On hard real-time scheduling of cyclo-static dataflow and its application in system-level design

Bamakhrama, M.A.M.

Chapter 5

System-Level Synthesis

Build a system that even a fool can use,
and only a fool will want to use it.

George Bernard Shaw

SYSTEM-level synthesis represents the fifth step in the proposed design flow. The inputs to this step are the program, architecture, and mapping specifications. The program specification consists of the PPNs derived in Chapter 3 together with the tasks' parameters and buffer sizes derived in Chapter 4. The architecture specification describes the number of processors as derived in Section 4.8. Finally, the mapping specification, derived in Section 4.8, associates each task with the processor on which it runs. All these specifications are used, as shown in Figure 5.1, to generate the MPSoC platform which consists of the hardware part together with the software running on that hardware. The whole system-level synthesis procedure is performed using ESPAM [NSD08]. ESPAM is a system-level synthesis tool for streaming systems with support for model-based design. We extended ESPAM in order to: (1) generate the hardware architecture explained in the following section, (2) generate the software with the proper scheduling and communication infrastructures explained later in Section 5.2, and (3) add support for Xilinx ML605 and Zynq boards that are used later in Chapter 6 for prototyping the synthesized systems.

5.1 Hardware

In this dissertation, we consider a hardware platform consisting of a tiled distributed memory MPSoC as shown in Figure 5.2. The on-chip interconnect is assumed to be a *predictable* on-chip interconnect. A predictable on-chip interconnect is one that

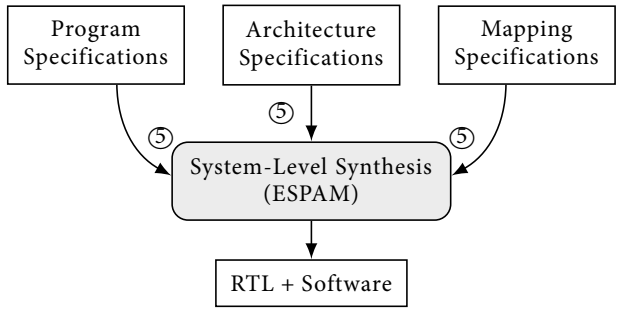


Figure 5.1: *Electronic System-Level Synthesis*

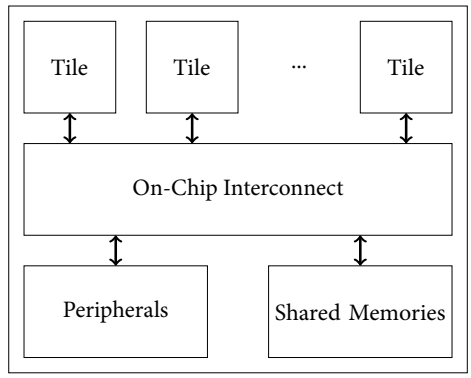


Figure 5.2: *Top-level block diagram of the hardware platform considered in this dissertation*

provides bounded worst-case latency on read/write operations between any communicating source and destination pair in the SoC. An example of such interconnect is the Æthereal network-on-chip [GDR05]. The aforementioned assumption is necessary in order to compute in Section 4.3 a safe upper bound on the worst-case execution time of each actor.

Each tile consists, as shown in Figure 5.3, of a processor, several memories, and a timer. Each tile contains three dedicated memories:

- Program Memory (PM) to store the programs' binaries
- Data Memory (DM) to store the data segments, heap and stack
- Communication Memory (CM) to store the data sent to other processors

Each processor writes the processed data to its local communication memory. After that, remote consumer processors read this data from the communication memory of the producer processor. This means that data writes are always local, and data reads are either local or remote depending on the actual mapping of tasks to processors. All the memories are implemented as dual-port memories which means that the commu-

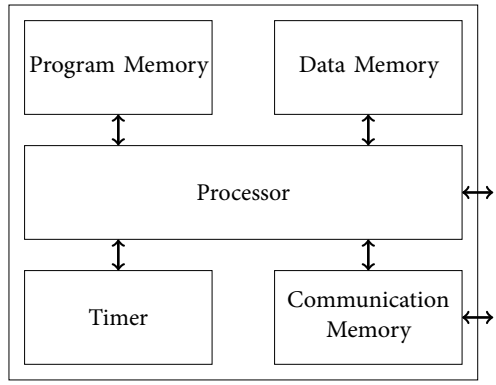


Figure 5.3: *Tile organization*

nication memory can be accessed by its owner processor and a remote processor at the same time.

A complete detailed picture of the SoC architecture integrated into ESPAM is shown in Figure 5.4. The on-chip interconnect in the SoC is a general-purpose, high performance AXI-4 [ARM10] crossbar switch which is provided by a commercial IP vendor. The crossbar features a Shared-Address, Multiple-Data (SAMD) topology as shown in Figure 5.5. It has two arbiters: one for read transactions and one for write transactions. Both arbiters are independent and can be active at the same time. The arbitration policy can be configured to be round-robin or priority-based. Parallel write and read data pathways connect each master to all the slaves that it can access according to a sparse connectivity map. When more than one source has data to send to different destinations, data transfers can occur independently and in parallel. We configure the crossbar to use the round-robin arbitration policy which enables us to derive a safe upper bound on the latencies of the communication operations.

In order to perform hardware generation, we store the hardware platform shown in Figure 5.4 as a parametrized template in ESPAM. Then, we use ESPAM to generate the actual platform in Xilinx Platform Studio (XPS) Microprocessor Hardware Specifications (MHS, [Xil11]) format. This allows importing the hardware project directly into the Xilinx XPS tool and performing FPGA synthesis.

5.2 Software

Recall from Chapter 3 that the code of the parallelized programs is generated by the `PNgen` compiler. The generated code has a form as the one shown in the example in Figure 3.2 on page 42. Thus, the remaining components that we have to generate are: (1) the scheduling infrastructure, and (2) the communication infrastructure implementing

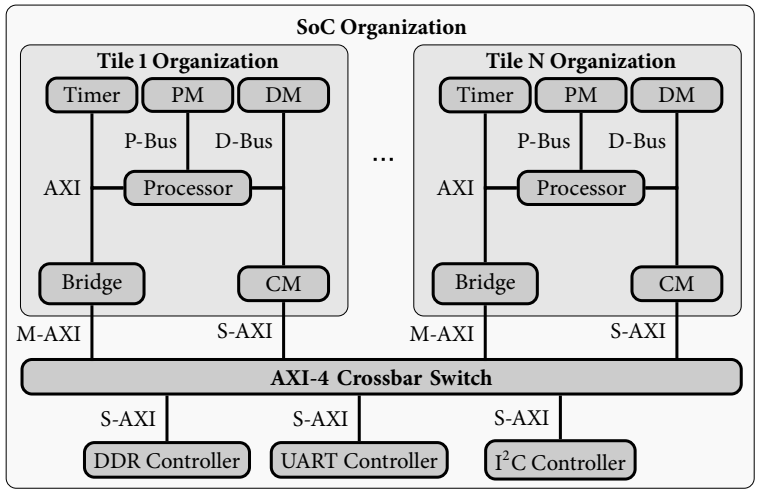


Figure 5.4: *Complete MPSoC architecture. P-Bus and D-Bus stand for program and data buses, respectively. M-AXI and S-AXI stand for AXI master and slave, respectively.*

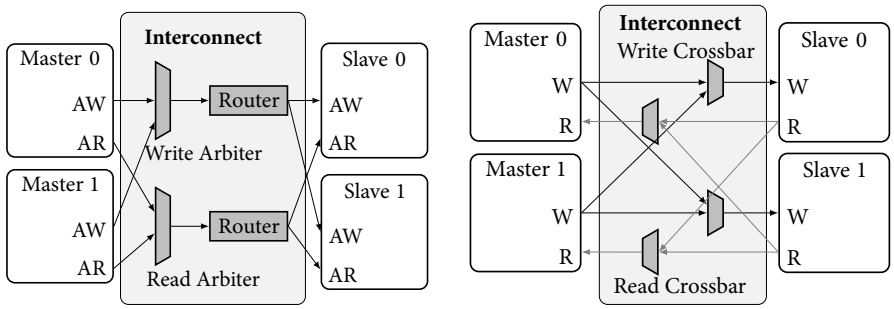
the FIFO reads/writes.

5.2.1 Scheduling Infrastructure

Recall from Chapter 4 that we schedule the tasks as periodic tasks. For the scheduler, we consider fixed task priority scheduling with Deadline Monotonic priority assignment. This choice is driven by the wide availability of real-time operating systems supporting fixed task priority scheduling. Nevertheless, it is important to note that different scheduling algorithms (e.g., EDF) can be used. We chose to implement the scheduling infrastructure using FreeRTOS [Reab]. FreeRTOS is an open source RTOS that implements fixed task priority scheduling and supports Xilinx FPGAs which are used later in Chapter 6 for evaluating the synthesized systems.

Tick-Based Implementation

FreeRTOS, as many real-time operating systems, relies on using **hardware timers** to keep track of time. Such timers generate periodic interrupts and these interrupts cause the OS to invoke the scheduler. A single interrupt and the associated scheduling event are called **OS clock tick**. The OS clock tick defines the shortest time granularity visible to the OS. OS clock tick is different from the processor (or CPU) clock tick. A processor clock tick refers to the duration of a single clock cycle of the clock signal used to operate the processor. Most of the WCET analysis tools measure the WCET of a task in terms of processor clock cycles. However, the RTOS can keep track only of time durations that



(a) Address Topology (b) Data Topology

Figure 5.5: *Crossbar Topology. AW stands for AXI write address channel, AR for AXI read address channel, W for write data channel, and R for read data channel.*

are multiple of the OS clock tick duration. Therefore, it is necessary during the system synthesis phase to ensure that all the timing parameters are converted to the appropriate OS clock tick values. This can be done, for example, by rounding the parameters up to the nearest multiple of the OS clock tick duration. To illustrate the previous concepts, we provide the following example.

Example 5.2.1. Suppose that we have a system comprised of a processor with a clock frequency equal to 1 GHz (i.e., processor clock cycle is 1 ns). Suppose that we want to run a task T_1 with the following parameters (all in processor clock cycles) $T_1 = (C_1 = 1.5 \times 10^6, P_1 = 2.5 \times 10^6, D_1 = 2.5 \times 10^6, S_1 = 0)$. Now, suppose that the OS clock tick frequency is 1000 Hz. This means that the OS performs scheduling events every 1 ms, which is equivalent to 10^6 processor clock cycles. We see that the period and deadline of T_1 are not multiples of the OS clock tick duration. Therefore, P_1 and D_1 must be rounded up to the nearest multiple of the OS clock tick duration which is 3.0×10^6. Such rounding might of course violate the timing requirements dictated by the designer. Therefore, it is important to keep in mind the effect of such rounding while specifying the system and program timing requirements. □

One effect of the tick-based implementation that must be taken into account is the ratio between the tasks' WCET and the OS clock tick duration. If the WCET is a fraction of the OS clock tick, then the resulting schedule has sub-optimal throughput with under-utilized processors and the overhead of the RTOS is not amortized. On the other hand, if the WCET is larger than the OS clock tick duration (preferably multiples of the OS clock tick), then the RTOS overhead is amortized. Therefore, it is important to consider this relation between the tasks' WCET and the OS clock tick duration when the timing parameters are converted from CPU clock cycles into OS clock ticks.

A periodic task T_i can be implemented in FreeRTOS as shown in Listing 3. Variable

Listing 3 Implementing a periodic task in FreeRTOS

```
1  void task(void *arg) {
2    portTickType LastReleaseTime;
3    const portTickType Period = 5;
4    LastReleaseTime = xTaskGetTickCount();
5
6    for (;;) {
7      function();
8      vTaskDelayUntil( &LastReleaseTime, Period );
9    }
10 }
```

`LastReleaseTime` records, as its name implies, the last release time of T_i in OS clock ticks. This variable is initialized when the task starts. Constant `Period` represents the period of the task in terms of OS clock ticks. For example, in Listing 3, the period is 5 OS clock ticks. Inside the `for`-loop, the task function (i.e., `function()`) is executed infinitely. Upon each execution, function `vTaskDelayUntil`, which is part of the FreeRTOS API, is called. The detailed description of `vTaskDelayUntil` is shown in Figure 5.6. The function takes two parameters: `LastReleaseTime` and `Period`. Upon calling it, it puts the task in the sleep state and schedules it for reactivation at time t = `LastReleaseTime` + `Period`. It also updates the value of `LastReleaseTime` accordingly.

Another effect of the tick-based implementation that must be also taken into account is the need to synchronize the time returned by `xTaskGetTickCount()` among the different processors. In an MPSoC, the clock signals of the different processors are usually generated from a single "reference" clock signal produced by an oscillator. Therefore, the clock signals used by the processors can be kept in phase. The moment, at which the OS clock tick count returned by `xTaskGetTickCount()` is initialized, can be synchronized through the use of a global barrier in the initialization code of each processor. For example, see line 14 in Listing 5.

Example 5.2.2. Consider the PPN shown in Figure 3.2 on page 42. Process $\mathcal{P}_{\texttt{snk}}$ is realized under FreeRTOS as shown in Listing 4. Note that the `while`-loop shown in Figure 3.2 is replaced with `for(;;)` in Listing 4. Note also that `vTaskDelayUntil` is placed such that after each invocation of function `snk`, the task postpones its next execution to the next release time in accordance with the real-time periodic task model as defined in Section 2.4.1. The `READ` primitive together with its counterpart `WRITE` are used to read/write from/to the FIFOs, respectively. These two primitives are explained later in Section 5.2.2. □

Function Prototype:

```
void vTaskDelayUntil( portTickType *LastReleaseTime,
                      portTickType Period );
```

Description:
Delay a task until a specified time. This function can be used by cyclical tasks to ensure a constant execution frequency.
This function differs from `vTaskDelay()` in one important aspect: `vTaskDelay()` specifies a time at which the task wishes to unblock relative to the time at which `vTaskDelay()` is called, whereas `vTaskDelayUntil()` specifies an absolute time at which the task wishes to unblock.
`vTaskDelay()` will cause a task to block for the specified number of ticks from the time `vTaskDelay()` is called. It is therefore difficult to use `vTaskDelay()` by itself to generate a fixed execution frequency as the time between a task unblocking following a call to `vTaskDelay()` and that task next calling `vTaskDelay()` may not be fixed (the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes).
Whereas `vTaskDelay()` specifies a wake time relative to the time at which the function is called, `vTaskDelayUntil()` specifies the absolute (exact) time at which it wishes to unblock.
It should be noted that `vTaskDelayUntil()` will return immediately (without blocking) if it is used to specify a wake time that is already in the past. Therefore a task using `vTaskDelayUntil()` to execute periodically will have to re-calculate its required wake time if the periodic execution is halted for any reason (for example, the task is temporarily placed into the Suspended state) causing the task to miss one or more periodic executions. This can be detected by checking the variable passed by reference as the `LastReleaseTime` parameter against the current tick count. This is however not necessary under most usage scenarios.
This function must not be called while the RTOS scheduler has been suspended by a call to `vTaskSuspendAll()`.

Parameters:
`LastReleaseTime`: Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialized with the current time prior to its first use (see the example below). Following this the variable is automatically updated within `vTaskDelayUntil()`.
`Period`: The cycle time period. The task will be unblocked at time `(*LastReleaseTime + Period)`. Calling `vTaskDelayUntil` with the same `Period` parameter value will cause the task to execute with a fixed interval period.

Figure 5.6: *Detailed description of function `vTaskDelayUntil`. Source: [Reaa].*

Enforcing Start Times

In many commercial real-time operating systems, the API provided by the RTOS does not allow the programmer to specify explicitly the start time of a task when the task is

Listing 4 Implementing process $\mathcal{P}_{\texttt{snk}}$ in Figure 3.2 as a periodic task under FreeRTOS

```
1  void task_snk(void *arg) {
2    portTickType LastReleaseTime;
3    const portTickType Period = 5;
4    LastReleaseTime = xTaskGetTickCount();
5
6    for (;;) {
7      for(i=1;i<=10;i++) {
8        for(j=1;j<=3;j++) {
9          if(j<=2)
10           READ(&in1,IP1,SIZE_OF_in1,SIZE_OF_FIFO_E4);
11         else
12           READ(&in1,IP2,SIZE_OF_in1,SIZE_OF_FIFO_E5);
13
14         READ(&in2,IP3,SIZE_OF_in2,SIZE_OF_FIFO_E3);
15
16         snk(in1,in2);
17         vTaskDelayUntil( &LastReleaseTime, Period );
18       }
19     }
20   }
21 }
```

created. Therefore, it is the programmer's responsibility to implement a mechanism which ensures that a task starts on its specified start time as derived in Section 4.4. The start time of a task may be realized under such an RTOS using several mechanisms. We list here two possible mechanisms:

1. The first mechanism is to use a *master* task that releases the programs' tasks at the time when they are supposed to start. This master task releases each task at the OS clock tick on which the task should begin its execution. After starting all the tasks, the master task can be terminated or put into a permanent sleep state.
2. The second mechanism is to release all the tasks simultaneously. After that, each task is put into sleep state from the moment of simultaneous release till the moment at which it should start.

The first mechanism provides tight control on when to start the tasks. However, a disadvantage of this mechanism is the extra overhead introduced by the master task. The utilization of the master task must be taken into account while deriving the architecture and mapping specifications (see Section 4.8) in order to avoid any deadline misses.

The second mechanism does not have the utilization overhead of the master task mechanism. This mechanism keeps the execution of the task conforming to the periodic

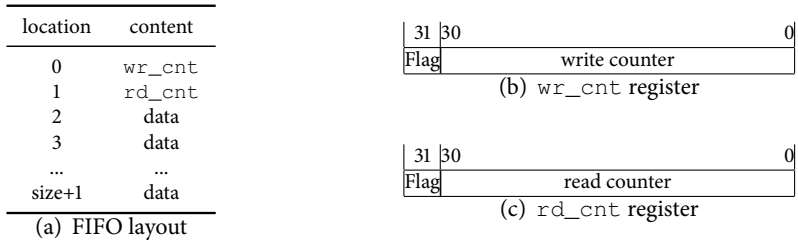


Figure 5.7: *FIFO layout in memory and the read/write registers*

task model as it does not cause the task to block the processor from other tasks.

Example 5.2.3. Consider task `snk` shown in Listing 4. The implementation of `snk` assuming the second mechanism (i.e., simultaneous release) is shown in Listing 5. Variable `SimultaneousReleaseTime` is passed from the `main` function that releases all the program's tasks. This variable is used to put the task in sleep state until its start time. After that, the task executes as a periodic task starting from the time assigned to `LastReleaseTime` at line 29. □

5.2.2 Communication Infrastructure

The communication infrastructure deals with the implementation of the `READ` and `WRITE` primitives shown for example in Figure 3.2 on page 42 and Listing 4 on page 82. These primitives provide the actors with the ability to communicate among each other. Recall from Section 5.1 that each actor produces data to its local communication memory, and reads data from its communication memory and/or remote communication memories. The FIFOs are implemented as circular buffers, and they are stored in the communication memories of the processors (see Figure 5.4 on page 78). The size of a single data word in the FIFOs is 32 bits. Each FIFO contains two special data words called `wr_cnt` and `rd_cnt` as shown in Figure 5.7(a). These data words store two pieces of information as shown in Figure 5.7(b) and 5.7(c): (1) the write and read counters of the FIFO, and (2) a special bit called "Flag" which is used for detecting counter overflows. Whenever the read/write counter exceeds the FIFO size, the flag bit is toggled. Storing the counter and flag in one data word enables updating the FIFO state in a producer/consumer task using a single atomic operation.

A detailed implementation of the read/write operations is depicted in Listings 6 and 7. The read/write operations accept four input parameters: (1) a pointer to the value read/written (`val`), (2) a pointer to the FIFO (`pos`), (3) the amount of data, in 32-bit words, being read/written during an invocation of the read/write operation (`len`), and (4) the size of the FIFO in 32-bit words (`size`). The implementation shown in Listings 6 and 7 assumes that the amount of data written/read by the producer/consumer,

Listing 5 Implementing the simultaneous release mechanism under FreeRTOS. The listing shows only the relevant code to the simultaneous release mechanism and other non-relevant details are omitted.

```
1  int main() {
2    static portTickType SimultaneousReleaseTime;
3    /* xTaskCreate() (part of FreeRTOS API) creates new tasks */
4    xTaskCreate( task_snk, "snk", SNK_STACK_SIZE, \
5                 &SimultaneousReleaseTime, SNK_PRIORITY, NULL);
6    /* Other xTaskCreate() invocations go here */
7
8    /* xTaskGetTickCount() (part of FreeRTOS API) returns the count
9       of OS clock ticks since vTaskStartScheduler() was called.
10      If vTaskStartScheduler() was not called, it returns 0 */
11   SimultaneousReleaseTime = xTaskGetTickCount();
12
13   /* Set up a global barrier to synchronize the processors */
14   waitForGlobalStartSignal(); /*  */
15
16   /* vTaskStartScheduler() (part of FreeRTOS API) invokes the
17      scheduler for the first time. It also resets the count of
18      OS clock ticks returned by xTaskGetTickCount() to 0 */
19   vTaskStartScheduler();
20 }
21 void task_snk(void *arg) {
22   portTickType LastReleaseTime, SimultaneousReleaseTime;
23   const portTickType Period = 5;
24   const portTickType StartTime = 20;
25   /* SimultaneousReleaseTime is set in main() */
26   SimultaneousReleaseTime = *((portTickType *) arg);
27   vTaskDelayUntil( &SimultaneousReleaseTime, StartTime );
28   /* Set LastReleaseTime to the actual start time */
29   LastReleaseTime = xTaskGetTickCount(); /*  */
30
31   for (;;) {
32     for(i=1;i<=10;i++) {
33       for(j=1;j<=3;j++) {
34         if(j<=2) READ(&in1,IP1,SIZE_OF_in1,SIZE_OF_FIFO_E4);
35         else     READ(&in1,IP2,SIZE_OF_in1,SIZE_OF_FIFO_E5);
36         READ(&in2,IP3,SIZE_OF_in2,SIZE_OF_FIFO_E3);
37         snk(in1,in2);
38         vTaskDelayUntil( &LastReleaseTime, Period );
39 } } } }
```

Listing 6 An example implementation of the read macro under FreeRTOS

```
1  READ(void *val, void *pos, int len, int size){
2    volatile int *fifo=(int *)pos;
3    int r_cnt = fifo[1];
4    int w_cnt = fifo[0];
5    int i = 0;
6    while(w_cnt == r_cnt){
7      taskDISABLE_INTERRUPTS();
8      xil_printf("PANIC! Buffer Underflow\n");
9      for (;;);
10   }
11   for(i = 0; i < len; i++){
12     ((volatile int *)val)[i]= fifo[(r_cnt & 0x7FFFFFFF)+2+i];
13   }
14   r_cnt += len;
15   if((r_cnt & 0x7FFFFFFF) == size){
16     r_cnt &= 0x80000000;
17     r_cnt ^= 0x80000000;
18   }
19   fifo[1] = r_cnt;
20 }
```

respectively, is always the same. That is, for a given communication channel, the value of `len` used in `WRITE` by the producer and the value of `len` used in `READ` by the consumer are the same. When a task T_i reads `len` words from the FIFO into a buffer `val`, the read macro performs the following steps:

1. The read and write counters are copied into local variables (lines 3 and 4)
2. If a buffer underflow occurs (i.e., FIFO is empty), then the interrupts are disabled and a "panic" message is printed to the user to indicate that a buffer underflow has occurred (lines 6-10). It is important to note that this situation should not occur under normal operating conditions since the start times and buffer sizes derived in Sections 4.4 and 4.5 are valid. Recall from Section 1.2 that normal operating conditions mean that both system hardware and software function properly without faults.
3. The `for`-loop copies the data from the communication memory into `val` and the read counter is incremented (lines 11-14).
4. After that, the read counter is checked for overflow condition and Flag is toggled accordingly (lines 15-18).
5. Finally, the macro updates `rd_cnt` register in the FIFO with the new value of the read counter by doing a single atomic assignment (line 19).

Analogously, when a task T_i writes `len` words to the FIFO from a buffer `val`, the

Listing 7 An example implementation of the write macro under FreeRTOS

```
1  WRITE(void *val, void *pos, int len, int size){
2    volatile int *fifo=(int *)pos;
3    int w_cnt = fifo[0];
4    int r_cnt = fifo[1];
5    int i = 0;
6    while(r_cnt == (w_cnt ^ 0x80000000)){
7      taskDISABLE_INTERRUPTS();
8      xil_printf("PANIC! Buffer overflow\n");
9      for (;;);
10   }
11   for(i = 0; i < len; i++) {
12     fifo[(w_cnt & 0x7FFFFFFF)+2+i] = ((volatile int *)val)[i];
13   }
14   w_cnt += len;
15   if((w_cnt & 0x7FFFFFFF) == size){
16     w_cnt &= 0x80000000;
17     w_cnt ^= 0x80000000;
18   }
19   fifo[0] = w_cnt;
20 }
```

write macro performs the following steps:

1. The read and write counters are copied into local variables (lines 3 and 4)
2. If a buffer overflow occurs (i.e., FIFO is full), then, similar to `READ`, the interrupts are disabled and a "panic" message is printed to the user (lines 6-10). Note again that this situation should not occur under normal operating circumstances since the start times and buffer sizes derived in Sections 4.4 and 4.5 are valid.
3. The `for`-loop copies the data from `val` into the communication memory and the write counter is incremented (lines 11-14).
4. After that, the write counter is checked for overflow condition and Flag is toggled accordingly (lines 15-18).
5. Finally, the macro updates `wr_cnt` register in the FIFO with the new value of the write counter by doing a single atomic assignment (line 19).