	3	8	8

Table 5.8. SIFv8 register overview.

```
struct flag_register_t {
    volatile uint32_t data_transmitted : 1;
    volatile uint32_t data_received :1;
    volatile uint32_t :6;
}
struct config_reg_t {
    volatile uint32_t tx_enable_parity : 1;
    volatile uint32_t tx_odd_parity    : 1;
    volatile uint32_t rx_enable_parity : 1;
    volatile uint32_t rx_odd_parity    : 1;
    volatile uint32_t unused           : 4;
}

struct reg_t {
    volatile uint8_t          rxd_reg;
    volatile uint8_t          txd_reg;
    volatile config_reg_t     config_reg;
    volatile flag_register_t  flag_reg;
} *reg_ptr;
```

Listing 5.23. Struct for byte addressing with byte data words.

As a drawback, only 8 bit values can be transmitted. Also (not shown), the hardware architecture must be changed accordingly.

5.4.3 Endianness

Mix of byte access and word access becomes challenging if different orders of bytes are implemented on the CPU side and on the serial interface side. This implementation aspect, also known as endianness, requires additional adaptation effort, either on the HW side, or on the SW side.

Best known are big endian, the numeric option, where the most significant byte comes first, and little endian, the literal option, where the most significant byte comes last. The little endian option also has the advantage of that a byte value followed by three zero byte values is read in the same way in case of byte, half-word, and word access.

Also known is a middle endian or mixed endian option (16 bit), where the most significant half word comes last and the least significant half word comes first. The advantage of this version is the seamless support of 16 bit encoded characters.

The easiest and most efficient adaptation of different endian encodings is on the hardware side. First, the peripheral can have a generic parameter, that allows to statically reconfigure a peripheral interface to big endian or little endian. This allows avoiding the remapping effort since the endianness of the CPU and the peripheral can be made identical for the cost of rewiring some signals, in other words, for free. A remapping of the bytes at the ports of the peripheral has the same effect. Also an easy and quite efficient adaptation is the use of CPUs that can be dynamically reconfigured to support big endian or little endian reading and writing of 4 byte words. This costs some hardware overhead but has the advantage that a mix of big endian and little endian peripherals can be supported as well.

Quite an overhead must be spent to adopt endianness in software. Listing 5.24 shows a possible implementation.

```
BE = (  ((LE)>>24)            | (((LE)&0xff0000)>>8) |
        (((LE)&0xff00)<<8)   | ((LE)<<24) )
```

Listing 5.24. Expression for converting little endian to big endian.

Sometimes—mainly in context of serial transmission—the term endianness is also used in conjunction with bits. In this case, the terms byte endianness and bit endianness are applied to make a distinction. Byte endianness is the classical endianness as described above. Bit endianness relates to bit orientation and defines if inside one byte, the most significant bit is first and the last significant bit is last, or vice versa.

5.4.4 Busses with Different Data Width

A hierarchical bus system also allows busses with different data widths, for instance a CPU bus size of 32 bit and a peripheral bus size and register size of 8 bit each. All presented codings shown above can be kept if the bus bridge serializes word access on the CPU side into 4 byte accesses—of course considering the right endianness.

Another option is to use the lower byte of the CPU word only and ignore the higher three bytes. In this case, a lot of unused bytes have to be filled into the struct representing the registers in software. This is shown in Listing 5.25.

```
struct flag_register_t {
    volatile uint32_t data_transmitted : 1;
    volatile uint32_t data_received :1;
    volatile uint32_t :6;
}
struct config_reg_t {
    volatile uint32_t tx_enable_parity : 1;
    volatile uint32_t tx_odd_parity    : 1;
    volatile uint32_t rx_enable_parity : 1;
    volatile uint32_t rx_odd_parity    : 1;
    volatile uint32_t unused           : 4;
}

struct unused_upper_bytes_t{
    volatile uint32_t : 24;
}

struct reg_t {
    volatile uint8_t               rxd_reg;
    volatile unused_upper_bytes    unused1;
    volatile uint8_t               txd_reg;
    volatile unused_upper_bytes    unused2;
    volatile config_reg_t          config_reg;
    volatile unused_upper_bytes    unused3;
    volatile flag_register_t       flag_reg;
    volatile unused_upper_bytes    unused4;
} *reg_ptr;
```

Listing 5.25. Struct for LSB byte addressing with byte data words.

5.4.5 Several Registers Share One Address

Not only read and write signals are used to reduce consumed address space of a peripheral. Also other techniques have been invented to extend the number of addressable registers and bit fields. These techniques are also heavily used in 8 bit CPU systems but lose their importance as more and more address space, for instance in 32 bit CPUs, becomes available.

Auto-shadow is the first technique to be discussed. Here, one register is visible at a specific address after hardware or software reset. After having written to such a register, the register hides behind another register that can be accessed instead. This is mostly applied for configuration registers, since they are mostly configured once. On the software side, these two registers share their address by being modeled as a union, as it is shown in Listing 5.26.

```
struct confi_reg_t {
   volatile uint32_t tx_enable_parity  : 1;
   volatile uint32_t tx_odd_paroity    : 1;
   volatile uint32_t tx_enable_parity  : 1;
   volatile uint32_t tx_odd_paroity    : 1;
   volatile uint32_t unused            : 28;
}

union w_reg_t {
   volatile uint32_t        txd_reg;
   volatile config_reg_t    config_reg;
} *w_reg_ptr;
```

Listing 5.26. Type and object declaration for writing hidden configuration bit fields.

Writing these registers does not distinguish between the union or struct version. The dot-operator is used in both cases. However, it must be ensured that the hidden register is accessible when needed.

Another technique is the use of indexing bit fields. This can be done by interpreting one part of a register as a value and the other as index. Also, value and index can come from different registers. Writing to a register requires in the first case to merge index and data into one word—comparable with merging bit field information as discussed above. Writing to a register in the second case requires an overhead of two write accesses, which may be acceptable if the index, that means the bit field to be written to, changes only infrequently.

5.4.6 One Register is Accessible via Several Addresses

This is exactly the opposite hardware implementation approach as described in the section above. Here, one register or bit field can be accessed under more than one address in the peripheral's address space. Since this technique allocates more addresses in the address space than absolutely necessary, it finds its application more in 32 bit CPUs.

The first reason for such a technique is compatibility with older versions. So to say, an alias to the old address is preserved. In the SW side of the interface, two registers are specified in the register struct, but using different names.

The second, and probably more important reason is the support of burst or block transfers by the bus protocol. This burst transfer moves blocks from one address to another—or the cache. To apply this protocol, for instance to send data stored in the memory via the serial interface, the txd-register must be accessible under a range of addresses. The register struct can be easily

```
struct flag_register_t {
   volatile uint32_t data_transmitted : 1;
   volatile uint32_t data_received :1;
   volatile uint32_t unused :30;
}
struct config_reg_t {
   volatile uint32_t tx_enable_parity  : 1;
   volatile uint32_t tx_odd_paroity    : 1;
   volatile uint32_t tx_enable_parity  : 1;
   volatile uint32_t tx_odd_paroity    : 1;
   volatile uint32_t unused            : 28;
}

struct reg_t {
   volatile uint32_t          unused1;
   volatile uint32_t          unused2;
   volatile config_reg_t      config_reg;
   volatile flag_register_t   flag_reg;
   volatile uint32_t          rxd_reg[32];
   volatile uint32_t          txd_reg[32];
   volatile config_reg_t      config_reg_alias;
   volatile flag_register_t   flag_reg_alias;
} *reg_ptr;
```

Listing 5.27. Type and object declaration for writing hidden configuration bit fields.

extended by putting an array—here of size 32—instead of a single element to the register struct. This is also known as mirror size 32.

Listing 5.27 shows exemplarily how alias registers and memory space for allocation of data registers have an impact on the register struct. First, two unused registers are introduced to keep the flag register and config register at their corresponding places. After the config registers follows the allocated space—implemented as arrays as described above—and finally the alias registers.

5.4.7 Multiple SIF-Peripherals in One System

If more than one serial interface has to be used in a system, several base addresses have to be served from the software point of view. This can be easily achieved by declaring and initiating two register pointers.

To access those peripherals, either the code has to be duplicated, for instance in object based access, or a register pointer has to be passed to the functions accessing the serial interfaces. The latter case is shown in Listing 5.28.

```
void write_cfg_tx_enable_parity(reg_t *reg_ptr) {
   reg_ptr -> config_reg |= 1;
}
```

Listing 5.28. Setting and clearing bit fields with reading.

To call the functions, the struct declaration of Listing 5.22 is needed. As pointers, now a `sif1_reg_ptr` and a `sif2_reg_ptr` must be declared (not shown in a listing). These pointers must be initialized with the serial interfaces' base addresses before they can be used in a call to those functions.

5.5 Textual Specification of the SIF

The following section gives an overview of the SIF peripheral. The hardware interfaces are explained, followed by the description of its registers and bit fields and their specification. The registers including their bit fields are the basic building blocks of the HW side of the HW/SW interface of the peripheral.

5.5.1 Overview

The SIF model can be used for connecting two hardware systems via a serial communication protocol.

Figure 5.6 shows the basic structure of the SIF peripheral model. The SIF contains four different hardware interfaces—the bus interface, the external interfaces, and an interface for interrupts. A detailed description of these interfaces will be given later on.

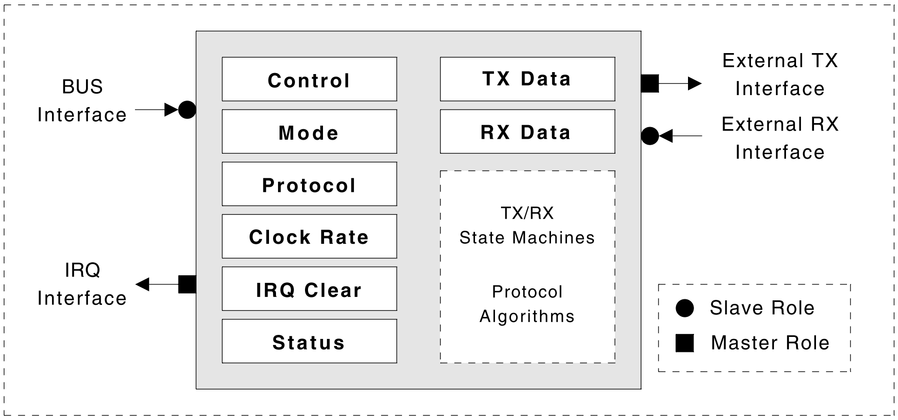

Figure 5.6. SIF structure overview.

Besides the hardware interfaces, the SIF model contains several registers as shown in Fig. 5.6. These registers are accessible from the CPU core (this denotes the CPU where the SW is executed) over the bus interface. Therefore, the registers are an essential part of the HW/SW interface. The SW can configure the behavior of the SIF by writing to the control and protocol registers. Furthermore, it can configure the baud rates for transmission and reception by writing to the clock rate register. The SW can clear interrupts by writing to the

IRQ clear register and it can query the peripheral status by reading from the status register. Data to be transmitted is written to the TX data (TXD) register, and received data can be retrieved by reading from the RX data (RXD) register.

Depending on the current protocol and control register settings, the TX and RX state machines perform different algorithms which influence the transmission and reception (e.g., heading reversal or bit inversion of transmitted/received bits). A detailed description of the registers including their bit fields is given later in this section.

5.5.2 Hardware Interfaces

Bus Interface. The SIF peripheral model can be connected to a bus through its bus interface. This bus interface acts as a slave when connected to a bus. Hence, the SIF cannot actively request a bus access. It reacts on read and write accesses from devices connected to the bus as master. The structure of the SIF bus interface is specified corresponding to the applied bus protocol—in this example a simple bus protocol.

The CPU core, which is usually connected as a bus master, can access the registers of the SIF through this bus interface. For selecting a specific register of the SIF, the core needs to set an address (in this case the sum of the SIF base address and the internal offset of the targeted register). The decoding of the register offset is implemented within the SIF. In case of a write access, the data from the bus is routed to the addressed register. In case of a read access, the content of the addressed register is made visible on the bus. Accesses to SIF registers, where applicable, trigger further actions, for instance starting of the TX state machine or clearing of an interrupt.

External TX Interface. The external TX interface of the SIF enables the data transmission to a connected external peripheral module or another SIF. Data is transmitted actively through this interface, therefore it is specified as a master interface. If data transmission is enabled, the data, which is written from the core to the TXD register, is directly transmitted to the connected external module.

External RX Interface. The module which is connected to the external RX interface of the SIF, can actively send data to the SIF. Therefore, this interface is specified as slave. The SIF can only receive data through this interface if the connected external module transmits data and the reception is enabled. If the serial reception has finished, the received data is written to the RXD register. An interrupt can be scheduled to notify the CPU core that data is available. Hence, the core can read the content of the RXD register.

Interrupt Interface. The interrupt interface of the SIF model contains all available interrupts as outgoing ports. This interface acts as a master. The interrupt ports are connected to the Interrupt Control Unit (ICU) of the system. The ICU handles all peripheral interrupts and notifies the core to serve an interrupt. The core in turn initiates an interrupt service routine corresponding to the active interrupt.

5.5.3 Registers and Bit Fields

As mentioned before, registers are key elements in the context of HW/SW interfaces. Table 5.9 provides an overview of all registers of the SIF including some general register parameters. These parameters are defined as follows:

- **Offset:** Specifies the address of the register relative to the base address of the peripheral
- **Width:** Specifies the data width of the register in terms of bits
- **AddrUnit:** Specifies the addressable unit of the register in terms of bits (e.g., $AddrUnit = 8$ denotes that one address value addresses 8 bits of data)
- **MirrorSize:** Specifies the number of consecutive addresses a register can be accessed through

Register	Offset	Width	AddrUnit	MirrorSize
SIF_TXD_REG	4	32	8	1
SIF_CTRL_REG	8	32	8	1
SIF_MODE_REG	12	32	8	1
SIF_PRTC_CFG_REG	16	32	8	1
SIF_CLK_RATE_REG	20	32	8	1
SIF_IRQ_CLEAR_REG	24	32	8	1
SIF_RXD_REG	28	32	8	1
SIF_STAT_REG	32	32	8	1

Table 5.9. SIF register overview.

The TXD register and RXD register are used for data transmission and data reception. The CTRL register is used for enabling or disabling data transmission or reception of the SIF. Furthermore, mode and interrupt behavior can be configured here. The protocol behavior of the SIF can be configured by writing to the PRTC_CFG register. Activated interrupts can be cleared using the IRQ_CLEAR register. The core can retrieve the current status of the SIF by reading the content of the STAT register. The transmission and reception rate can be controlled with the CLK_RATE register.

A register can be seen as an alias for the offset address. The actual values are accessible through the bit fields it contains. The complete protocol configuration of the SIF, like parity settings, inversion, etc. happens through the

PRTC_CFG register. Therefore, this register needs to be structured into bit fields which are referring to specific protocol settings. A detailed description of the registers including their bit fields is given in the following sections.

Register SIF_TXD_REG. The TXD register is written by the core. Its content is transmitted through the external TX interface. Table 5.10 shows the bit fields of the TXD register including their parameters.

- **Offset:** specifies the offset of the bit field within the register in bits
- **Width:** specifies the width of the bit field in bits
- **AccessExt:** specifies the external access type of the bit field via the bus interface (e.g., from the core)—read only (R), write only (W), read and write (R/W)
- **AccessInt:** specifies the internal access type of the bit field from inside the SIF (e.g., from the state machines)

The TXD register contains only one bit field which covers the complete register width. It is writable and readable from the core, and only readable internally. Every time the core writes data to the register and the transmit bit field, respectively, the TXD state machine gets notified and transmits the data by accessing the content of the bit field internally.

BitField	Offset	Width	AccessExt	AccessInt
transmit	0	32	R/W	R

Table 5.10. Register SIF_TXD_REG bit fields overview.

Register SIF_CTRL_REG. The mode and control configuration of the SIF happens through the CTRL register which is structured into ten bit fields as shown in Table 5.11.

Transmission can be enabled or disabled by setting the *transmit_enable* bit field. In case the transmission of the SIF is disabled, the content available in the TXD register will not be sent. The value is stored until the transmission is enabled.

A similar behavior applies to the *receive_enable* bit field which is used to enable or disable the data reception of the SIF. In disabled state, any incoming data through the external RX interface is ignored.

Interrupts for successful or failed data transmission and reception, respectively, can be enabled or disabled by setting the *interrupt_on_** bit fields.

BitField	Offset	Width	AccessExt	AccessInt
interrupt_enable_on_transmit	0	1	R/W	R
interrupt_enable_on_transmit_error	1	1	R/W	R
transmit_enable	2	1	R/W	R
reserved0	3	1	R/W	R
interrupt_enable_on_receive	4	1	R/W	R
interrupt_enable_on_receive_error	5	1	R/W	R
receive_enable	6	1	R/W	R
reserved1	7	1	R/W	R

Table 5.11. Register SIF_CTRL_REG bit fields overview.

Register SIF_MODE_REG. The operation mode of the SIF can be configured by setting the *loop_back* and *echo_mode* bit fields within the MODE register. In loop back mode, the SIF loops the transmitted data back into the RX state machine. In echo mode, received data is transmitted instantly through the external TX interface. The bit field structure of the MODE register is shown in Table 5.12.

BitField	Offset	Width	AccessExt	AccessInt
loop_back	0	1	R/W	R
echo_mode	1	1	R/W	R

Table 5.12. Register SIF_MODE_REG bit fields overview.

Register SIF_PRTC_CFG_REG. The TX and RX protocol behavior of the SIF can be specified by writing the bit fields of the PRTC_CFG register. Table 5.13 gives an overview of the bit field specification of the PRTC_CFG register. The bit field *tx_stop_bit2* specifies whether a second stop bit should be appended to the data. Parity checking can be activated with *tx_enable_parity*, and the *tx_odd_parity* bit field specifies if an even or odd parity bit should be appended. The *tx_inversion* and *tx_heading* bit fields are used for data inversion and reversal of data heading, respectively. The SIF supports different character length values for transmission or reception—8 bit, 16 bit, and 32 bit. The TX character length can be specified with the *tx_character_length* bit field.

The same bit fields are available for the RX protocol with respective meanings.

Register SIF_CLK_RATE_REG. The settings of the CLK_RATE register define the baud rate for transmission and reception. The bit field specification

BitField	Offset	Width	AccessExt	AccessInt
tx_stop_bit2	0	1	R/W	R
tx_enable_parity	1	1	R/W	R
tx_odd_parity	2	1	R/W	R
tx_heading	3	1	R/W	R
tx_inversion	4	1	R/W	R
tx_char_length	5	6	R/W	R
reserved0	11	5	R/W	R
rx_stop_bit2	16	1	R/W	R
rx_enable_parity	17	1	R/W	R
rx_odd_parity	18	1	R/W	R
rx_heading	19	1	R/W	R
rx_inversion	20	1	R/W	R
rx_char_length	21	6	R/W	R
reserved1	27	5	R/W	R

Table 5.13. Register SIF_PRTC_CFG_ REG bit fields overview.

is shown in Table 5.14. The clock rate for the TX data path can be specified within the *tx_clock_rate* bit field and the rate for the RX path within the *rx_clock_rate* bit field. The value of these bit fields is interpreted as a multiplier to the bus clock period. The baud rate defines the speed at which the serial bits are shifted out or read in, respectively.

BitField	Offset	Width	AccessExt	AccessInt
tx_clock_rate	0	16	R/W	R
rx_clock_rate	16	16	R/W	R

Table 5.14. Register SIF_CLK_RATE_ REG bit fields overview.

Register SIF_IRQ_CLEAR_REG. Active interrupts of the SIF can be cleared by writing the corresponding interrupt ID into the *irq_clear* bit field within the IRQ_CLEAR register. The register specification is shown in Table 5.15.

BitField	Offset	Width	AccessExt	AccessInt
irq_clear	0	4	R/W	R

Table 5.15. Register SIF_IRQ_ CLEAR_REG bit fields overview.

Register SIF_RXD_REG. The bit field specification of the RXD register is shown in Table 5.16. The RXD register contains only one bit field with the same width as the register itself. This *receive* bit field is declared as external read only. If the SIF receives data through its external RX interface, the core can read the received data by accessing this bit field.

BitField	Offset	Width	AccessExt	AccessInt
receive	0	32	R	W

Table 5.16. Register SIF_RXD_REG bit fields overview.

Register SIF_STAT_REG. The access type of all bit fields within the STAT register is specified as external read-only as it is shown in Table 5.17. The core can access status information of the current state of the SIF by reading the bit fields of this register. The bit fields *data_transmitted* and *data_received* contain information about a successful data transmission and reception, respectively. The bit fields *data_transmit_error* and *data_receive_error* contain information about a failed data transmission and reception, respectively.

BitField	Offset	Width	AccessExt	AccessInt
data_transmitted	0	1	R	R/W
data_received	1	1	R	R/W
data_transmit_error	2	1	R	R/W
data_receive_error	3	1	R	R/W

Table 5.17. Register SIF_STAT_REG bit fields overview.

5.6 Register Header File

A C header file, as explained in the beginning, is generated to enable an easy software access to the registers and bit fields of the SIF. The template based generation framework is described in Sect. 5.9. The generated header file contains a struct and a union for each register access function. A struct of the complete address space of the SIF is also generated, which is used by the access functions. The mechanism of the register access is explained in detail in the following.

5.6.1 Register Bit Field Structure

A struct is declared for each register which represents its corresponding bit field structure. Hereby the complete width of a register is divided into bit fields. Listing 5.29 shows an example of the structs for the *SIF_TXD_REG* and *SIF_*

STAT_REG registers. The *SIF_TXD_REG* register only contains the *transmit* bit field. Since this bit field has width 32 bit the struct *sSIF_TXD_REGStructure* contains only one member referring to the *transmit* bit field. The declaration of a bit field member within the register struct is shown in following rule:

["const"] *data_type* [*bitfield_name*] " : " *bitfield_width* "; "

A bit field gets declared as *const* if it is specified as externally read-only. The *data_type* refers to the *data_type* of the register. The *bitfield_name* is optional and refers to the specified name of the corresponding bit field. The *bitfield_width* refers to the width of the bit field. If no bit field is specified for a specific area of a register, this area must be marked as unused. This is done by declaring a *const data_type* without a name and the width of the unused area. The declaration order of the bit field members within the struct is given by the specified bit offset of the bit field. The *SIF_STAT_REG* register contains four bit fields which are declared as members within the struct *sSIF_STAT_REGStructure*. All of these bit fields are specified as read-only, therefore they are declared as *const*. The unused area of the bit field is declared as previously explained.

```
// SIF_TXD_REG
typedef struct {
  uint32_t transmit                     :32;
} sSIF_TXD_REGStructure;

// SIF_STAT_REG
typedef struct {
  const uint32_t data_transmitted       :1;
  const uint32_t data_received          :1;
  const uint32_t data_transmit_error    :1;
  const uint32_t data_receive_error     :1;
  const uint32_t                        :28; /* unused area */
} sSIF_STAT_REGStructure;
```

Listing 5.29. Register bit field structure.

5.6.2 Register Union

A union is generated for each register which contains an entry referring to the register value and an entry for the register structure. Listing 5.30 shows an example declaration of the *uSIF_TXD_REG* union for the *SIF_TXD_REG* register. The value of the complete register is accessible through *SIF_TXD_REG_Content* which is declared using the data type of the register. The values of the bit fields are accessible through *SIF_TXD_REG_Structure* which is declared using the previously described bit field struct *sSIF_TXD_REGStructure* as data type. The register unions are used as register types in the following described module struct.

```
// SIF_TXD_REG
typedef union {
  uint32_t SIF_TXD_REG_Content;
  sSIF_TXD_REGStructure SIF_TXD_REG_Structure;
} uSIF_TXD_REG;
```

Listing 5.30. Register union declaration.

5.6.3 Module Structure

A declaration of a struct is needed which represents the complete register address range of the module. This struct contains all registers which are ordered referring to their specified offset. Listing 5.31 shows the module structure *_sSif* of the SIF module. The previously described register unions are used as data type for each register member. If an unused address area exists where no register is specified, the area has to be marked as a reserved area. In case of the SIF, no register is specified for offset “0”. Therefore, the first entry of the *_sSif* refers to a reserved area and is declared as *const* using *uint32_t* as data type and the name *reservedArea#*. The # represents a number starting with ”0” which is incremented for each reserved area declaration. With the value in brackets, which represents the array size, it is specified how many sub sequential addresses should be marked as reserved. Subsequent to the struct declaration, the type definition *sSif* of the *_sSif* struct is declared. The *sSif* type is used within the following described register and bit field access functions.

```
struct _sSif {
  const uint32_t reservedArea0 [1];              // Address offset = 0x0
  uSIF_TXD_REG SIF_TXD_REG;                      // Address offset = 0x4
  uSIF_CTRL_REG SIF_CTRL_REG;                    // Address offset = 0x8
  uSIF_MODE_REG SIF_MODE_REG;                    // Address offset = 0xc
  uSIF_PRTC_CFG_REG SIF_PRTC_CFG_REG;            // Address offset = 0x10
  uSIF_CLK_RATE_REG SIF_CLK_RATE_REG;            // Address offset = 0x14
  uSIF_IRQ_CLEAR_REG SIF_IRQ_CLEAR_REG;          // Address offset = 0x18
  uSIF_RXD_REG SIF_RXD_REG;                      // Address offset = 0x1c
  uSIF_STAT_REG SIF_STAT_REG;                    // Address offset = 0x20
};

typedef struct _sSif sSif;
```

Listing 5.31. Component register structure.

5.6.4 Register Access Functions

The implementation of the functions for accessing the content of a complete register are described now. Depending on the external access type of a register, a *set* (for writing) and *get* (for reading) function are generated. The external access type of a register is obtained using the bit field information. If a register contains only bit fields which are read-only from external, then the access type of the register is also read-only. In this case, only a *get* access function is generated. Listing 5.32 shows the register access functions for the *SIF_TXD_REG* register. The *transmit* bit field within this register is specified as external readable and writable. Therefore, both a *get* and a *set* function are generated as shown in the listing. The *set* function has no return value and has in its formal argument list the *sSif* pointer *_sif_* and the value which should be written to the register. The target SIF instance is specified using the *_sif_* argument which refers to the base address of the SIF. The corresponding register is accessed by using the −> operator on the *_sif_* pointer of the struct member *SIF_TXD_REG*. This means the base address plus the address offset of the register is obtained because the position of the *SIF_TXD_REG* member refers to the offset address of the register. The register union *uSIF_TXD_REG* is used as the data type for the *SIF_TXD_REG* member. Therefore, the value has to be assigned to the *SIF_TXD_REG_Content* member of *SIF_TXD_REG*. The *get* function is implemented in a similar way. It has the register value as a return value and has no value in its argument list. The implementation of the *get* function returns the value of *SIF_TXD_REG_Content*.

```
/* Set complete register SIF_TXD_REG */
static inline void setSif_SIF_TXD_REG(volatile sSif *_sif_,
    uint32_t value) {
  _sif_->SIF_TXD_REG.SIF_TXD_REG_Content = value;
}

/* Get complete register SIF_TXD_REG */
static inline uint32_t getSif_SIF_TXD_REG(volatile sSif *_sif_) {
  return _sif_->SIF_TXD_REG.SIF_TXD_REG_Content;
}
```

Listing 5.32. Register read and write access functions.

5.6.5 Bit Field Access Functions

After the generation of the access functions for the registers, the access functions for the bit fields are generated. Depending on the external access type of a bit field, a *set* and *get* function has to be implemented. Listing 5.33 shows the implementation of the access functions for bit field *transmit* within the *SIF_TXD_REG* register. The argument lists and return values are equal to the register access functions. The difference is located in the implementation of the bit field *set* and *get* functions. In case of the bit field *set* function, the value is assigned to the *transmit* member of *SIF_TXD_ REG_Structure* of register union *uSIF_TXD_REG*. Within the register union, the bit field structure *sSIF_TXD_REGStructure* is used as data type for *SIF_TXD_REG_Structure*. Hence the register union enables the access to the complete register content, or to a specific bit field of the register.

```
/* Set element transmit of register SIF_TXD_REG */
static inline void setSif_SIF_TXD_REG_transmit(volatile sSif *
    _sif_, uint32_t value) {
  _sif_->SIF_TXD_REG.SIF_TXD_REG_Structure.transmit = value;
}

/* Get element transmit of register SIF_TXD_REG */
static inline uint32_t getSif_SIF_TXD_REG_transmit(volatile
    sSif *_sif_) {
  return (uint32_t)_sif_->
      SIF_TXD_REG.SIF_TXD_REG_Structure.transmit;
}
```

Listing 5.33. Bit field read and write access functions.

5.7 SIF Driver Functions

The register and bit field access functions which were described in the previous section are not directly used in a software application since they are too low-level. Driver functions have to be implemented to enable a higher-level software access to the hardware module. These driver functions make use of the lower-level functions offered by the register header file. Unlike the register header file, the driver functions cannot be generated using a meta-model that only describes interfaces but no functionality. A part of the driver header file including the most important driver functions is shown in Listing 5.34. The generated register header file *sif_reg.h* is included in the driver header file *sif_drv.h*, which is shown in the listing. The implementation of the *sifOpen*, *sifInit*, *sifWrite* and *sifRead* functions is explained in detail in the following. Other functions like *sifIOCtl*, *sifClose*, *sifSelfTest*, etc. are not explained within the context of this section. One possibility for realizing software access to the SIF module is represented by the following driver functions. Depending on the operating system and the software environment, the implementation of the drivers may be different.

5.7.1 SIF Open Function

The *sifOpen* function, which returns the *sSif* pointer of the specified SIF module, is shown in Listing 5.35. The *sifOpen* function contains in its formal argument list the unsigned integer *id*, whose value corresponds to a specific SIF module of the hardware system. In case of the example which is shown in

```
// include register header file
#include "sif_reg.h"

// open specified sif
volatile sSif* sifOpen(unsigned id);
// initialize sif
void sifInit(volatile sSif *p_sif);
// write block to sif
void sifWrite(volatile sSif *p_sif, const char *string);
// read block from sif
void sifRead(volatile sSif *p_sif, char* buffer);
...
```

Listing 5.34. SIF driver header file.

the listing, the hardware system contains two SIF modules, one with the base address 0x20000000, and another with the base address 0x30000000. Depending on the value of *id*, the base address of the corresponding SIF module is assigned to the *sSif* pointer and returned.

```
volatile sSif* sifOpen(unsigned id) {
  volatile sSif *pSIF;
  switch(id) {
    case 0:
      pSIF = (sSif*)0x20000000; break;
    case 1:
      pSIF = (sSif*)0x30000000; break;
    default:
      pSIF = 0; break;
  }
  return pSIF;
}
```

Listing 5.35. SIF open function.

5.7.2 SIF Init Function

After a specific SIF has been opened using the *sifOpen* function, it has to get initialized. This is done by calling the *sifInit* function using the returned *sSif* pointer as the argument. The implementation of the *sifInit* function is shown in Listing 5.36. The specified SIF module gets initialized with a default control and protocol configuration. The low-level bit field access functions of the register header file are used for configuring the *SIF_CTRL_REG* and *SIF_PRTC_CFG_REG* registers. First, the SIF is enabled for data transmission and reception, then the character length of both the TX and RX data are set to 32 bits. Not all configurations are shown in the listing. After the SIF module is opened and initialized, the application can write data to the SIF and read data from it. The necessary driver functions for the data transfer are described in the following.

```
void sifInit(volatile sSif *p_sif) {
  // set default rx and tx control
  setSif_SIF_CTRL_REG_transmit_enable(p_sif, 0x1);
  setSif_SIF_CTRL_REG_receive_enable(p_sif, 0x1);
  ...
  // set default rx and tx protocol
  setSif_SIF_PRTC_CFG_REG_tx_char_length(p_sif, 0x20);
  setSif_SIF_PRTC_CFG_REG_rx_char_length(p_sif, 0x20);
  ...
}
```

Listing 5.36. SIF initialization function.

5.7.3 SIF Write Function

The *sifWriteChar* function, which is shown in Listing 5.37, is used within the *sifWrite* function to write one character to the SIF. The argument list of the *sifWriteChar* function contains the *sSif* pointer and the value to be written. The function polls the value of the *data_transmitted* bit field within the *SIF_STAT_REG* register, until it is "0", which means that the previous data has been transmitted and a new value can be written. In that case, the *set* function for the *SIF_TXD_REG* register is called and the value gets written to the SIF. The *sifWriteChar* function is not accessible by the application software, since the interaction of the application software with the SIF is realized in data blocks.

```
void sifWriteChar(volatile sSif *p_sif, uint32_t value){
  while(!getSif_SIF_STAT_REG_data_transmitted(p_sif))
  ;
  setSif_SIF_TXD_REG(p_sif, value);
}
```

Listing 5.37. SIF write data word function.

The application software uses the *sifWrite* function for data transmission which is shown in Listing 5.38. This function defines in its argument list the character pointer *string*. Hence, the application software can write complete strings to the SIF module. The implementation of the *sifWrite* function serializes the given string into single data words depending on the specified *tx_char_length*, and sends each data word to the SIF by using the previously described *sifWriteChar* function. The implementation of the *sifWrite* and the following *sifRead* functions are simplified for explanation purposes.

```
void sifWrite(volatile sSif *p_sif, const char *string) {
  uint32_t ch_length=getSif_SIF_PRTC_CFG_REG_tx_char_length(
      p_sif);
  uint32_t pkg_length, tx_data;
  while(*string) {
    pkg_length = 0;
    tx_data = 0;
    while((pkg_length < ch_length) && *string) {
      tx_data = tx_data | (*string++ << pkg_length);
      pkg_length = pkg_length + 8;
    }
    sifWriteChar(p_sif, tx_data);
  }
}
```

Listing 5.38. SIF write block function.

5.7.4 SIF Read Function

The data interface for the application software to the SIF is implemented in the *sifRead* function, as shown in Listing 5.39. This function reads a data word from the SIF, separates it into single characters depending on the *rx_char_length* value, and appends the characters to the *string* array.

```
void sifRead(volatile sSif *p_sif, char* string) {
 uint32_t ch_length=getSif_SIF_PRTC_CFG_REG_rx_char_length(
     p_sif);
 uint32_t rx_data, pkg_length;

 while(getSif_SIF_STAT_REG_data_received(p_sif)) {
    rx_data = getSif_SIF_RXD_REG(p_sif);
    pkg_length = 0;

    while(pkg_length < ch_length) {
      *string++ = (rx_data & (255 << pkg_length)) >>
          pkg_length;
      pkg_length = pkg_length + 8;
    }
  }
}
```

Listing 5.39. SIF read block function.

5.7.5 Test Software Application

Listing 5.40 shows a small test application which demonstrates the interaction of the application software with two SIF modules. This test application opens the first SIF (ID $= 0$), initializes it, writes a “Hello World!” string to it, and reads the string again (the TBE connected to the external interfaces of the SIF receives all data and transmits it back to the SIF). The same is done for the second SIF (ID = 1) within a loop. This small application shows that all low-level details of the HW/SW interface are hidden from the application software. Hence a developer of application software does not need to know all hardware specific details of a specific peripheral.

```
#include "sif_drv.h"

int main() {
  unsigned id = 0;
  volatile sSif *pSIF;

  do {
    char buffer[1024];
    pSIF = sifOpen(id++);
    sifInit(pSIF);
    sifWrite(pSIF, "Hello␣World!");
    sifRead(pSIF, buffer);
  }
  while(id < 2);

  return 1;
}
```

Listing 5.40. SIF test application.

5.8 Synchronization

This section provides an overview on concepts which together form the synchronization of HW/SW interfaces.

5.8.1 Register-Access Synchronization Schemes

Clock domains and synchronization. At the HW level, the settings of the bus and the master and slave interfaces ensure that the accesses of the core to peripherals are synchronized in terms of clock frequencies. It is possible that a core runs with a different clock than a peripheral. In such a case, it is common to use clock divider circuits and data buffers in the HW for synchronizing the data flow appropriately. Usually, the SW needs only to take care of the configuration of these components to maximize throughput.

Blocking vs. non-blocking bus protocols. As explained in the previous sections, SW accesses the HW via pointers (i.e., addresses). An access to an address is in turn broken down in the HW to read and write transactions. The duration of each access can vary depending on the underlying bus protocol. Such accesses can be categorized in the following way:

- Non-Blocking: The duration of the access is a priori defined; the duration of read transactions may differ from write transactions, however.
- Blocking: The duration of the access can dynamically vary, depending on the current bus load and peripheral activity.

These categories need to be taken into account when modeling peripheral drivers, because they can have a huge impact on the overall performance of a system.

5.8.2 Functional Synchronization Schemes

Polling. The easiest way to functionally synchronize SW behavior to the behavior of a peripheral is to use polling. For instance, the SIF peripheral offers a status register which yields whether the SIF is ready to transmit data or respectively has data available to be fetched by the core. The SW can retrieve the status of the SIF by accessing the corresponding register. After initiating one transmission, the SW cannot know when the SIF is ready to transmit further data. Therefore, one possible way to resolve this, is to read the status register periodically. As soon as the read value indicates that the SIF is ready for another transmission, the SW can stop reading the status register and proceed with the next value to be sent. Listing 5.41 shows an example.

```
uint32_t txd_array[4] = {10,20,30,40};
for(int i = 0; i < 4; i++){
  w_reg_ptr->txd_reg = txd_array[i];
  while !(r_reg_ptr->flag_reg.data_transmitted)
  ;
}
```

Listing 5.41. Polling of SIF status.

Yet being a simple solution, polling is not very efficient with regard to performance, due to the periodic read access to the SIF status register. The next section shows how interrupts can be used in order to notify the SW by the peripheral, as soon as it is ready to transmit further data.

Interrupt Handling. Interrupt-based synchronization of SW with HW is one of the dominant schemes used in embedded systems design. Herein the basic principles of interrupt handling are explained. Most CPU cores in industrial use provide interrupt mechanisms in order to decouple SW from the state of a peripheral. The interrupt lines of each peripheral are connected to the so called Interrupt Control Unit (ICU). By setting an interrupt line, a peripheral indicates that it requests to interrupt the CPU core execution. Since it is possible that several peripherals can issue interrupt requests at the same time, the

ICU implements some sort of interrupt prioritization scheme and notifies the CPU core in turn by raising an interrupt to the CPU core. The HW-based interrupt infrastructure of a core can vary. It has to take into account when exactly to halt the current execution of a program and to save its context, in order to serve a specific interrupt request. An ICU usually contains registers for each interrupt input. A programmer can configure these registers by storing the address of a specific interrupt service routine in each of these registers. Once the CPU receives an interrupt from the ICU, it can retrieve the previously stored address to the corresponding service routine and execute it. Such a service routine, among other things, needs to take care of clearing the specific interrupt in the peripheral, which had raised it. Once the interrupt service routine has been executed by the CPU core, the execution of the previously halted program resumes. For that purpose, the CPU restores the program context automatically. In the following example, it is assumed that the ICU interrupt input of number 42 is connected to the *data_transmitted* interrupt line of the SIF. Furthermore, it is assumed that the general setup of the ICU has been taken care of. Listing 5.42 shows a simple code example, which contains an interrupt service routine. This routine is invoked whenever the *data_transmitted* interrupt is raised. It takes care of sending a further data value to the TXD register.

The function *enable_IRQ* sets up the CPU internal interrupt infrastructure. For that purpose, some assembly language code (omitted in the example) needs to be inlined. The *SIF_TXREQ* function is the actual interrupt service routine. It ensures that the corresponding ICU interrupt line is enabled again by writing value 42 to specific ICU register addressed by *ICU_REENABLE_HW_LINE*. Furthermore, the function clears the raised interrupt request in the SIF peripheral by writing to the interrupt clear register addressed by *SIF_CLEAR_REG*. Following that, one value is written to the TXD register of the SIF. Within function *send_data*, the ICU register, which is associated with port 42, is written with the address of the interrupt service routine. Hence, when line 42 shows an interrupt, this service routine will be called. Afterwards, the IRQ infrastructure is enabled. Following that, a while loop is entered, which can only be left if the service routine has been called 4 times. Note, that instead of the while loop, the SW could perform any other actions, because the data transmission is handled by the interrupt service routine.

5.9 Template Based Code Generation

Before the implementation of an HW/SW system starts, a specification has to be created which is normally provided as a textual description and not in a formal way. This often leads to implementation inconsistencies or misunderstandings of the specification. Due to the increasing complexity of HW systems formal methods for specifying interface information were developed.

```
__inline void enable_IRQ(void) {
  int tmp;
  __asm {
    // core dependent asm code
    // for enabling IRQs
  }
}

volatile int i;
uint32_t txd_array[4] = {10,20,30,40};

void __irq SIF_TXREQ(void) {
  *ICU_REENABLE_HW_LINE = 42;
  *SIF_CLEAR_REG = 0xF;
  if(4 != i) {
    w_reg_ptr->txd_reg = txd_array[i++];
  }
}

void send_data() {
  i = 0;
  // setup ICU Register for port 42
  *ICU_CBPORT_42 = (volatile unsigned)SIF_TXREQ;
  enable_IRQ();
  while(i < 4)
  ;
  printf("Data␣Transmitted");
}
```

Listing 5.42. Interrupt based transmission.

On the one hand, these formal descriptions can be used for IP reuse, on the other hand, they can be used for the generation of consistent hardware and HW/SW interfaces. The following sections introduce a UML based meta model and a generated API which supports easy access to the specification data, followed by a technique to import XML based textual specifications into the meta model. In connection to that, a template-based generation framework is described which enables a flexible use of the meta model for code generation.

5.9.1 UML Meta Model and its API

The basis of the generation framework is a UML based meta model which contains the information about how to model the hardware interfaces, like the bus interface and the register and bit field information referring to the HW/SW interface. The specification of each register and bit field of a hardware module would be tedious using UML. Hence, a UML meta model is developed containing all interface and top-level mapping data. Most UML tools provide source code generation but only for a few target languages. Due to the variety of target languages (SystemC, SystemVerilog, C, etc.), a flexible UML meta model and generation framework is required. The existing standard IP-XACT [SPI], which is based on XML, targets IP reuse and IP exchange by focusing on the packaging of IP. But since we also need to incorporate more functional aspects in order to support IP generation (e.g., generate stubs for the IP development) as well, we defined our own data model instead. In order to fully leverage the benefits of IP-XACT for automating third party IP integration we developed import and export filters for IP-XACT as well.

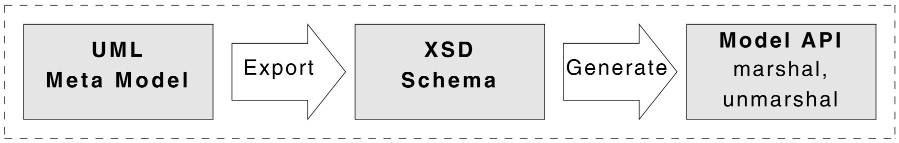

Figure 5.7. Meta model API generation.

Figure 5.7 shows the generation of the API for accessing the data of the meta model. The meta model itself is developed using a common UML tool and exported as an XSD schema. From this schema a class library is generated which offers marshaling and unmarshaling of XML meta model data and provides `set`, `get`, and `add` functions for the elements.

5.9.2 Specification Import

The previously described API for the meta model supports marshaling to create an XSD schema compliant XML file. Hence, a tool is developed which supports an easy import of textual specifications that were created compliant to company specific guidelines. Figure 5.8 gives an overview of the tool which converts textual specifications using the API. A Python [Pyt] conversion plug-in and a textual specification—written with an editor and saved as XML—are used as input while the meta model compliant XML is the output.

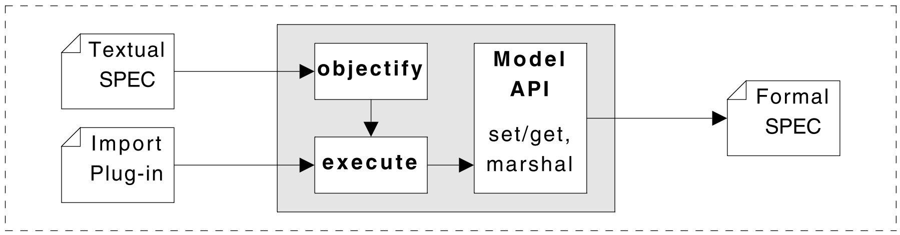

Figure 5.8. Specification import mechanism.

Within the tool the XML based textual specification is objectified using the Python Gnosis library. The obtained object structure is iterated with the conversion plug-in and its values and elements are passed to the meta model API. At the end the object created by the API is marshaled to a formal XML specification. At the moment many different XML methodologies exist within a single company which can be imported to support a company wide XML specification.

5.9.3 Template Based Code Generator

A flexible generation methodology has been developed which provides the code generation for different target applications, for instance firmware header files or the register interface of RTL or TLM models. This is achieved by linking the meta model API to a template engine to access the data of the XML from the template. Templates allow the separation of model and view, in our case the data provided by the XML and the target code to be generated. We use the Python MAKO template engine [Mak] which offers a template language for conditional branches, loops, and hierarchical templates. Furthermore, the complete Python scripting functionality can be embedded within a template. A template can be composed of hierarchical templates, hence, it is possible to reuse so called sub-templates for the generation of different target applications.

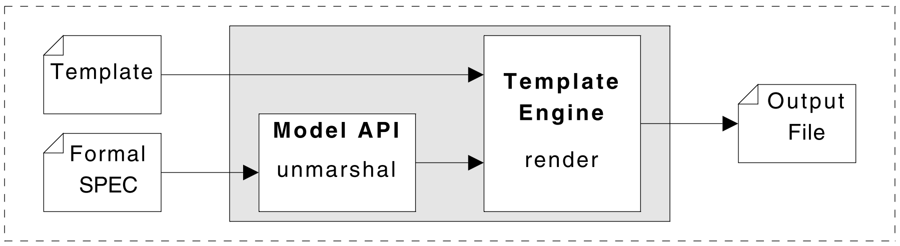

Figure 5.9. Template based code generator.

Figure 5.9 gives an overview of the template generator tool. The template gets rendered by the MAKO template engine using the data provided within the XML file.

5.9.4 HW/SW Interface Generation

The HW/SW interface is described by the registers of a hardware module and by the connection of the CPU core through the bus to the module. Hence, the meta model must contain both the register and bit field information as well as the interconnect information of the module and the CPU core. In the following an overview of the generation of software header files and the register interface of a module is given.

Register header files offer low-level access functions to the memory mapped registers of a hardware module. A register within a hardware module is addressable by using the base address of the module and the corresponding register offset. Each register itself is divided into one or more bit fields. A bit field is described by a bit offset, bit width and an access type which specifies if a bit field is readable, writable, or both.

A hierarchical template structure was developed for the generation of a register header file as it is shown in Fig. 5.10.

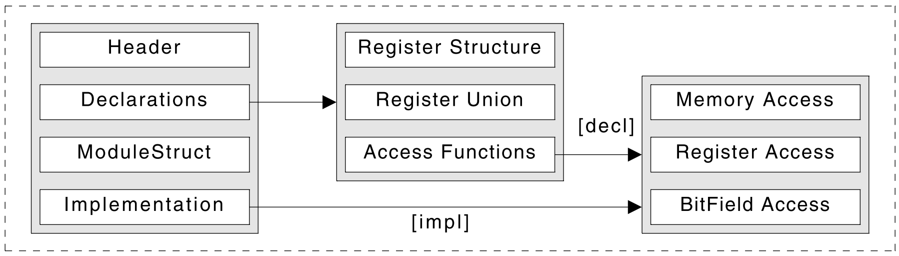

Figure 5.10. Hierarchical register header template.

The main template includes four sub-templates. The *Header* sub-template generates register header file specific information like version, name, module description, and so on. Furthermore, required type definitions are generated by this sub-template.

The *Declaration* sub-template iterates over the registers of a module and generates the register bit field structure and a register union which provides the access to the bit field structure and the value of the registers. The declaration of the register and bit field access functions is also generated here.

The *ModuleStruct* sub-template generates a representation of the address space of the hardware module using the register offsets, the register data width, and the addressable unit of the bus interface.

The *Implementation* sub-template generates the implementation of the access functions.

Following the same methodology the RTL or TLM register interfaces are generated in a consistent way.

5.10 Modeling the HW/SW Interface

The previous sections focus only on the software side of the HW/SW interface. This section describes the modeling of a abstracted hardware module of the SIF module using transaction level modeling in SystemC. First, a introduction to transaction level modeling is given. After that it is shown how the SIF is integrated into a hardware system, including the description of its interface implementations. At the end of this section a methodology for modeling high performance simulation models is introduced, which enables complex SW tests in an early design stage.

5.10.1 Transaction Level Modeling

Transaction level modeling plays a major role in the success of the development of so called virtual prototypes (VP) which represent abstract models of HW platforms. This allows breaking down a system to a set of components or blocks (representing the actual architecture of the platform top level) comprised of concurrent processes. These blocks communicate with each other via so-called transactions. A transaction represents a high-level form of a communication protocol. All protocol-specific details are encapsulated within a transaction. Hence, the actual act of initiating a transaction results in a remote function call from within a process (parent). A designer focuses more or less on the data that has to be transported rather than the protocol specifics.

In SystemC, the most established modeling language for TLMs, transactions are modeled as functions which are defined in pure virtual interface classes and are implemented in corresponding child classes which inherit from these interfaces. The implementation details of a transaction strongly depend on the targeted abstraction level. Yet two distinctions with regard to transactions can be made:

- Blocking: A blocking transaction may suspend its parent process which means that the transaction is resumed in a later delta-cycle. This kind of transaction can be invoked in suspendable SystemC processes, only (i.e., `SC_THREAD`).

- Non-Blocking: A non-blocking transaction is atomic and may not suspend its parent process; the whole transaction is executed within the same delta-cycle it has been invoked. This kind of transaction can be called from within any SystemC process (i.e., `SC_THREAD` and `SC_METHOD`).

Invoking a transaction results in dereferencing a pointer that holds the address of the target object and in calling a member function of that object. The whole call or even several calls can happen within a single delta-cycle (e.g., with non-blocking transactions). In contrast to that, communication in RTL models is obtained via signals and hence, always consumes at least one delta-cycle due to the induced value-changes that form the protocol. In order to provide connection semantics for transactions as well, SystemC provides a port concept. Figure 5.11 shows a graphical representation.

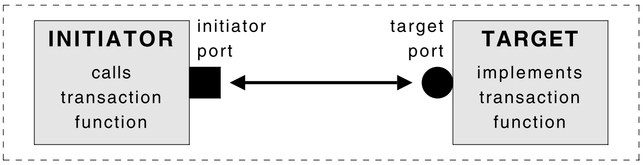

Figure 5.11. TLM interface.

Transactions are called over initiator ports. Transactions are provided by target ports. Initiator ports are modeled in SystemC using `sc_port` and target ports using `sc_export`. Both these ports expect a template argument which holds the interface definition, i.e., the pure virtual class which defines the signatures of all transactions accessible through this interface. In order to ensure easy IP reuse and interoperability, a transaction level modeling standard has been developed by the Open SystemC Initiative. This standard defines different interface classes including transaction signatures and argument types. These interfaces are organized in terms of their characterizations, i.e., into blocking or non-blocking interfaces, and the flow of data, i.e., unidirectional or bi-directional. A module which contains a target port has to provide an implementation for the transaction defined in the interface class which was given as template argument to this target port. In the following sections the so-called transport interface (see Listing 5.43) from the TLM standard is used.

```
// bidirectional blocking interfaces
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_core::sc_interface {
public:
  virtual RSP transport( const REQ & ) = 0;

  virtual void transport( const REQ &req , RSP &rsp ) {
    rsp = transport( req );
  }
};
```

Listing 5.43. Blocking transport interface class.

This interface is a bidirectional blocking interface. It defines a function called “transport” which in turn is templated with two abstract data types (`REQ`, `RSP`)—one which holds the information on a specific bus request, e.g., target address and data, and one holding the response or rather the result of the transaction. The user needs to define request and response classes and needs to provide these as template arguments to customize the transport interface. Both initiator and target port need to use the very same interface, and thus also the same classes for request and response in order to be connected.

5.10.2 SIF Transaction Level Model

Figure 5.12 shows the connections of the SIF to a bus of a hardware system. This system includes a CPU core, a RAM and the SIF, which is externally connected to a test-bench element (TBE). In the following the implementation of the transaction level bus interface and the external RX/TX interface of the SIF is explained.

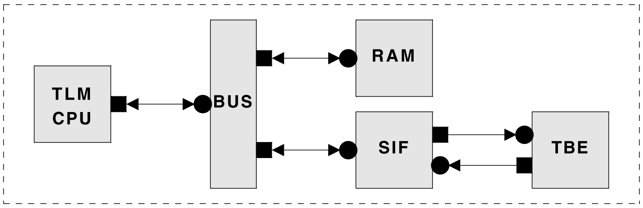

Figure 5.12. TLM system.

Bus Interface. The bus interface of the SIF is specified as a blocking transport interface of OSCI TLM 1.0 [OSC]. It is a bi-directional interface which enables read and write access to a module. The initiator port (*sc_port*) of the bus is mapped to the target port (*sc_export*) of the SIF. The target implements the *transport* interface function. Listing 5.44 shows the implementation of the bus interface of the SIF. First, the declarations of the *bus_request* and *bus_response* classes are shown. The *bus_request* class contains *addr*, *data*, and *access* members. The enumeration type *access_type*, which contains the values *READ* and *WRITE*, is used to indicate the type of the bus access. The *bus_response* class includes a *data* member which contains the requested data in case of a read access and the integer variable *status*. The port *bus_port* of type *sc_export* which is templated with the transport interface using the *bus_request* and *bus_response* classes.

SIF RX/TX Interfaces. The non-blocking put interface is used for the external RX/TX interfaces of the SIF. This interface enables a uni-directional transfer of the external payload. Listing 5.45 shows the implementation of the external interfaces. First, the external payload class *ext_payload* is declared. It contains the data, the start and stop bits, and the parity information. The external RX interface is declared as a non-blocking put *sc_export* target port. Therefore, the *nb_put* transaction function has to be implemented within the SIF. The external TX interface is implemented as a non-blocking put *sc_port* initiator port. Both non-blocking put interfaces are templated with the *ext_payload* class.

```
#include <stdint.h>

// bus protocol
enum access_type {READ, WRITE};

class bus_request {
public:
   access_type access;
   uint32_t addr;
   uint32_t data;
};

class bus_response {
public:
   int status;
   uint32_t data;
};

// SIF bus interface port
sc_export< tlm_transport_if<bus_request, bus_response> >
    bus_port;
```

Listing 5.44. Blocking transport bus interface.

```
#include <stdint.h>

// external interface payload
class ext_payload {
public:
   unit32_t data;
   unsigned char_length;
   bool start_bit, stop_bit, stop_bit2, has_stop_bit2,
        has_parity, odd_parity_bit;
};

// external RX interface
sc_export<tlm_nonblocking_put_if<ext_payload> > rxd_port;
// external TX interface
sc_port<tlm_nonblocking_put_if<ext_payload> > txd_port;
```

Listing 5.45. External non-blocking put interface.

5.10.3 Data Flow Abstraction

Fast simulation models of the hardware are required for the development of complex application software. Current transaction level simulation models mostly do not meet this requirement. Therefore, new methodologies need to be developed to close this gap. The first step in this direction is the abstraction of the data flow from the software to the hardware, and vice versa. Figure 5.13

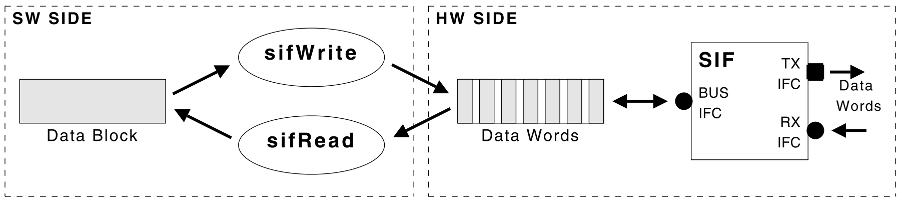

Figure 5.13. HW/SW data flow.

shows the data flow of the previously described SIF module and the application software.

As it is shown in the figure, the software side deals with data blocks, and the hardware side with single data words. In case of a write access to the SIF, the data block gets divided into sequential data words by the *sifWrite* function. In case of a read access, the data words are put together into a data block. The basic concept of the data flow abstraction is explained in the following.

Basic Concept. As it was previously explained, the software side deals with data blocks, but the hardware side with data words. The *sifWrite* and *sifRead* driver functions act like a transactor in between which converts a block to words and vice versa. Not only the conversion costs simulation performance, but also the bus accesses for each data word. The solution for this problem is the abstraction of the data flow, like it is shown in Fig. 5.14.

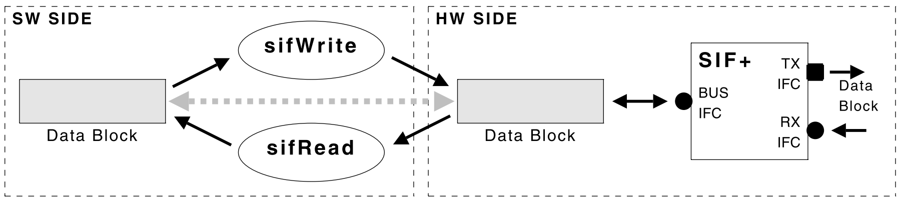

Figure 5.14. Abstracted HW/SW data flow.

As it is shown in the figure, the concept is that the data blocks are not converted by the driver function to data words, but passed directly to the extended SIF module, which is referred to as SIF+ module in the following. This means that software and hardware functionality are not considered separately, but in a common view. In this common view, both the software and the hardware side are dealing with data blocks. Some requirements need to be fulfilled before the common view of hardware and software can be realized.

Requirements. The first requirement for merging hardware and software functionality is that both the software and the hardware are executed on the CPU of the host simulation system. That means that the software cannot be instruction set simulated on the TLM CPU core, but must be emulated on the simulation host. This can be solved by replacing the TLM CPU core of the hardware system with a SystemC module, which wraps the C++ class members referring to the TLM bus interface to C, and executes the C software. As it is shown in Fig. 5.15 the TLM CPU core was replaced by the *EMUCPU* SystemC module and the *RAM* module is no longer part of the system.

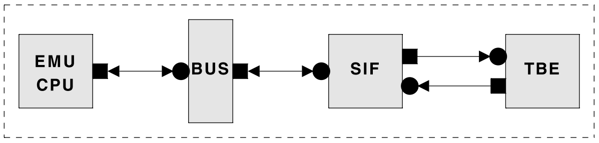

Figure 5.15. Host emulated system.

Now the implementation of the register and bit field access functions within the register header file directly accesses the TLM bus interface. The template, which generates the access function, can be reused by adding an argument which specifies whether a header file should be generated for an instruction set simulated or a host system emulated environment.

This leads to the next requirement to assure the consistency of the TLM model and the abstracted model. The template-based generator helps to achieve this requirement, since the existing templates can be reused for the generation of the abstracted model; the normal TLM interfaces are still used for the control flow.

The last requirement is that both the bus interface and the external interfaces of the SIF module need to be abstracted to support block data transfer. The abstraction of the interfaces is described in the following.

Interface Abstraction. Both, the bus interface and the external RX and TX interface need to be extended to support block transfer. In addition to the bus interface, the SIF gets extended by an abstract interface which enables read and write block transfers. The control flow of the SIF still happens through the TLM bus interface, but the abstracted data flow is realized using the abstract interface. An overview of the interfaces is given in Fig. 5.16.

As shown in the figure, the driver functions *sifRead* and *sifWrite* do not use the register and bit field access functions anymore, but they directly access the abstract interface of the SIF. The *sifInit* and *sifIOCtl* functions are still accessing the TLM bus interface of the SIF using the register and bit field access functions.

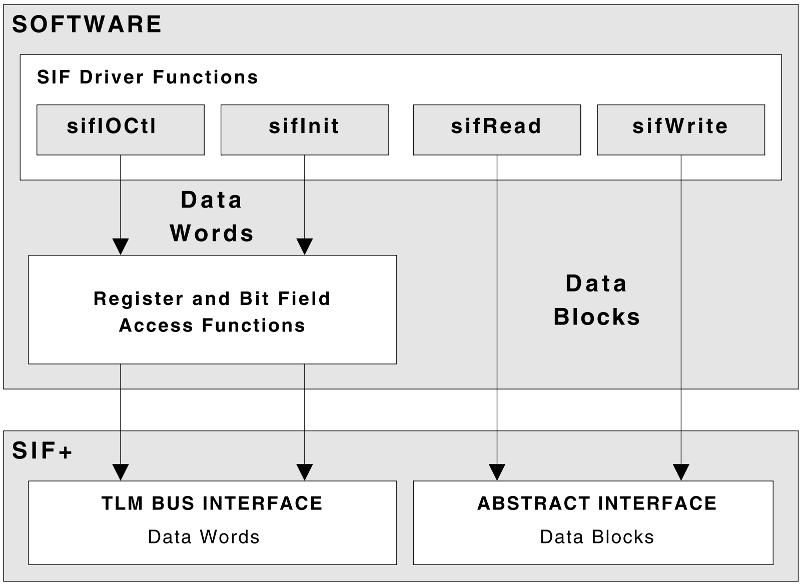

Figure 5.16. Abstracted SIF interface.

The abstraction of the external RX and TX interfaces is quite simple. It is achieved by extending the *ext_payload* class (see Listing 5.45) with a *data_block* member of data type *char** and an integer *block_size* member referring to the number of characters contained within the *data_block* array.

The *SIF+* model is fully compatible to the SIF TLM module, since it still contains the full TLM functionality. Therefore, it can be used in classical TLM simulations.

5.11 Conclusions

Within this chapter the modeling and implementation of HW/SW interfaces was explained, by high-lighting all involved areas, step by step. First some basic concepts were presented on both the HW side and the SW side explained how accessing HW through SW is accomplished. This concept was elaborated on in more detail taking also HW module internal register layouts into account. Many examples were presented illustrating alternative modeling styles which were also discussed. It was also illustrated how the internal communication infrastructure within a HW architecture is dealt with by the SW along with synchronization concepts for on-chip communication. Following these general considerations more detailed examples were provided using an example peripheral model. Based on this example the structure of low-level drivers was explained in detail.

As new contributions, it was also described how to improve the overall consistency of the HW/SW interface by using a single-source approach for obtaining its implementation. In this approach a peripheral specification is formalized in terms of its register layout and internal address map. The formal description serves as a basis for generating most parts of the HW/SW interface, including also the generation of different abstraction views on the HW as well some layers of the SW driver development. Furthermore, new concepts were introduced for raising the abstraction level for HW modeling by abstracting the data flow within the communication between HW and SW and also merging parts of HW and SW to a single abstract HW interface.

The explained concepts on modeling HW/SW interfaces show the huge diversity and the non-negligible complexity of modeling HW/SW interfaces. Using virtual prototyping, a close interaction of designers developing drivers and HW designers becomes possible at early stages of the whole design process. By getting HW and SW even closer through data and interface abstractions, a much better quality of the HW/SW interface can be achieved due to team working over different design domains namely HW and SW. Hence, virtual prototyping can also be considered as a bridge in between these domains.

References

[Mak] Mako Templates for Python. *Hyperfast and lightweight templating for the Python platform*. www.makotemplates.org

[MIT] raw Homepage. *raw Architecture Workstation*. www.cag.csail.mit.edu/raw

[OSC] OSCI TLM Working Group. *OSCI standard for SystemC TLM*. www.systemc.org

[Pyt] Python Software Foundation (PSF). *Python Programming Language*. www.python.org

[SPI] SPIRIT Consortium. *IP-XACT Standard*. www.SPIRITconsortium.org/tech/docs

[Wik] Wikipedia's Z80 Article. *Zilog Z80*. en.wikipedia.org/wiki/Zilog_Z80