Learn the architecture - Generic Timer

Version 1.0

Non-Confidential

Issue 02

102379_0100_02_en

Learn the architecture - Generic Timer

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	13 August 2019	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview 6
2. Before you begin 7
3. What is the Generic Timer? 8
4. The processor timers 10
5. System Counter 17
6. External timers 21
7. Example using Arm Development Studio 23
8. Check your knowledge 26
9. Related information 27
10. Next steps 28

1. Overview

This guide introduces the Generic Timer, the timer framework for A-profile PEs. The guide introduces the different components of the timer framework within a modern SoC and covers the programming interfaces that are available to software.

The guide is targeted at developers writing low-level software to initialize or use the timers in an Arm-based system. Users of this guide will usually be working on low-level code.

At the end of this guide, you can Check your knowledge. You will have learned the names and purposes of the different components that make up the timer sub-system. You will be able to write code to set up the timers in a bare metal environment. You will also be able to describe which timers are present, based on the implemented architectural features.

2. Before you begin

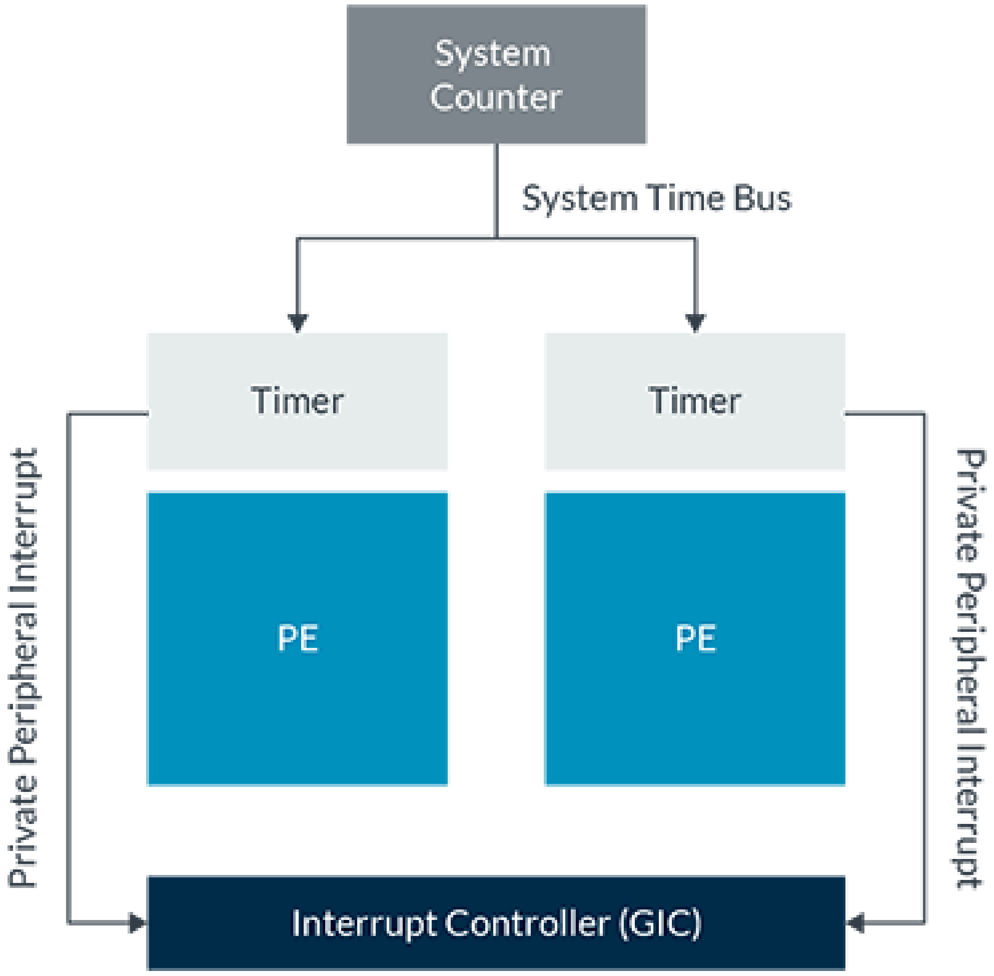
We assume that you are familiar with the Arm exception model. If you are not, you might want to first read our Arm v8-A Exception model guide.

This guide includes a short code example written in C and assembler. If you are unfamiliar with Arm assembler syntax, you can review our Armv8-A Instruction Set Architecture (ISA) guide for a brief introduction. The example requires Arm Development Studio. If you do not already have a copy, you can download an evaluation copy.

3. What is the Generic Timer?

The Generic Timer provides a standardized timer framework for Arm cores. The Generic Timer includes a System Counter and set of per-core timers, as shown in the following diagram:

Figure 3-1: System counter


System Counter
System Time Bus
Timer
Timer
PE
PE
Private Peripheral Interrupt
Private Peripheral Interrupt
Interrupt Controller (GIC)

The System Counter is an always-on device, which provides a fixed frequency incrementing system count. The system count value is broadcast to all the cores in the system, giving the cores a common view of the passage of time. The system count value is between 56 bits and 64 bits in width, with a frequency typically in the range of 1MHz to 50MHz.

Note

The Generic Timer only measures the passage of time. It does not report the time or date. Usually, an SoC would also contain a Real-Time Clock (RTC) for time and date.

Each core has a set of timers. These timers are comparators, which compare against the broadcast system count that is provided by the System Counter. Software can configure timers to generate interrupts or events in set points in the future. Software can also use the system count to add timestamps, because the system count gives a common reference point for all cores.

In this guide, we will explain the operation and configuration of both the timers and the System Counter.

4. The processor timers

This table shows the processor timers:

Timer name	When is the timer present?
EL1 physical timer	Always
EL1 virtual timer	Always
Non-secure EL2 physical timer	Implements EL2
Non-secure EL2 virtual timer	Implements ARMv8.1-VHE
EL3 physical timer	Implements EL3
Secure EL2 physical timer	Implements ARMv8.4-SecEL2
Secure EL2 virtual timer	Implements ARMv8.4-SecEL2

Count and frequency

The `CNTPCT_EL0` system register reports the current system count value.

Reads of `CNTPCT_EL0` can be made speculatively. This means that they can be read out of order regarding the program flow. This could be important in some cases, for example comparing timestamps. When the ordering of the counter read is important, an `ISB` can be used, as the following code shows:

```
loop:           // Polling for some communication to indicate a requirement to read
                // the timer
  LDR X1, [X2]
  CBZ x1, loop
  ISB           // Without this, the CNTPCT could be read before the memory location
 in
                // [X2] has had the value 0 written to it
  MRS X1, CNTPCT_EL0
```

`CNTFRQ_EL0` reports the frequency of the system count. However, this register is not populated by hardware. The register is write-able at the highest implemented Exception level and readable at all Exception levels. Firmware, typically running at EL3, populates this register as part of early system initialization. Higher-level software, like an operating system, can then use the register to get the frequency.

Timer registers

Each timer has the following three system registers:

Register	Purpose
`<timer>_CTL_EL<x>`	Control register
`<timer>_CVAL_EL<x>`	Comparator value
`<timer>_TVAL_EL<x>`	Timer value

In the register name, `<timer>` identifies which timer is being accessed. The following table shows the possible values:

Timer	Register prefix	EL<x>
EL1 physical timer	CNTP	EL0
EL1 virtual time	CNTV	EL0
Non-secure EL2 physical timer	CNTHP	EL2
Non-secure EL2 virtual timer	CNTHV	EL2
EL3 physical timer	CNTPS	EL1
Secure EL2 physical timer	CNTHPS	EL2
Secure EL2 virtual timer	CNTHVS	EL2

For example, `CNTP_CVAL_EL0` is the Comparator register of the EL1 physical timer.

Test yourself

What is the name of the control register for the EL3 physical timer and Non-secure EL2 virtual timer?

Accessing the timers

For some timers, it is possible to configure which Exception levels can access the timer:

EL1 Physical and Virtual Timers: EL0 access to these timers is controlled by `CNTKCTL_EL1`. EL2 Physical and Virtual Timers: When `HCR_EL2.{TGE,E2H}=={1,1}`, EL0 access to these timers is controlled by `CNTKCTL_EL2`. These timers were added as part of the support for the Armv8.1-A Virtualization Host Extension, which is beyond the scope of this guide EL3 physical timer: S.EL1 and S.EL2 access to this timer is controlled by `SCR_EL3.ST`.

Configuring a timer

There are two ways to configure a timer, either using the comparator (`CVAL`) register, or using the timer (`TVAL`) register.

The comparator register, `CVAL`, is a 64-bit register. Software writes a value to this register and the timer triggers when the count reaches, or exceeds, that value, as you can see here:

```
Timer Condition Met: CVAL <= System Count
```

The timer register, `TVAL`, is a 32-bit register. When software writes `TVAL`, the processor reads the current system count internally, adds the written value, and then populates `CVAL`:

```
CVAL = TVAL + System Counter
Timer Condition Met: CVAL <= System Count
```

You can see this populating of `CVAL` in software. If you read the current system count, write 1000 to `TVAL`, and then read `CVAL`, you will see that `CVAL` is approximately 1000 + system count. The count is approximate, because time will move on during the instruction sequence.

Reading `TVAL` back will show it decrementing down to 0, while the system count increments. `TVAL` reports a signed value, and will continue to decrement after the timer fires, which allows software to determine how long ago the timer fired. `TVAL` and `CVAL` gives software two different models for how to use the timer. If software needs a timer event in X ticks of the clock, software can write X to `TVAL`. Alternatively, if software wants an event when the system count reaches Y, software can write Y to `CVAL`.

Remember that `TVAL` and `CVAL` are different ways to program the same timer. They are not two different timers.

Interrupts

Timers can be configured to generate an interrupt. The interrupt from a core's timer can only be delivered to that core. This means that the timer of one core cannot be used to generate an interrupt that targets a different core.

The generation of interrupts is controlled through the `CTL` register, using these fields:

- `ENABLE` - Enables the timer.
- `IMASK` - Interrupt mask. Enables or disables interrupt generation.
- `ISTATUS` - When `ENABLE==1`, reports whether the timer is firing (`Cval <= System Count`).

To generate an interrupt, software must set `ENABLE` to 1 and clear `IMASK` to 0. When the timer fires (`CVAL <= System Count`), an interrupt signal is asserted to the interrupt controller. In Armv8-A systems, the interrupt controller is usually a Generic Interrupt Controller (GIC).

The interrupt ID (INTID) that is used for each timer is defined by the Server Base System Architecture (SBSA), shown here:

Timer	SBSA recommended INTID
EL1 Physical Timer	30
EL1 Virtual Timer	27
Non-secure EL2 Physical Timer	26
Non-secure EL2 Virtual Timer	28
EL3 Physical Timer	29
Secure EL2 Physical Timer	20
Secure EL2 Virtual Timer	19

Note

These INTIDs are in the Private Peripheral Interrupt (PPI) range. These INTIDs are private to a specific core. This means that each core sees its EL1 physical timer as INTID 30. This is described in more detail in our Generic Interrupt Controller guide.

The interrupts generated by the timer behave in a level-sensitive manner. This means that, once the timer firing condition is reached, the timer will continue to signal an interrupt until one of the following situations occurs:

- `IMASK` is set to one, which masks the interrupt.
- `ENABLE` is cleared to 0, which disables the timer.
- `TVAL` or `CVAL` is written, so that firing condition is no longer met.

When writing an interrupt handler for the timers, it is important that software clears the interrupt before deactivating the interrupt in the GIC. Otherwise the GIC will re-signal the same interrupt again.

The operation and configuration of the GIC is beyond the scope of this guide.

Timer virtualization

Earlier, we introduced the different timers that are found in a processor. These timers can be divided into two groups: virtual timers and physical timers.

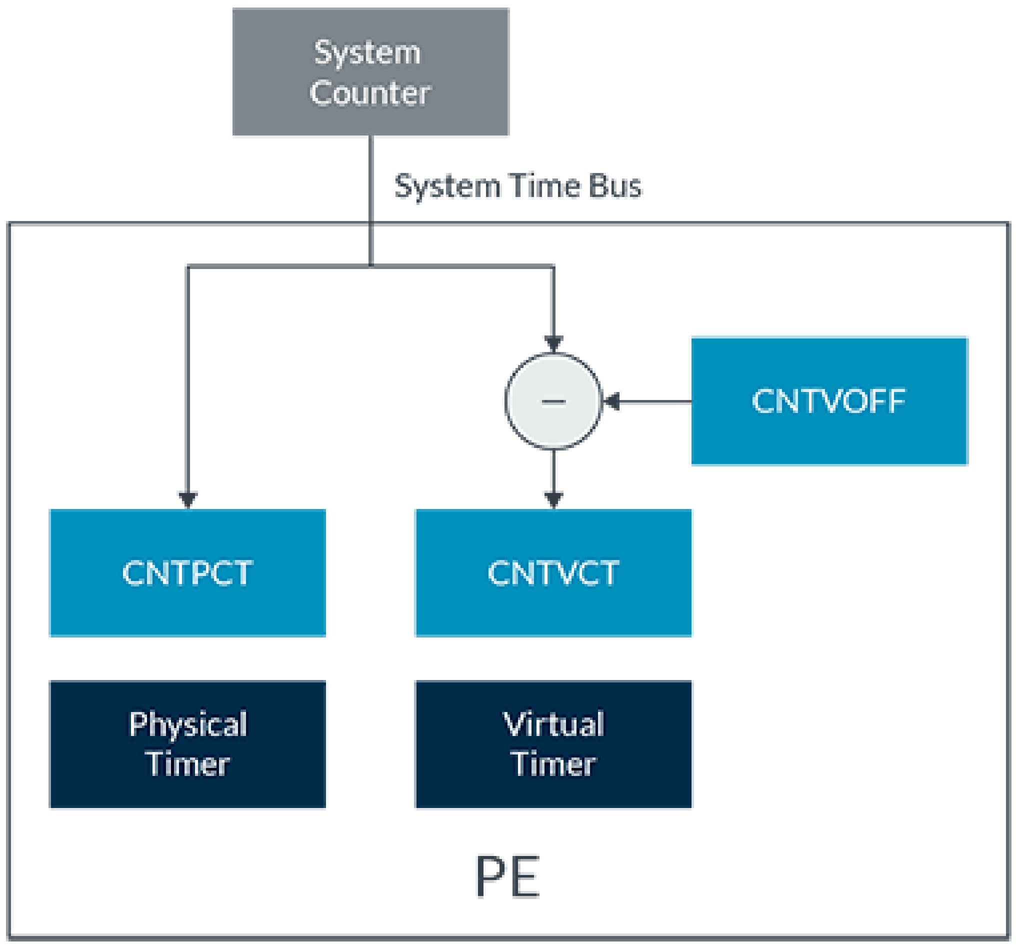
Physical timers, like the EL3 physical timer, `CNTPS`, compare against the count value provided by the System Counter. This value is referred to as the physical count and is reported by `CNTPCT_EL0`.

Virtual timers, like the EL1 Virtual Timer, `CNTV`, compare against a virtual count. The virtual count is calculated as:

```
Virtual Count = Physical Count - <offset>
```

The offset value is specified in the register `CNTVOFF_EL2`, which is only accessible at EL2 or EL3. This configuration is shown in the following diagram:

Figure 4-1: System counter



Note

If EL2 not implemented, the offset is fixed as 0. This means that the virtual and physical count values are always the same.

The virtual count allows a hypervisor to show virtual time to a Virtual Machine (VM). For example, a hypervisor could use the offset to hide the passage of time when the VM was not scheduled. This means that the virtual count can represent time experienced by the VM, rather than wall clock time.

Event stream

The Generic Timer can also be used to generate an event stream as part of the Wait for Event mechanism. The `WFE` instruction puts the core into a low power state, with the core woken by an event.

Details about the `WFE` mechanism are beyond the scope of this guide.

There are several ways to generate an event, including:

- Executing the `SEV` (Send Event) instruction on a different core
- Clearing the Global Exclusive Monitor of the core
- Using the Event stream from the core's the Generic Timer

The Generic Timer can be configured to generate a stream of events at a regular interval. One use for this configuration is to generate a timeout. `WFE` is typically used when waiting for a resource to become available, when the wait is not expected to be long. The event stream from the timers means that the maximum time that the core will stay in the low power state is bounded.

An event stream can be generated from the physical count, `CNTPCT_EL0`, or from the virtual count, `CNTPVT_EL0`:

`CNTKCTL_EL1` - Controls event stream generation from `CNTVCT_EL0` `CNTKCTL_EL2` - Controls event stream generation from `CNTPCT_EL0`

For each register, the controls are:

- `EVNTEN` Enables or disables the generation of events
- `EVNTI` Controls the rate of events
- `EVNTDIR` Controls when the event is generated

The control `EVNTI` specifies a bit position in the range 0 to 15. When the bit at the selected position changes, an event is generated. For example, if `EVNTI` is set to 3 then an event is generated when bit[3] of the count changes.

The control `EVNTDIR` controls whether the event is generated when the selected bit transitions from 1-to-0 or from 0-to 1.

Summary table

This table summarizes the information about the different timers discussed in this section:

Timer	Registers	Typically used by	Trappable?	Using counter	INTID
EL1 Physical Timer	`CNTP_<>_EL0`**	EL0 and EL1	To EL2	`CNTPCT_EL0`	30
EL2 Non-secure Physical Timer	`CNTHP_<>_EL2`	NS.EL2		`CNTPCT_EL0`	26
EL2 Secure Physical Timer	`CNTHPS_<>_EL2`	S.EL2		`CNTPCT_EL0`	20
EL3 Physical Timer	`CNTPS_<>_EL1`	S.EL1 and EL3	To EL3	`CNTPCT_EL0`	29
EL1 Virtual Timer	`CNTV_<>_EL0`**	EL0 and EL1		`CNTPCT_EL0`	27
EL2 Non-secure Virtual Timer	`CNTHV_<>_EL2`	NS.EL2		`CNTPCT_EL0`	28
EL2 Secure Virtual Timer	`CNTHVS_<>_EL2`	S.EL2		`CNTPCT_EL0`*	19

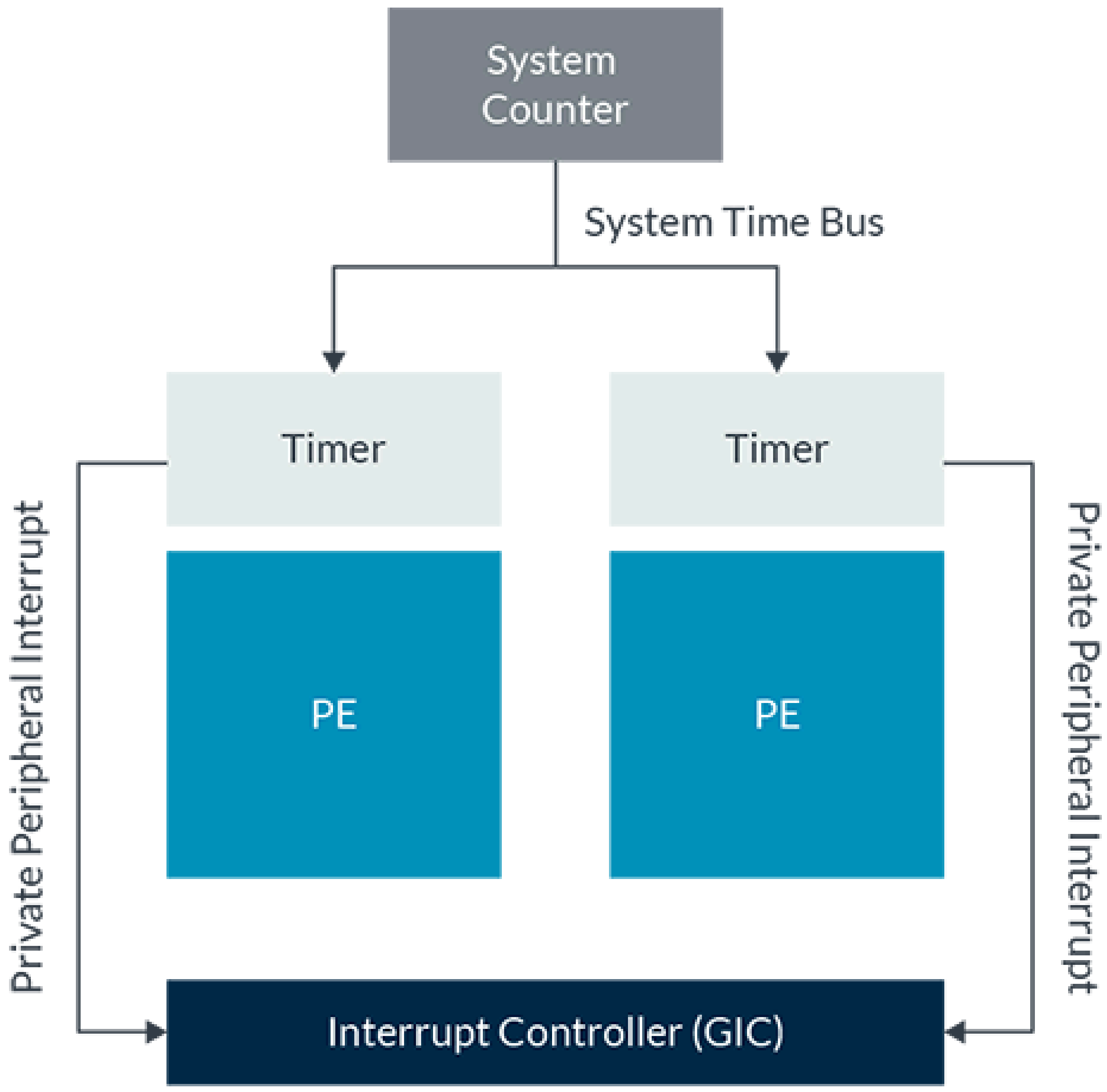
*For these timers, the virtual offset (`CNTVOFFSET_EL2`) always behaves as 0. Therefore, although these timers compare against the virtual count value, they are in practice using the physical counter value.

** Subjects to re-direction when `HCR_EL2.E2H==1`.

5. System Counter

In What is the Generic Timer?, we introduced the System Counter. The System Counter generates the system count value that is distributed to all the cores in the system, as shown in the following diagram:

Figure 5-1: System counter


System Counter
System Time Bus
Timer
Timer
PE
PE
Private Peripheral Interrupt
Private Peripheral Interrupt
Interrupt Controller (GIC)

The SoC implementer is responsible for the design of the System Counter. Usually, the System Counter requires some initialization when a system boots up. Arm provides a recommended register interface for the System Counter, but you should check with your SoC implementer for details of a specific implementation.

One physical system count value broadcasts to all cores. This means that all cores share the same view of the passing of time. Consider the following example:

- Device A reads the current system count and adds it to a message as a timestamp, then sends the message to Device B.
- When Device B receives the message, it compares the timestamp to the current system count.

In this example, the system count value that is seen by Device B can never be earlier than the timestamp in the message.

The System Counter measures real time. This means that it cannot be affected by power management techniques like Dynamic Voltage and Frequency Scaling (DVFS) or putting cores into a lower power state. The count must continue to increment at its fixed frequency. In practice, this requires the System Counter to be in an always-on power domain.

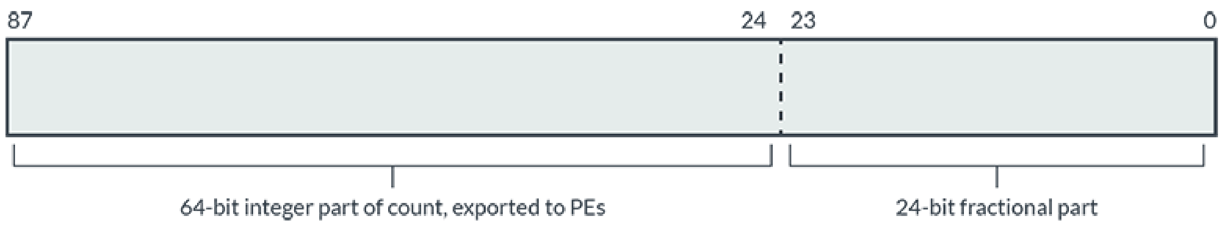
To save power, the System Counter can vary the rate at which it updates the count. For example, the System Counter could update the count by 10 every 10th tick of the clock. This can be useful when the connected cores are all in low power state. The system count still needs to reflect time advancing, but power can be saved by broadcasting fewer counter updates.

Counter scaling

The option to scale the system count was introduced in Armv8.4-A. Instead of incrementing by one on every tick of the clock, the count can increment by X, where X is configured by software during system initialization. This feature allows the count to effectively increment faster or slower than the frequency of the counter.

To support scaling, the System Counter internally expands the counter value to 88 bits, as you can see in the following diagram:

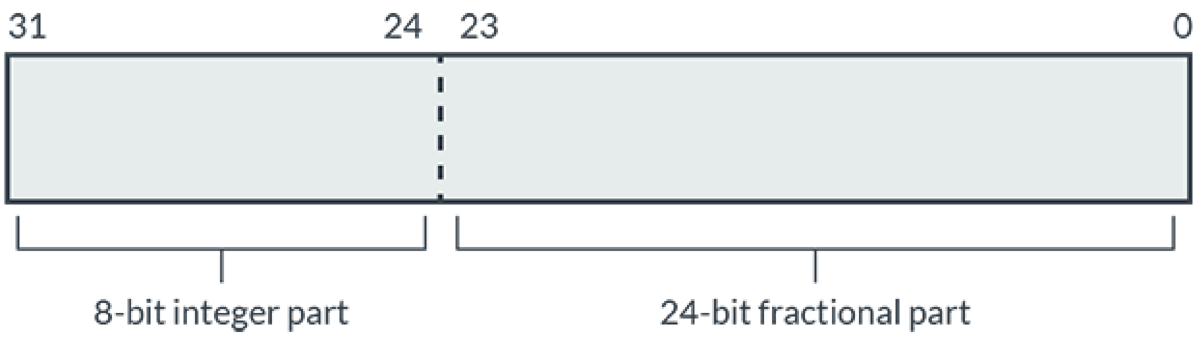
Figure 5-2: bit integer part of count image



The count is represented as an 88-bit fixed point number, with 64 bits for the integer part and 24 bits for the fractional part. The integer portion of the count is what is reported by `CNTPCT_EL0` on the connected processors. The fractional part is used internally by the System Counter.

The increment amount comes from a 32-bit register called `CNTSCR`, and its format is shown below:

Figure 5-3: bit integer part



The increment value is split into an integer part of 8 bits and a fractional part of 24 bits.

When scaling is enabled, on every tick the count is incremented by the value in `CNTSCR`. For example, if `CNTSCR` is set to `0x0180_0000`, that means that the count increments by 1.5 (integer part `0x01`, fraction part `0x80_0000`) on every tick. This is illustrated in the following table:

Tick	Internal counter value Integer part / Fractional part	Exported counter value (Visible via `CNTPCT_EL0`)
0	`0x0000_0000_0000_0000_0000_00`	`0x0000_0000_0000_0000`
1	`0x0000_0000_0000_0001_8000_00`	`0x0000_0000_0000_0001`
2	`0x0000_0000_0000_0003_0000_00`	`0x0000_0000_0000_0003`
3	`0x0000_0000_0000_0004_8000_00`	`0x0000_0000_0000_0004`
4	`0x0000_0000_0000_0006_0000_00`	`0x0000_0000_0000_0006`
5	`0x0000_0000_0000_0007_8000_00`	`0x0000_0000_0000_0007`
6	`0x0000_0000_0000_0009_0000_00`	`0x0000_0000_0000_0009`

Scaling can only be configured while the System Counter is disabled. Changing whether scaling is enabled, or the scaling factor, while the counter is running can result in unknown count values being returned.

Basic programming

The guidance in this section assumes that the System Counter implements the recommended Arm register interface.

The System Counter provides two register frames: `CNTControlBase` and `CNTReadBase`.

The register frame `CNTControlBase` is used to configure the System Counter and is Secure access only on systems that support TrustZone. The registers in this frame are shown in the following table:

Register	Description
CNTCR	Control register, includes: • Counter enable • Counter scaling enable (Armv8.4-A or later) • Update frequency selection • Halt-on-debug control. Stops the counter from incrementing when requested by debugger.
CNTSCR	Increment value when using scaling (Armv8.4-A or later)
CNTID	ID register, reports which features are implemented.
CNTSR	Status register. Reports whether the timer is running or stopped.
CNTCV	Reports the current count value. Returns only the integer portion of the count.
CNTFID<n>	Reports the available update frequencies.

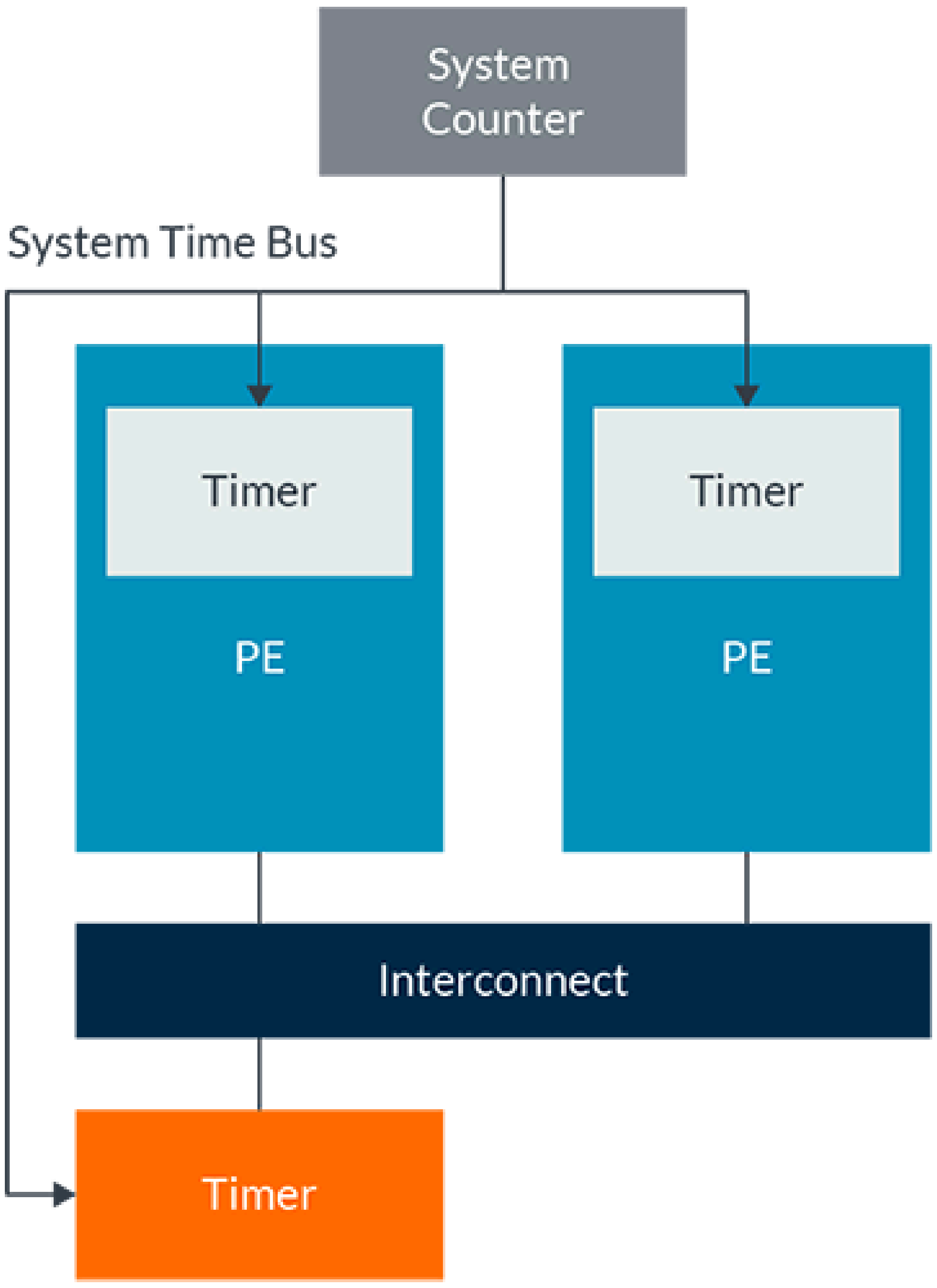
To enable the System Counter, software must select an update frequency and set the counter enable.

`CNTReadBase` is a copy of `CNTControlBase` that only includes the `CNTCV` register. This means that `CNTReadBase` only reports the current system count value. However, unlike `CNTControlBase`, `CNTReadBase` is accessible to Non-secure accesses. This means that Non-secure software can read the current count, but cannot otherwise configure the System Counter.

6. External timers

In What is the Generic Timer?, we introduced the timers that are in the processor. A system can also contain additional external timers. The following diagram shows an example of this:

Figure 6-1: Example additional external timer



The programming interface for these timers mirrors that of the internal timers, but these timers are accessed via memory-mapped registers. The location of these registers is determined by the SoC implementor, and should be reported in the datasheet for the SoC that you are working with.

Interrupts from the external memory-mapped timers will typically be delivered as Shared Peripheral Interrupts (SPIs) by the GIC.

7. Example using Arm Development Studio

This section of the guide includes a short example, downloadable as a zip file, to demonstrate the configuration of the System Counter and the generation of interrupts using `TVAL` and `CVAL`.

The example requires Arm Development Studio. If you do not already have a copy of Arm Development Studio, you can download an evaluation copy.

The example includes a `ReadMe.txt` file which lists the included files, and instructions for building and running the example.

Within `main.c` is the code for configuring the Timers. Going through `main()`, it starts with:

```
//
// Configure the interrupt controller
//
rd = initGIC();

// Secure Physical Timer (INTID 29)
setIntPriority(29, rd, 0);
setIntGroup(29, rd, 0);
enableInt(29);

// Non-secure EL1 Physical Timer (INTID 30)
setIntPriority(30, rd, 0);
setIntGroup(30, rd, 0);
enableInt(30, rd);
```

The previous code configures the GIC. The operation of the GIC is beyond the scope of this guide, but configuring the GIC is necessary to generate timer interrupts.

The function `initGIC()` performs top-level initialization of the interrupt controller. The following calls configure and enable the interrupt sources associated with the Secure physical timer and Non-secure EL1 physical timer. Each interrupt is configured as follows:

- Group 0. This means the interrupt will be signaled as a FIQ.
- Priority 0. This is the highest priority value in the GIC architecture.
- Enabled. This allows the interrupt to be signaled to the core.

Next, the System Counter is initialized, as shown here:

```
//
// Configure and enable the System Counter
//
setSystemCounterBaseAddr(0x2a430000); // Address of the System Counter
initSystemCounter(SYSTEM_COUNTER_CNTCR_HDBG,
                  SYSTEM_COUNTER_CNTCR_FREQ0,
                  SYSTEM_COUNTER_CNTCR_nSCALE);
```

The first call sets the location of the System Counter, so that the driver functions can access its registers. This address is based on the Arm Base Platform Model. More information on this model's memory map can be found in the Fast Models Reference Manual.

The second call writes the `CNTCR` register. The code selects frequency update scheme 0, disables scaling and sets the enable bit. After this point, the system count will start incrementing.

Next `main()` has the following:

```
//
// Configure timer
//

// Configure the Secure Physical Timer
// This uses the CVAL/comparator to set an absolute time for the timer to fire
current_time = getPhysicalCount();
setSEL1PhysicalCompValue(current_time + 10000);
setSEL1PhysicalTimerCtrl(CNTPS_CTL_ENABLE);

// Configure the Non-secure Physical Timer
// This uses the TVAL/timer to fire the timer in X ticks
setNSEL1PhysicalTimerValue(20000);
setNSEL1PhysicalTimerCtrl(CNTP_CTL_ENABLE);
```

The preceding code configures two of the timers:

- Sets up the Secure physical timer, `CNTPS`, using the `CVAL`.
- Sets up the Non-secure EL1 physical timer, `CNTP`, using `TVAL`.

The code in `main()` then waits for both interrupts to be generated before exiting.

The interrupt handler is also within `main.c`:

```
void fiqHandler(void)
{
  uin32_t ID;

  // Read the IAR to get the INTID of the interrupt taken
  ID = readIARGrp0();

  printf("FIQ: Received INTID %d\n", ID);

  switch (ID)
  {
    case 29:
      setSEL1PhysicalTimerCtrl(0); // Disable timer to clear interrupt
      printf("FIQ: Secure Physical Timer\n");
      break;
    case 30:
      setNSEL1PhysicalTimerCtrl(0); // Disable timer to clear interrupt
      printf("FIQ: Non-secure EL1 Physical Timer\n");
      break;
    case 1023:
      printf("FIQ: Interrupt was spurious\n");
      return;
    default:
      printf("FIQ: Panic, unexpected INTID\n");
  }

  // Write EOIR to deactivate interrupt
  writeEOIGrp0(ID);

  flag++;
  return;
}
```

The interrupt handler reads the Interrupt Acknowledge Register (IAR) of the GIC to get the ID of the interrupt that has been taken. Based on the returned value, the handler then disables the appropriate timer to clear the interrupt. An alternative approach would be to set `IMASK`, and mask the interrupt, or update the comparator.

Finally, the interrupt handler writes the End of Interrupt Register (EOIR) of the GIC. This updates the internal state machine of the GIC for the taken interrupt.

8. Check your knowledge

The following questions will help you test you knowledge:

CNTFRQ_EL0 is automatically set by hardware to report the frequency of the system count. True or False?

FALSE. It is the responsibility of boot software in EL3 to populate the register with the correct value.

Describe the two ways to configure a given timer.

- `TVAL` sets the timer to trigger in X ticks (where X is the written value).
- `CVAL` sets the comparator in the timer to an absolute value, the timer fires when the count reaches that number.

In an interrupt handler, how can software clear a timer interrupt?

It can set `IMASK` (masking interrupts), it can clear `ENABLE` (disabling the timer) or update `CVAL`/`TVAL`.

Do the Generic Timer interrupts have edge-triggered or level-sensitive semantics?

Level-sensitive

When the Generic Timer is used to generate an event stream, how is the rate of events controlled?

`EVNTI` controls the rate of events by selecting which bit in the count must change for the event to be generated. `EVNTDIR` controls whether it is a 0-to1 or a 1-to-0 transition of that bit which triggers the event.

9. Related information

Here are some resources related to material in this guide:

Arm Community (ask development questions, and find articles and blogs on specific topics from Arm experts)

Here are some resources related to topics in this guide:

- Interrupts: The Generic Timer generates interrupts, which are handled by the GIC. For more information on the operation of the GIC, and how to configure it, refer to our GIC guide which is in development.
- Virtualization: Support for the Armv8.1-A Virtualization Host Extensions, and the topic of virtualization generally, is discussed in our Virtualization guide.
- Event stream: This guide introduced the Generic Timer's ability to generate an event stream as part of the Wait for Event mechanism. More information on Wait for Event will be found in our Synchronization guide that is in development.
- Example using Arm Development Studio: Learn about the memory map of the Arm Base Platform Model.

10. Next steps

The Generic Timer provides a common timer framework for Arm systems. In this guide, we have learned about the different components of the Generic Timer and their programming interfaces. You can put this learning into practice when you write your own code for low-level system initialization.

We reference the Generic Interrupt Controller (GIC) in this guide. If you want to learn more, look out for our series of guides on the GIC which are in development.

To keep learning about the Armv8-A architecture, see more in our series of guides.