Timers

Prof. John McLeod

ECE3375, Winter 2022

This lesson discusses timers. Every microcontroller has at least one, and often several timers, for various purposes. The basic operation of timers is discussed, and specific examples of watchdog timers and system clocks are given. Two peripherals that are based on timers: output compare units, and input capture units are discussed. Finally, the structure and operation of the interval timers on the DE1-SoC board is explained, with example code given in Assembly and C.

General Motivation

Microcontrollers are usually required to interact with other devices. This often requires the ability to accurately measure intervals of time — and it should be self-evident that a **timer** is just the sort of peripheral for handling this. The basic component of a timer

is an n-bit counter, like the ones you previously studied in digital logic. Of course a useful times has a bunch of extra functionality to control this counter, and as the interval timers on the DE1-SoC board demonstrate all-to-well, using a timer in software can be frustatingly complicated. Fortunately, this handy lesson note is with you every step of the way.

Timing Peripherals

Every microcontroller contains on or more timing peripherals. These are essential for synchronizing the microcontroller with other peripherals, or for allowing controlled delays in program execution. A microcontroller often has several timing peripherals, many with special purpose.

Definition: A timing peripheral, or **timer**, is a device that counts increments of a well-defined time interval.

This time interval is often not the same as the CPU clock, but is usually a multiple of the clock.

- Some timers may only count up, some may only count down, and others may be configurable to count up or down.
- Timers can be started or stopped.
- Timers may count to some limit and then stop, or may count and then repeat.

Consequently, most timers are reasonably complicated peripherals, and have some **structure**.

- Most timers have a set of **control bits** (in a control register) that determine how the timer operates.
- Timers also have a **counter** register (a data register) that is adjusted to keep track of the time.
- Many timers also have a register that can store a time limit which tells the counter when to stop (another data register).
- The operating state of the timer may also be indicated by a set of **status bits** (in a status register).

Clearly a timer needs to accept both input and output, as control flags and count limits must be set and the current time needs to be read.

Watchdog Timers

Microcontrollers are typically programmed to run continuously in an endless loop. But — as everyone who has ever used Windows knows all too well — sometimes things go wrong in software. To prevent a microcontroller system from getting stuck forever, and to allow it to restart without user input, every microcontroller has at least one **watchdog timer**. When the watchdog timer is used:

- The watchdog timer is always counting, and if the count ever *wraps around*, [1] the microcontroller completely resets.
- Under normal operation, the microcontroller software always *rolls back* the watchdog timer before it overflows.
- If the system ever gets stuck, the watchdog timer will eventually cause a reset.
- This process is entirely transparent to the user.

Each of the two CPU cores in the Cortex-A9 has a watchdog timer, and two additional watchdog timers are present outside of the CPU on the DE10-Standard board (the *HPS L4 Watchdog Timers*). There is more than one watchdog because each of these timers has a control bit that determines whether they are to act as a watchdog timer or as a normal general-purpose timer. [2] There isn't any point in using more than one timer as a watchdog at once.

[1] Meaning the counter overflows when counting up, or underflows when counting down.

[2] I couldn't figure out how to fool the online simulator into generating a watchdog-related reset, so there isn't much I can demonstrate on this subject.

Example: Consider a system with a $1\,\mathrm{MHz}$ 16-bit counter watchdog timer. The counter on this timer will overflow in:

$$2^{16} \times (1\,\mathrm{MHz})^{-1} = 65.536\,\mathrm{ms}.$$

Before $65.536\,\mathrm{ms}$ have elapsed, the CPU needs to manually roll back, or otherwise reset the watchdog timer.

- If the CPU ever freezes for any reason, it will automatically reset after hanging for $65.536\,\mathrm{ms}$.

System Clock Timers

The most fundamental use of timers is to allow us to perform some action at regular intervals. This is accomplished by setting the counting interval of a timer, and monitoring the appropriate **status bit** to determine when that interval is complete.

- This only works if the timer interval is significantly longer than the CPU clock speed (as is often the case) — we typically assume that software instructions are executed instantaneously.

The exact operation of a system clock timer depends on whether it counts up or down.

Example: An up-counting system clock with a 16-bit counter will set its **timeout flag** status bit on each overflow from `0xFFFF` to `0x0000`. To set a specified time interval Δt, we should write `0x10000` $- f\Delta t$ to the **data register**, where f is the clock speed.

- The timer will start counting at this value, and count up to `0xFFFF` before setting the flag.
- If $f = 1\,\text{MHz}$, and we want to measure an interval of $1\,\text{ms}$, we should set the 16-bit counter to $(64\,536)_{10}$.

Example: A down-counting system clock with a 16-bit counter will set its **timeout flag** status bit on each underflow from `0x0000` to `0xFFFF`. To set a specified time interval Δt, we should write $f\Delta t$ to the **data register**, where f is the clock speed.

- The timer will start counting at this value, and count down to `0x0000` before setting the flag.
- If $f = 1\,\mathrm{MHz}$, and we want to measure an interval of $1\,\mathrm{ms}$, we should set the 16-bit counter to $(1000)_{10}$.

In a simple timer, you may need to reset the interval each time you want to count for that period. More sophisticated timers can "remember" the time interval, so after a timeout you just need to tell the timer to restart counting. Often these timers can also be operated in a **continuous** counting mode, where the flag is set after an overflow or underflow, but the count doesn't stop.

Model Timer

For conceptual questions in this course we will use a very simplified model for a timer, as shown in Figure 1.

- This timer always counts **down**.
- The timer interval is written to the *timer* **data register**.
- The current counting time can also be read from this register.
- Bit 0 in the **status register** is set to 1 when the timer reaches zero.

It is worth stressing that this is *not* how the ARM timers work (see below for those gory details).

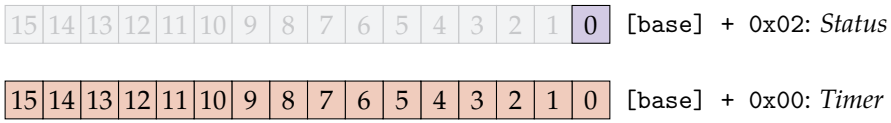

Figure 1: Structure of the simplified model of a timer used in conceptual questions in this course. Note the addresses are based on 16-bit registers, not the 32-bit registers used in an ARM.

Output Compare Units

One common peripheral that is based on a timer is an **output compare unit**. This is a hardware peripheral that will generate some particular output event at specific, precise times.

- Essentially, the output compare unit responds to instructions like: "I want bit 4 of port C to turn on in exactly $100\,\mathrm{ms}$ from now".
- Of course, this can also be done in software, but then the CPU constantly needs to check the status of the timer and may need to suddenly branch out of some other (possibly complicated) routine to set the output.
- Once the instruction is written to the output compare unit, however, the CPU can ignore it and continue on with the main program.

An output compare typically requires the following steps to be implemented:

1. Enable an appropriate timer. Sometimes this is built into the unit.
2. Specify a time, usually as some number of "ticks" from the present time.
3. Specify an event to occur when the time interval has elapsed: often toggling some pin on a port. Sometimes this pin is fixed, sometimes it can be specified.

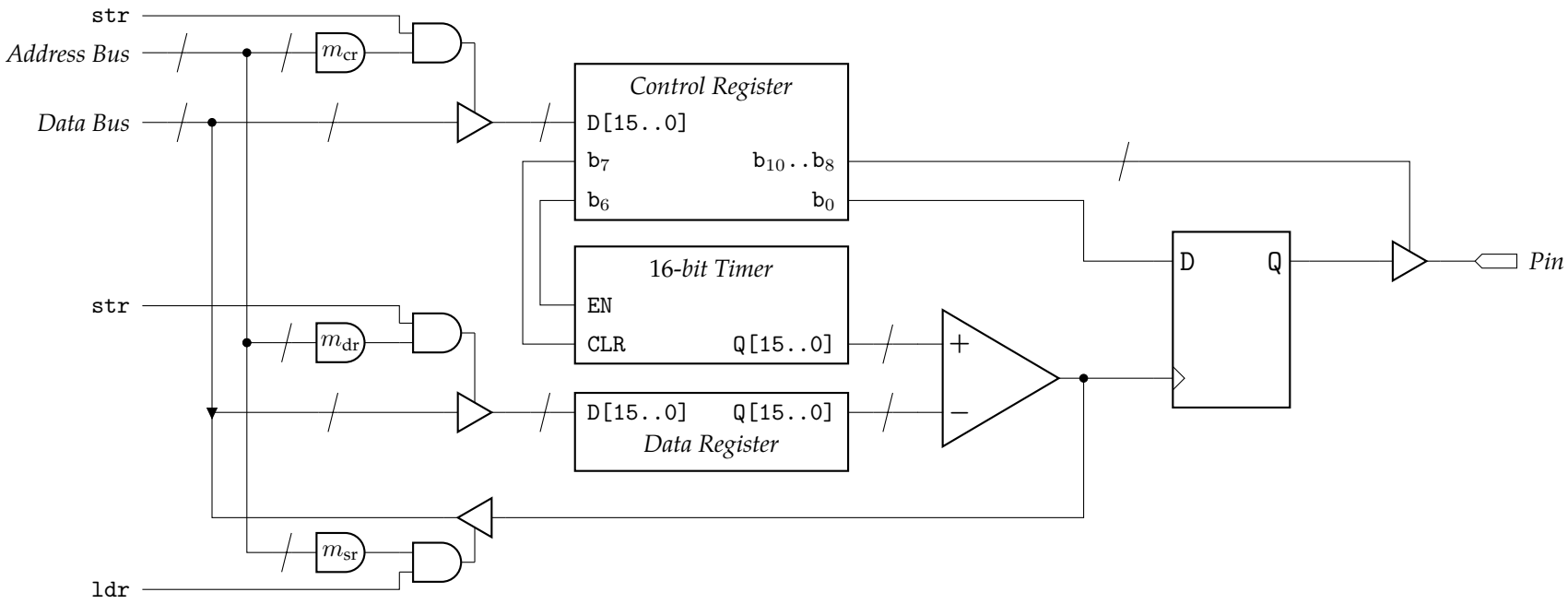

Figure 2: A simplified schematic of an output compare unit (that "sort of works" — probably a few details are missing). Only a single output pin is shown, but consider the tristate buffer by the pin to allow any one of 8 pins to be selected. The count interval is written to the data register, a single status bit (at the bottom) is set when the current count on the 16-bit timer goes above the interval, this also triggers the clock for the D flipflop, allowing the control register value in bit 0 to be written to the output pin.

The output compare unit will start the timer, set the output after the interval is complete, the unusually set some status flag after it has completed the task. The whole point of an output compare unit is that the above processes occur by the hardware unit, not by direct instruction from the processor.
A model output compare unit is shown schematically in Figure 2, the memory mapped structure is shown in Figure 3. This model operates as follows:

- The timer counts up.
- Control register bits 8, 9, and 10 select one of 8 output pins.

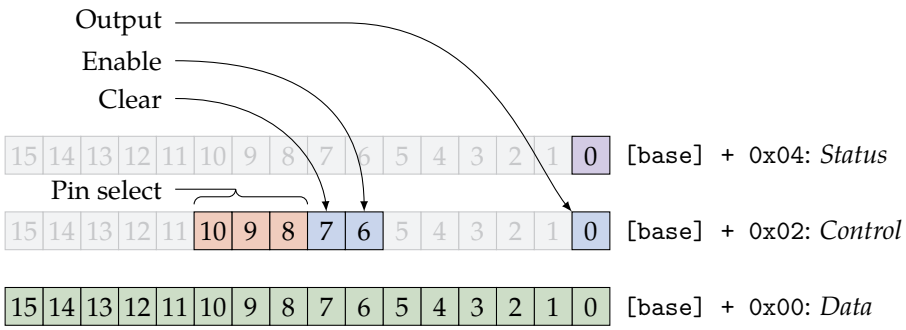

Figure 3: Structure of the model output compare unit. Note the addresses are based on 16-bit registers, not the 32-bit registers used in an ARM.

- Control register bit 7 clears the timer, control register bit 6 is used to start or stop the timer. Control register bit 0 is the value to be written to the output pin at the given time.
- The time interval is loaded into the data register.
- The time interval is compared to the current count, as soon as the count exceeds that interval, the value in control register bit 0 is written to the output pin and the status register bit 0 is set to one.

Please note that this is just a model that we may use in conceptual questions for this course, this does not necessarily correspond to an actual output compare unit on a real microcontroller.

Note: In various textbooks and technical documentation for microcontrollers, output compare units are often implemented to produce a **pulse train** of a particular frequency, and sometimes even a particular duty cycle. [3] This is basically a clock output, and is an extension of the simple functionality described here.

[3] This latter functionality, of changing the duty cycle, is sometimes instead described as **pulse width modulation** (PWM).

Input Capture Units

A second common peripheral that is based on a timer is an **input capture unit**. This is a hardware peripheral that is the mirror of an output compare unit: this peripheral will record the exact time a particular input is received.

- Essentially, the input capture unit responds to instructions like: "watch pin 7 of port B. Tell me exactly when it goes high."
- Again, this can be done in software, but that requires the CPU to constantly be checking that input port. If the input result isn't needed immediately (i.e., it just needs to be recorded for future use), then the CPU probably has more important things to do.
- Once the instruction is written to the input compare unit, the CPU can ignore it and continue on with the main program.

An input compare typically requires the following steps to be implemented:

1. Set a pin to watch.
2. Set a state to watch for (high, low, toggle).
3. Start the timer.

When the appropriate state is present on the particular pin, the timer will stop and save the current count to some data register.

Often a status bit is also set to indicate the event happened. Input capture units often share components — particularly the timer — with output capture units. Again, the whole point of an input capture unit is that the above processes occur by the hardware unit, not by direct instruction from the processor.
A model input capture unit is shown schematically in Figure 4 — I tried to draw the circuit so that the similarity to an output compare unit is reasonably clear. The memory mapped structure would be the same as the output compare unit shown in Figure 3, except now the data register is for *reading from*, rather than *writing to*. This model operates as follows:

- The timer counts up.
- Control register bits 8, 9, and 10 select one of 8 input pins.
- Control register bit 7 clears the timer, control register bit 6 is used to start or stop the timer. Control register bit 0 is the value to be checked against the input pin.
- The input pin is assumed to be digital.
- The first time the input pin matches the specified value, the current count on the timer is stored in the data register.

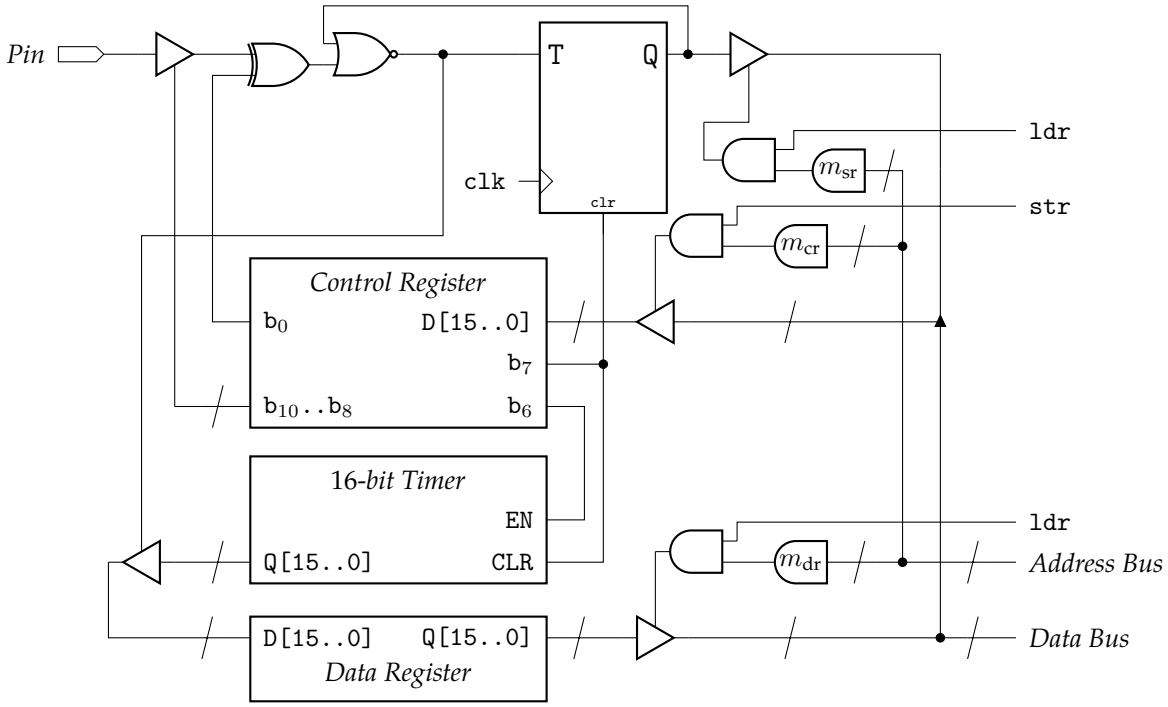

Figure 4: A simplified schematic of an input capture unit (that "sort of works" — probably a few details are missing). Only a single input pin is shown, but consider the tristate buffer by the pin to allow any one of 8 pins to be selected. The count interval is written to the data register the first time the input pin reaches the value in bit 0 of the control register.

DE10-Standard Interval Timer

There are two general-purpose **interval timers** on the DE10-Standard board that can be configured and used in software. These have the same basic functionality as the model timer described above, but a rather more complicated **structure**.

- These timers operate at $100\,\mathrm{MHz}$, which is considerably slower than the ARM®Cortex-A9 CPU clock.
- These timers always **count down** to zero.

We can set the **count down interval** by writing the appropriate *multiple of the timer clock cycles* to the appropriate **data register**. One or both of the *interval timers* may be used for the labs in this course whenever a timer is needed. These are memory mapped to the base addresses `0xFF202000` and `0xFF202020`. For simplicity, the **structure** of these timers is based on 32-bit **words**. However, due to a bunch of reasons, none of the actual inputs or outputs from the timer exceed 16-bit **half-words**.

- The first word at the base address of the timer is the **status register**. This uses only two (2) bits!
  - Bit 0 is a flag that indicates whether or not the timer has reached a **timeout** — i.e., counted down to zero.
  - Bit 1 is a flag that indicates whether or not the timer is running.

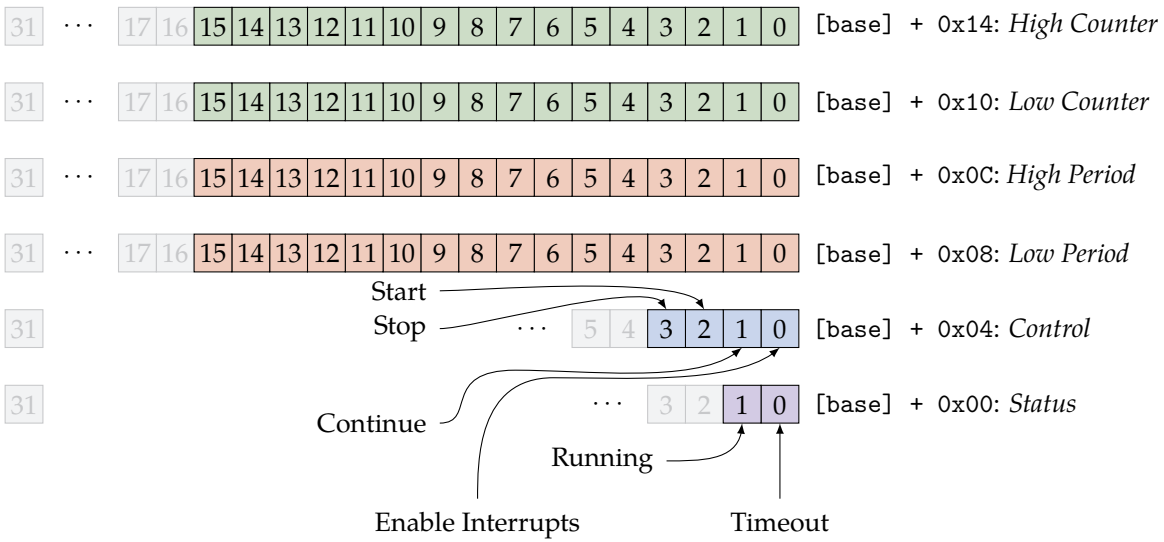

Figure 5: Structure of the interval timers on the DE10-Standard board. The base addresses are `0xFF202000` and `0xFF202020`. Note that although the timer is structured in words (32 bits), for legacy reasons at most only half-words (16 bits) are used for each part.

This **status register** can be read to determine the current state of the timer. This register can also be *partially* written to: if the **timeout flag** was previously set, writing *anything* to this register will clear it.

- The second word after the base address of the timer is the **control register**. This uses a whopping four (4) bits.
  - Bit 0 is a flag that enables the timer to send **interrupts** to the CPU. We will discuss interrupts in a later lesson. [4]
  - Bit 1 is a flag that tells the timer to count continuously, "wrapping around" back to the period value whenever the present interval is reached.
  - Bit 2 is a flag that tells the timer to start counting.
  - Bit 3 is a flag that tells the timer to stop counting.

[4] Set this bit to 1 in order to generate hardware interrupts, but unless interrupts are also enabled in the CPU nothing will happen.

This register can obviously be written to, as this is how the timer is controlled. It can also be read from, if you forget what the controls were.

- The third and fourth words after the base address are the **period registers**. These are a type of data register, and are used to set the count interval. Although the count interval is a 32-bit value — and the registers themselves are 32-bits — for legacy reasons this is split into the least significant 16 bits of two registers.

- The final two words after the base address are the **counter registers**. These are also a type of data register, and are used to record the current count state. Again, for legacy reasons the 32-bit state is split into the least significant 16 bits of two registers.

The structure of the timer is pretty frustrating — why, for a 32-bit timer, on a 32-bit architecture, is it necessary to split the timer interval across two half-word registers? Presumably these timers were first developed for 16-bit CPUs, and for "reasons" needs to keep it that way. [5]

- My griping aside, reading a writing to the timer isn't a huge problem as long as you remember how to use the barrel shifter to switch two half-words for word.

To continue complaining about this timer, I find operating it also very confusing.

[5] Incidentally, this timer silliness is an issue with the DE10-Standard development board, *not* the ARM®Cortex-A9 microcontroller, as the timer is an on-board peripheral, not an on-chip peripheral. So direct your complaints to Terasic, not ARM.

- I can't think of a reason why the **control register** has one bit for "start" and another bit for "stop".
- Setting the start bit to 1 will start the timer counting down.
- It seems that setting both the start and stop bit to 1 will also start the timer — it seems the "start bit" takes priority.
- If the timer is running, it will continue to run if the the entire **control register** is cleared to zero — it seems the "running bit" in the **status register** takes priority.
- You can't write directly to the **status register**, so it seems the only way to turn off the timer (other than letting it count down to zero if the "continue bit" is not set) is to set the "stop bit" to 1 and the "start bit" to 0 in the *control register*.

Finally, one last complexity! If you want to read the current state of the timer, *do not* read the **period registers**. The current count is stored in these registers, but attempting to read them directly will use one or more clock cycles and will make the actual timed interval less accurate.

- Instead, write some value — any value, as it is ignored — to either of the **counter registers**.
- This will cause the timer to mirror the current count to the **counter registers**.
- After that, you can read from the **counter registers** normally.

In this simulator, the timer is not completely accurate but it is still pretty close. Note that if you start the timer to run continuously [6] it will do so — even if you stop, or pause the assembly program. This usually isn't a problem, but can lead to unexpected results if you are trying to time a precise interval and also stepping through your code line-by-line to debug it.

[6] i.e. write `0x06` to the **control register** to set both the "start bit" and the "continue bit"

DE10-Standard HPS Timers

There are four additional general-purpose **timers** on the DE10-Standard board that are considered part of the "hard processor system" (HPS). These can be configured and used in software. These have the same basic functionality as the model timer described above, and a simpler **structure** compared to the *interval timers* previously discussed.

- Two of these timers operate at 100 MHz, the remaining two operate at 25 MHz.
- These timers always **count down** to zero.

We can set the **count down interval** by writing the appropriate *multiple of the timer clock cycles* to the appropriate **data register**. Any of these *HPS timers* may be used for the labs in this course whenever a timer is needed. [7] These timers are memory mapped to the base addresses `0xFFC08000` and `0x0xFFC09000` for those with the 100 MHz clock, and `0xFFD00000` and `0xFFD01000` for those with the 25 MHz clock. Again, the **structure** of these timers is based on 32-bit **words**. Fortunately the data registers use a full 32 bits, unlike the *interval timers*.

- The first word at the base address of the timer is a **data register**. This uses all 32 bits, and is used to load the period that the timer should count for.
- The second word after the base address is another **data register**. This also uses all 32 bits, and is used to read the current

[7] Please note that these timers are **not implemented in the simulator**, they are available in hardware only!

count of a running timer.

- The third word is a **control register**. This uses only the lowest three bits.
  - Bit 0 is used to enable the timer. Set it to 1 to start the timer counting down, clear it to 0 to stop the timer. [8]
  - Bit 1 is used instruct the timer to use the period loaded into the first data register for counting. [9]
  - Bit 2 if a flag that enables the timer to send **interrupts** to the CPU. Again, we will discuss interrupts in a later lesson so you can ignore this bit for now. [10]
- The fourth word is a **status register**. This uses only the lowest bit. This register is called the *end-of-interrupt* register in the manual, and is related to generating hardware interrupts.
  - For our purposes, we just use this register to clear the **timeout flag** by **reading** from this register.
- The fifth word is another **status register**. This also uses only the lowest bit. This register is called the *interrupt* register in the manual, but should probably be called the "timeout" register.
  - When bit 0 is set to 1, the timer has finished counting down. [11]

In general I find the *HPS timers* much simpler to use than the *interval timers*. Just remember that you should check to see if the timer is

[8] The documentation for the *HPS timer* emphasizes that bit 0 should always be cleared to 0 **before** loading a new period into the data register.

[9] I don't understand why this control bit is necessary, and since the *HPS timer* is only available in hardware I am too lazy to tinker with the system to figure it out.

[10] Somewhat confusingly, if bit 2 is cleared to 0 then interrupts *are* generated, and if it is set to 1 then interrupts *are not* generated. Still, it doesn't matter whether or not hardware generates interrupts unless interrupts are also enabled in the CPU.

[11] Even more confusingly, if bit 2 in the *control register* is set to 1 to disable interrupts, then the *interrupt* and *end-of-interrupt* registers are never updated. The only way to check the operation of the timer is to read from the *count* register — but frequently checking this register interferes with the accuracy of the timer!

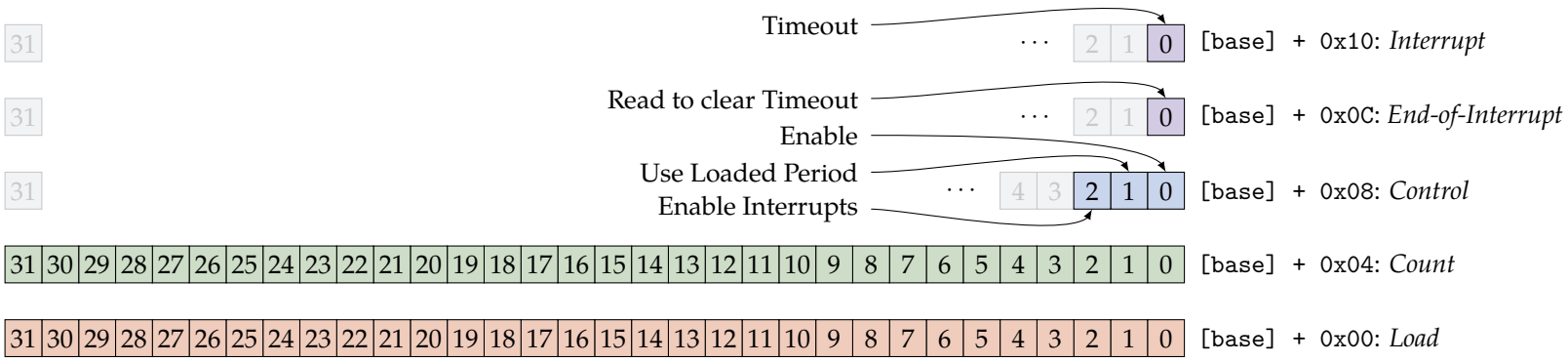

Figure 6: Structure of the HPS timers on the DE10-Standard board. The base addresses are `0xFFC08000`, `0xFFC09000`, `0xFFD00000`, and `0xFFD01000`.

finished counting by reading from the *interrupt* register — but then, after the timer has finished, you should clear bit 0 in the *interrupt* register by reading from the *end-of-interrupt* register. This step is the only (slightly) confusing aspect of using the *HPS timer*.

The major difficulty with this timer is that it is not available in the simulator.

ARM®Cortex-A9 Private Timers

There are also two more general-purpose timers available. These are built into the A9 processor, and as such each can only be accessed by one of the dual-cores in the CPU. This doesn't really matter for us: we aren't writing any multi-threaded code, so we don't care which CPU core is used in our code. These timers can again be configured and used in software, and are fairly simple.

- Both of these timers operate at 200 MHz.
- These timers always **count down** to zero.

We can set the **count down interval** by writing the appropriate *multiple of the timer clock cycles* to the appropriate **data register**. Any of these *A9 private timers* may be used for the labs in this course whenever a timer is needed. [12] These timers are memory mapped to the *same* base address `0xFFFEC600`. [13] Again, the **structure** of these timers is based on 32-bit **words**, and again the data registers use a full 32 bits, unlike the *interval timers*. In fact, the structure of the *A9 private timers* is so similar to the structure of the *HPS timers* that it is easy to confuse the two — which can be a problem because the status and control bits are a bit different.

- The first word at the base address of the timer is a **data register**. This uses all 32 bits, and is used to load the period that the timer should count for.
- The second word after the base address is another **data register**. This also uses all 32 bits, and is used to read the current

[12] Fortunately, these timers are implemented in the simulator.

[13] As each timer can only be accessed by one CPU core, there isn't a problem having two pieces of hardware at the same address. In principle this means we can only use one of these timers unless you want to explicitly write multi-threaded code.

count of a running timer.

- The third word is a **control register**. This uses the lowest three bits for controlling the timer.
    - Bit 0 is used to enable the timer. Set it to 1 to start the timer counting down, clear it to 0 to stop the timer.
    - Bit 1 is used instruct the timer to count continuously, "wrapping around" back to the period value once the present interval is finished counting.
    - Bit 2 if a flag that enables the timer to send **interrupts** to the CPU. Again, we will discuss interrupts in a later lesson so you can ignore this bit for now. [14]

  There is also a section of this register for **prescaling** the clock.
    - Bits 8 to 15 are used as an 8-bit prescaler number N.
    - The timer counter is decremented every $N + 1$ clock cycles, so if the prescaler is left blank then the counter changes at the base clock rate of $200\,\mathrm{MHz}$.
    - Essentially, having $N > 0$ in the prescaler means the counting frequency is reduced by a factor of $N + 1$.
- The fourth word is a **status register**. This uses only the lowest bit as the **timeout** flag: when bit 0 is set to 1, the timer has finished counting down. Somewhat confusingly, to clear the timeout flag you need to write 1 to this register.

[14] Like the *interval timer*, if bit 2 is set to 1 then interrupts *are* generated. This is more typical behaviour for an interrupt control flag.

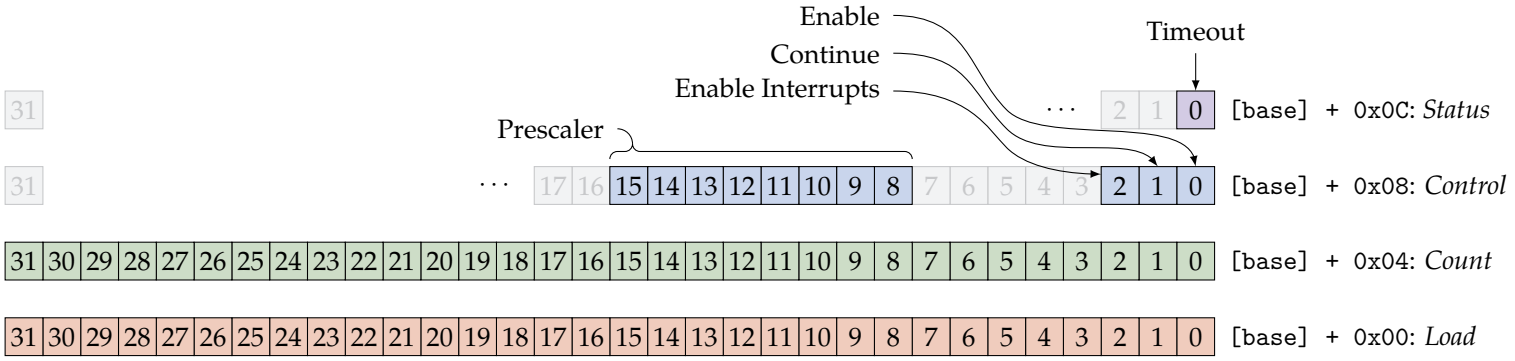

Figure 7: Structure of the A9 private timers in the ARM®Cortex-A9. The base address is `0xFFFEC600` for both, each CPU core can only access their own timer.

The *A9 private timer* is arguably even simpler to use than the *HPS timer*, so it is the one we recommend using for the labs. The only thing to watch out for is if you want to use the prescaler to change the clock rate, make sure you don't accidentally clear it when starting/stopping the timer.

Assembly Code for Controlling the Interval Timer

Example: This is how to initialize the *interval timer* to count an interval of $5\,\mathrm{s}$:

```
@ initialize the timer
ldr r0, =0xFF202000 @timer address
@ to set the timer for 5 s
ldr r2, =500000000
str r2, [r0, #8] @write interval to timer
     @low-period register
     @only lowest-16 bits will be written
mov r2, r2, lsr #16 @shift right by 16 bits
str r2, [r0, #12 ] @write rest of interval
     @to timer high-period
```

Example: We want to use the timer to measure a $5\,\mathrm{s}$ interval, and also sequentially light up LEDs for each $1\,\mathrm{s}$ subinterval. After the $5\,\mathrm{s}$ period is over, the LED bank should be reset and the process should repeat.

A working implementation is given below. Those of you who have a shred of skill at coding will recognize that the example below is *not* a particularly good solution to the problem. For one thing, it would be easier (and probably better coding practice too) to have the timer just count a $1\,\mathrm{s}$ interval and use a separate counter to count to 5 — but then that would not demonstrate how to read the current count from the timer.

```
.global _start
_start:
@ get the addresses
ldr r4, =0xFF202000
ldr r5, =0xFF200000

@ initialize the 5 s count
ldr r0, =500000000
str r0, [r4, #8]
mov r1, r0, lsr #16
str r1, [r4, #12]
@ start the timer for continuous
@ counting (0b0110)
mov r1, #6
str r1, [r4, #4]
@ get the 1 s interval
ldr r1, =100000000
@ initialize LED pattern
mov r8, #0
str r8, [r5]

@ main loop
main_loop:
  @ get the current count
  @ first write junk to counter
  str r1, [r4, #16]
  @ now get the low count
```

```
ldr r2, [r4, #16]
@ get the high count
ldr r3, [r4, #20]
@ combine them
add r2, r3, lsl #16
@ check if current count has
@ passed 1 s
cmp r2, r0
bhi main_loop
@ increment LED pattern
lsl r8, #1
add r8, #1
str r8, [r5]
@ check if we have passed 5 s
@ #63 is bit pattern 111111
cmp r8, #63
@ if not, subtract 1 s from interval
@ and loop back
sublo r0, r1
blo main_loop
@ if we passed 5 s, reset interval
ldr r0, =500000000
@ blank LEDs
mov r8, #0
str r8, [r5]
b main_loop
```

Example: Just turn the last LED on and off at a 1 s interval. This example shows how to check for timeouts instead of how to read the current count.

```
.global _start
_start:
@ get the addresses
ldr r0, =0xFF200000
ldr r1, =0xFF202000
@ last LED is bit 9, 2^9 = 0x200
ldr r2, =0x200
@ turn on LED
str r2, [r0]

@ clear any timeout
mov r3, #0
str r3, [r1]
@ set the count interval
ldr r3, =100000000
@ write to low period register
str r3, [r1, #8]
lsr r3, #16
@ write to high period register
str r3, [r1, #12]
@ start the timer
mov r3, #0b0110
str r3, [r1, #4]
```

```
@main loop
loop:
  @ toggle LED
  str r2, [r0]
@ loop until timer counts down
@ by checking for timeout in status register
wait:
  ldr r3, [r1]
  ands r3, #1
  beq wait
  @ clear timeout flag
  str r3, [r1]
  @ toggle LED pattern
  eor r2, #0x200
  b loop
```

Note the use of `ands r3, #1` instead of `cmp r3, #1` to check for a timeout. Using `ands r3, #1` checks only bit 0 (the timeout flag), while `cmp r3, #1` will only work if all other bits in the status register are zero. [15]

[15] They aren't, bit 1 is the "running" flag which should be set. Furthermore, the remaining 30 bits are unassigned but may not be zero — the timer is a piece of external hardware and we don't always know how it is initialized.

C Code for Controlling the Interval Timer

Example: The memory-mapped registers of the *interval timer* are easier to deal with as a structure.

```
#define TIMER_1_BASE  0xFF202000
typedef struct _interval_timer
{
  int status;
  int control;
  int low_period;
  int high_period;
  int low_counter;
  int high_counter;
} interval_timer;

volatile interval_timer* const timer_1
          = (interval_timer*)TIMER_1_BASE;
```

Example: This is how to initialize the timer, using the structure defined above, to count an interval of 5 s:

```
// interval for counting 5 s
int interval = 500000000;
// write to low period
// only lowest-16 bits will be written
timer_1->low_period = interval;
// write to high period
timer_1->high_period = interval >> 16;
```

Example: We want to use the timer to measure a 5 s interval, and also sequentially light up LEDs for each 1 s subinterval. After the 5 s period is over, the LED bank should be reset and the process should repeat. As mentioned above in the Assembly language implementation of this program, this program uses a silly way of doing things, just to demonstrate the timer functionality. This code using the timer structure defined previously, and assumes the timer has already been initialized as described above.

```
#define LED_BASE      0xFF200000
#define TIMER_BASE  0xFF202000

typedef struct _interval_timer
{
  int status;
  int control;
  int low_period;
  int high_period;
  int low_counter;
  int high_counter;
} interval_timer;

void main()
{
  // pointers to hardware
  volatile interval_timer* const timer_1
            = (interval_timer*)TIMER_BASE;
```

```
volatile int* const led = (int*)LED_BASE;

// interval for counting 5 s
int interval = 500000000;
// keeping track of the count
int current_count = 0;
int last_count = interval;
// pattern for the LEDs
int led_pattern = 0;

// initialize timer for 5 s interval
// write to low period
// only lowest-16 bits will be written
timer_1->low_period = interval;
// write to high period
timer_1->high_period = interval >> 16;

// start timer for continuous counting
// 6 is(0b0110)
timer_1->control = 6;
// blank out LEDs
*led = led_pattern;

// main loop
while(1)
{
  // write junk to get current count
  timer_1->low_counter = 1;
```

```
    // now get updated count
    current_count = timer_1->low_counter
         + (timer_1->high_counter << 16);
    // check if 1s has passed
    if (current_count <= last_count)
    {
      //increment led pattern
      led_pattern = (led_pattern << 1) + 1;
      *led = led_pattern;
      // check if we have passed 5 s
      // 63 is 0b111111
      if ( led_pattern == 63 )
      {
        //blank LEDs
        led_pattern = 0;
        *led = led_pattern;
        // reset last count to 5 s
        last_count = interval;
      }
      else
      {
        //subtract 1 s from interval
        last_count -= 100000000;
      }
    }
  }
}
```

Example: Just turn the last LED on and off at a 1 s interval. This example shows how to check for timeouts instead of how to read the current count. Unlike the previous example, this version of code uses pointer arithmetic to access different registers in timer instead of defining a structure. Personally, I prefer using structures but both methods work.

```
#define LED_BASE   0xFF200000
#define TIMER_BASE 0xFF202000

int main(void)
{
  // pointers to hardware
  volatile int* const led
      = (int*)LED_BASE;
  volatile int* const timer
      = (int*)TIMER_BASE;
  // turn on last LED
  int led_pattern = 0x200;
  // interval for counting 1 s
  int counter = 100000000;
  // turn LEDs on
  *led = led_pattern;
  // clear timeout flag
  *timer = 0;
  // set low period
  *(timer + 2) = counter;
  // set high period
```

```
  *(timer + 3) = counter >> 16;
  // start timer
  *(timer + 1) = 0b0110;

  // main loop
  while(1)
  {
    // write to LEDs
    *led = led_pattern;
    // wait until timer counts down
    while((*timer & 1) == 0)
      ;
    // clear timeout flag
    *timer = 0;
    // toggle LED pattern
    led_pattern ^= 0x200;
  }
}
```

Notice that because the `timer` pointer was defined as a pointer to an `int`, `(timer+1)` points to an address one `int` (i.e., one word) beyond the address of `timer`.

- Pointer arithmetic is therefore similar to the addressing modes in Assembly, except the address increment depends on the *type* of pointer, whereas in Assembly the address increment was always in bytes.

Assembly Code for Controlling the A9 Private Timer

Example: This is how to initialize the *A9 private timer* to count an interval of $5\,\mathrm{s}$:

```
@ initialize the timer
ldr r0, =0xFFFEC600 @timer address
@ to set the timer for 5 s
ldr r2, =1000000000
str r2, [r0] @write interval to timer
     @load register
```

Remember that the *A9 private timer* has a $200\,\mathrm{MHz}$ clock, I could use the prescaler to reduce the speed of the timer, but instead I chose to load a larger count interval than I used previously for the *interval timer*.

Example: We want to use the timer to measure a $5\,\mathrm{s}$ interval, and also sequentially light up LEDs for each $1\,\mathrm{s}$ subinterval. After the $5\,\mathrm{s}$ period is over, the LED bank should be reset and the process should repeat. In this example I will use the prescaler, since I am only interacting with the control register of the timer once (as it is set to count continuously). A working implementation is given below.

```
.global _start
_start:
@ get the addresses
ldr r4, =0xFFFEC600
ldr r5, =0xFF200000
```

```
@ initialize the 5 s count
@ this assumes a 100 MHz rate
ldr r0, =500000000
str r0, [r4]

@ start the timer for continuous
@ counting with prescaler
@ first set prescaler to 1
@ clock is divided by 1+1=2
mov r1, #1
@ shift this over as prescaler is
@ bits 7 to 15
lsl r1, #8
@ now include control for continuous
@ counting (0b011)
add r1, #3
str r1, [r4, #8]

@ get the 1 s interval
@ also assumes a 100 MHz rate
ldr r1, =100000000

@ initialize LED pattern
mov r8, #0
str r8, [r5]

@ main loop
```

```
main_loop:
  @ get the current count
  ldr r2, [r4, #4]
  @ check if current count has
  @ passed 1 s
  cmp r2, r0
  bhi main_loop
  @ increment LED pattern
  lsl r8, #1
  add r8, #1
  str r8, [r5]
  @ check if we have passed 5 s
  @ #63 is bit pattern 111111
  cmp r8, #63
  @ if not, subtract 1 s from interval
  @ and loop back
  sublo r0, r1
  blo main_loop
  @ if we passed 5 s, reset interval
  @ again assumes 100 MHz rate
  ldr r0, =500000000
  @ blank LEDs
  mov r8, #0
  str r8, [r5]
  @ loop back to start
  b main_loop
```

Example: Just turn the last LED on and off at a 1 s interval. This example shows how to check for timeouts instead of how to read the current count. Note that this code also does not use the prescaler, so the timer counts for 200 000 000 cycles at 200 MHz for a 1 s interval.

```
.global _start
_start:
@ get the addresses
ldr r0, =0xFF200000
ldr r1, =0xFFFEC600
@ last LED is bit 9, 2^9 = 0x200
ldr r2, =0x200
mov r4, r2
@ turn on LED
str r4, [r0]

@ set the count interval
ldr r3, =200000000
str r3, [r1]
@ start timer
mov r3, #0b011
str r3, [r1, #8]

@main loop
loop:
  @toggle LED
  str r2, [r0]
```

```
@ loop until timer counts down
@ by checking for timeout in status register
wait:
  ldr r3, [r1, #12]
  cmp r3, #0
  beq wait
  @ clear timeout flag
  str r3, [r1, #12]
  @ toggle LEDs
  eor r2, #0x200
  b loop
```

C Code for Controlling the A9 Private Timer

Example: The memory-mapped registers of the *A9 private timer* are easier to deal with as a structure.

```
#define TIMER_A9_BASE  0xFFFEC600
typedef struct _a9_timer
{
  int load;
  int count;
  int control;
  int status;
} a9_timer;

volatile a9_timer* const timer_1
          = (a9_timer*)TIMER_A9_BASE;
```

Example: This is how to initialize the timer, using the structure defined above, to count an interval of $5\,\mathrm{s}$. Again, I will use the prescaler in this example.

```
// interval for counting 5 s
// assuming 100 MHz rate
int interval = 500000000;
// load interval
timer_1->load = interval;
```

Example: We want to use the timer to measure a 5 s interval, and also sequentially light up LEDs for each 1 s subinterval. After the 5 s period is over, the LED bank should be reset and the process should repeat. This also uses the prescaler.

```
#define LED_BASE       0xFF200000
#define TIMER_A9_BASE 0xFFFEC600

typedef struct _a9_timer
{
  int load;
  int count;
  int control;
  int status;
} a9_timer;

void main()
{
  // pointers to hardware
  volatile a9_timer* const timer_1
          = (a9_timer*)TIMER_A9_BASE;
  volatile int* const led = (int*)LED_BASE;

  // interval for counting 5 s
  int interval = 500000000;
  // keeping track of the count
  int current_count = 0;
  int last_count = interval;
```

```
// pattern for the LEDs
int led_pattern = 0;

// initialize timer for 5 s interval
// assumes 100 MHz rate
timer_1->load = interval;

// start timer for continuous counting
// 3 is (0b011) for control
// (1 << 8) is for prescaler
timer_1->control = 3 + (1<<8);
// blank out LEDs
*led = led_pattern;

// main loop
while(1)
{
  // get count
  current_count = timer_1->count;
  // check if 1s has passed
  if (current_count <= last_count)
  {
    //increment led pattern
    led_pattern = (led_pattern << 1) + 1;
    *led = led_pattern;
    // check if we have passed 5 s
    // 63 is 0b111111
    if ( led_pattern == 63 )
```

```
        {
          //blank LEDs
          led_pattern = 0;
          *led = led_pattern;
          // reset last count to 5 s
          last_count = interval;
        }
        else
        {
          //subtract 1 s from interval
          last_count -= 100000000;
        }
      }
    }
}
```

Example: Just turn the last LED on and off at a 1 s interval. This example shows how to check for timeouts instead of how to read the current count. As with the interval timer,this version of code uses pointer arithmetic to access different registers in timer instead of defining a structure.

```
#define LED_BASE   0xFF200000
#define TIMER_BASE 0xFFFEC600

int main(void)
{
  // pointers to hardware
  volatile int* const led
      = (int*)LED_BASE;
  volatile int* const timer
      = (int*)TIMER_BASE;
  // turn on last LED
  int led_pattern = 0x200;
  // interval for counting 1 s
  int counter = 200000000;
  // turn LEDs on
  *led = led_pattern;

  // set period
  *timer = counter;
  // start timer
  *(timer + 2) = 0b011;
```

```
  // main loop
  while(1)
  {
    // write to LEDs
    *led = led_pattern;
    // wait until timer counts down
    while(*(timer + 3) == 0)
      ;
    // clear timeout flag
    *(timer + 3) = 1;
    // toggle LED pattern
    led_pattern ^= 0x200;
  }
}
```

Note that this code assumes that only bit 0 of the status register is changed to reflect the timeout status, and all other 31 bits always remain zero. This is probably the case with bits that are unassigned in a peripheral's register, but you should be careful.

- In particular, there can a difference between bits that are "unassigned", bits that are "unused", and bits that are "reserved".
- In all cases these are bits that you are not supposed to use when interacting with the peripheral.
- Bits that are "unassigned" don't physically exist, so you can probably safely assume they are treated as zeros by

the CPU.

- Bits that are "unused" are *probably* zeros (and also may not physically exist), but it might be hardware specific.
- Bits that are "reserved" *might* be always zero, but also *might* be used by the hardware and consequently — from the CPU's perspective — have random values.

In short: whenever you are in doubt, or whenever you code isn't responding to a peripheral as expected, *make no assumptions about the properties of the peripheral* and use bitmasks to purposefully isolate only the bits you are interested in.