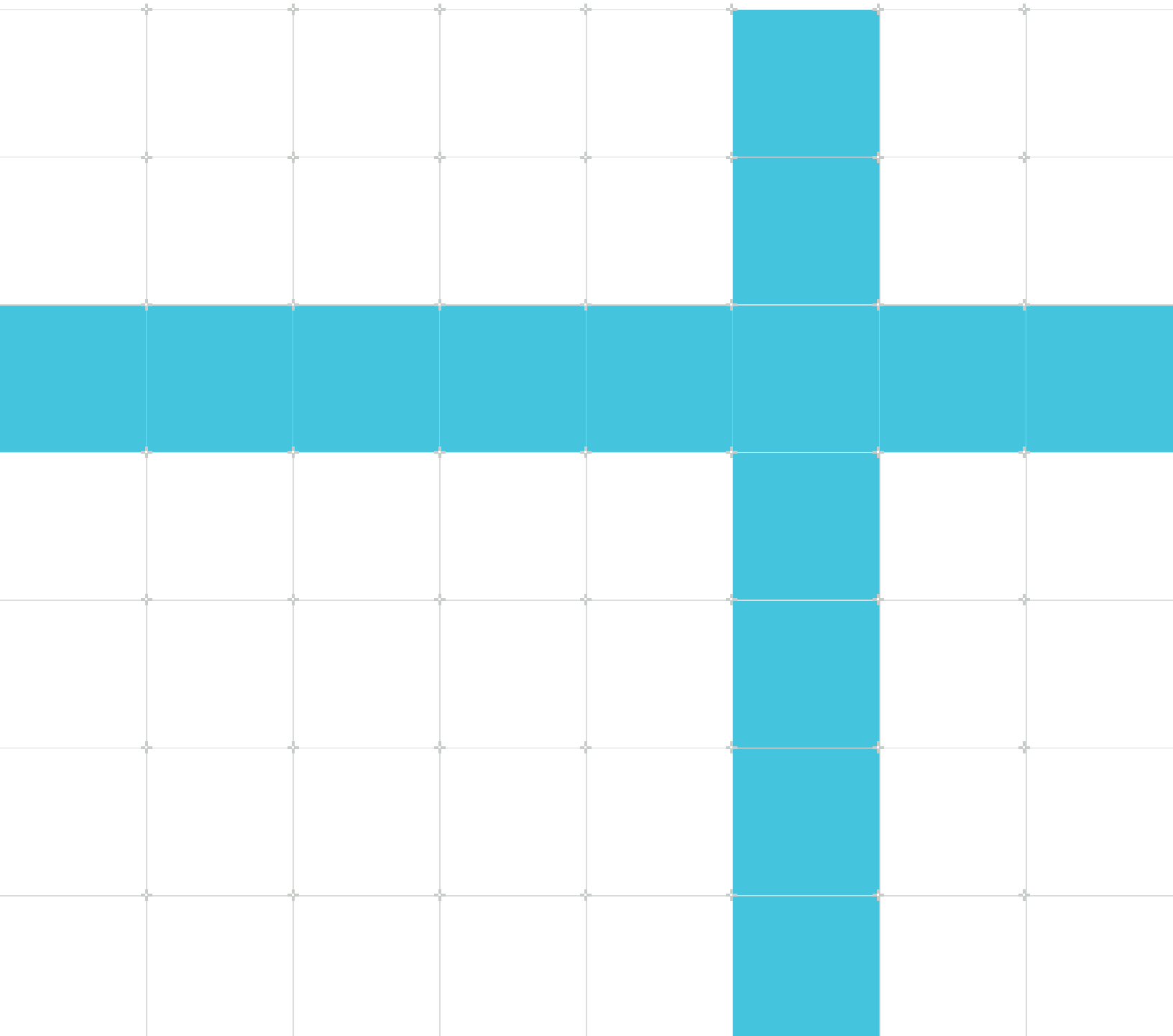
Armv8-R virtualization

Version 1.0

Issue
102909_0100_en



Armv8-R virtualization

Release information

Document history

Issue	Date	Confidentiality	Change
0100	29 March 2022	Non-Confidential	Initial release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

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

This guide introduces virtualization concepts and possibilities in the Armv8-R architecture. We explain these concepts using four examples, most of which are based on automotive industry applications. These examples help you understand the virtualization concepts and become familiar with the Arm development tools.

At the end of this guide, you will be able to:

- Describe essential virtualization concepts
- Build and run example code to demonstrate the virtualization features of the Armv8-R architecture

The source code and registry diagrams in this guide are specific to the Cortex-R52 processor, which is the first processor to implement the Armv8-R architecture. Other processors differ from the Cortex-R52 processor, however the ideas in this guide are still relevant.

Before you begin

This guide assumes that you have a basic understanding of virtualization and the role of the hypervisor. This guide also assumes that you are generally familiar with embedded programming and the C language. The Arm tools that we use in the examples use Arm assembly code and C.

Detailed instructions and system register descriptions are not included in these examples. For more information, refer to the following guides:

- Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile
- Arm Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile.

Before you get started with the examples, download the following tools:

- Arm Development Studio for compilation tools, debugging, and a simulation platform. Arm Development Studio contains the Arm Compiler 6 toolchain, Arm Debugger, and the Arm Development Studio Integrated Development Environment (IDE). It also contains the FVP_BaseR_Cortex-R52x1 Fixed Virtual Platform, which is required to run the Simple guest OS switcher, see The Generic Interrupt Controller and Guest OS switcher with virtual interrupts example. You can use the free 30-day trial version of Arm Development Studio to run these examples. To learn about how to install and use Arm Development Studio, read the Arm Development Studio documentation. You do not need any experience with Arm Development Studio to complete the examples.
- A multi-core Fixed Virtual Platform (FVP) from Fixed Virtual Platforms. FVP is required to run the SPIs and SGIs example.

You will then need to import and build the examples. For more information, see Download and build the examples.

2 Introduction to virtualization

In this section of the guide, we describe some important virtualization concepts. One of the largest target markets of the Cortex-R52 is the automotive industry. The Cortex-R profile cores are ideally suited to this market because they provide real-time, deterministic responses. If a response is deterministic, the time between an event occurring and the processor response is guaranteed to be within a specified timeframe. For example, a deterministic response is used in a Braking Control Module (BCM).

Let's consider a car containing several Electronic Control Units (ECUs) including the following:

- BCM
- Transmission Control Module (TCM)
- Suspension Control Module (SCM)

Typically, a car can contain 70 or more ECUs, and each ECU contains a processor running a Real-time Operating System (RTOS). In this case we can see the benefit of using fewer processors, each capable of controlling multiple ECUs, when designing the electronics system of a modern vehicle. Consolidating ECU processing reduces manufacturing costs, saves space, and reduces the weight of the vehicle.

Virtualization distributes processing power to multiple ECUs. In a consolidated system, each ECU still runs an RTOS, but the RTOS runs on a single hardware system that is controlled by a hypervisor. The RTOS becomes a guest Operating System (OS) and runs in a virtual machine that is managed by the hypervisor. Only one ECU and guest OS are active at a time, and the hypervisor switches between guest OSs as required.

A guest OS runs on a virtual machine. In this guide, the term guest OS refers to both the virtual machine and the guest OS that is running on it. If a hypervisor switches guest OSs, it also switches virtual machines.

Virtualization is possible on an Armv8-R processor because code can run at Exception level 2 above the guest OS Exception level 1. This privilege level enables a hypervisor to sit above one or more guest OSs.

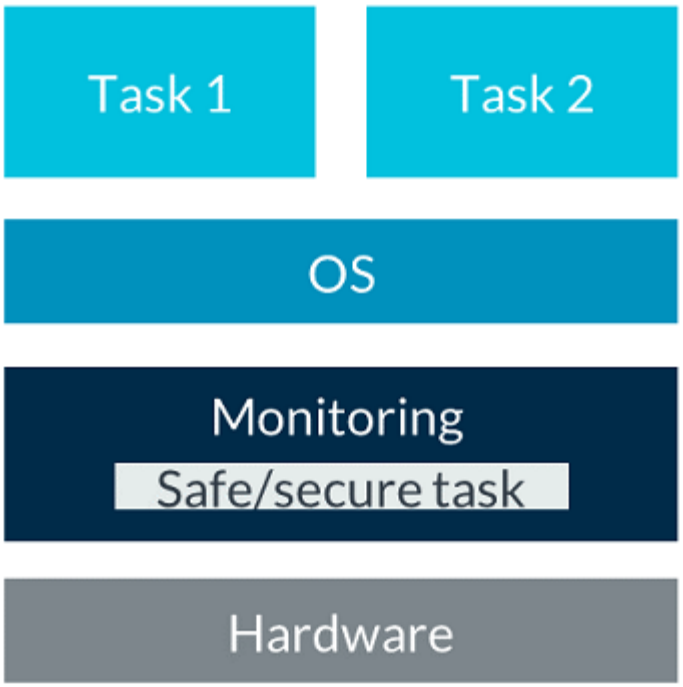
If you run a single RTOS, you can still use the highest privilege level and monitor the following safety and security tasks:

- Execute online and offline Built-In Self-Test (BIST) software
- Perform RAM scrubbing
- Deal with safety events
- Reboot an RTOS as required
- Perform secure boots
- Handle firmware updates

- Authenticate with a Hardware Security Module (HSM)

The following figure shows a simple scheduler running at Exception level 2 to allow safety tasks to run:

Figure 2-1: Scheduler example



Now that you understand what virtualization is and how it works, let's look at some essential virtualization concepts. These concepts explain how virtualization allows each guest OS to function as if it is an independent RTOS that is running on its own machine.

Virtual interrupts

A physical interrupt is an interrupt that physically occurs on the system, for example, a response to a button press or a timer. An interrupt results in either an asynchronous IRQ or FIQ exception. A virtual interrupt generates an exception. A guest OS handles this exception in the same way that a physical interrupt is handled on an unvirtualized RTOS. Hypervisors use virtual interrupts to distribute physical interrupts to guest OSs. Virtual FIQs and IRQs are both possible, and from the perspective of the guest OS, they behave like physical interrupts.

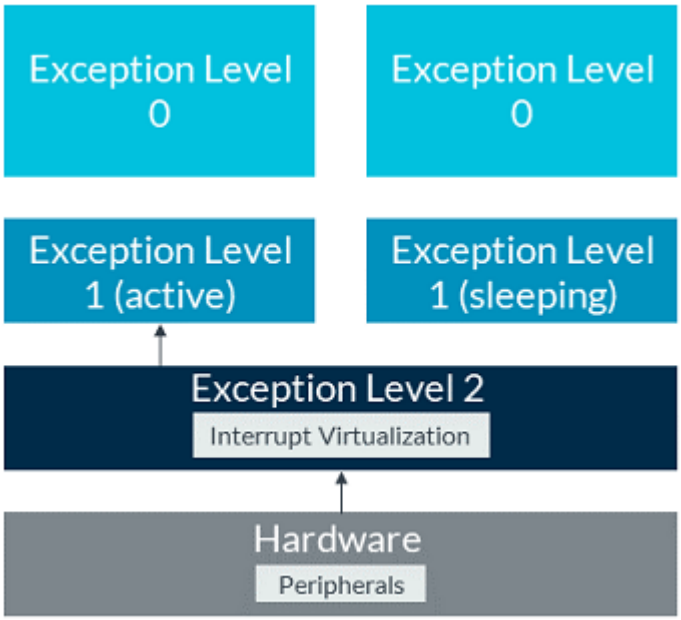
There are two ways to use virtual interrupts: using Hyp Configuration Register (HCR) flags, see Trapping exceptions and using the Generic Interrupt Controller (GIC) see Guest OS switcher with virtual interrupts example.

Trapping exceptions

Trapping exceptions are a response to interrupts and relate to virtual interrupts. Trapping exceptions allow the hypervisor to intercept exceptions destined for a guest OS.

Exceptions that occur in either an Exception level 0 or an Exception level 1 process are usually raised at Exception level 1. These exceptions are interpreted as trapped when they are raised at Exception level 2 instead. When an exception is trapped, a hypervisor running at Exception level 2 can decide whether to create a virtual interrupt for a guest OS to receive. Only an Exception level 2 process can create a virtual interrupt. In this case, trapping exceptions are an essential part of creating virtual interrupts, which then raise Exception level 1 exceptions. The process is shown in the following figure:

Figure 2-2: Trapping exceptions



There are three types of exceptions that can be trapped:

- FIQ
- IRQ
- Synchronous abort

When a trapped exception occurs in a process running at Exception level 1 or Exception level 0, the value of the Current Program Status Register (CPSR) F, I, or A flag is ignored. The program execution immediately jumps to the corresponding Exception level 2 exception handler.

When an exception is configured to be trapped at Exception level 2 when Exception level 1 and Exception level 0 processes are running, the related CPSR F, I, or A flag corresponds to the virtualized form. For example, the I flag masks virtual IRQs at Exception level 1 and Exception level 0 when physical IRQs are forwarded to Exception level 2.

When running an Exception level 2 process, the CPSR A, I, and F flags correspond to forwarded physical forms. Therefore, it is possible to mask forwarded physical interrupts when running at Exception level 2. When an interrupt is masked, it means it is not handled as an exception.

To trap either FIQ, IRQ, and asynchronous abort exceptions at Exception level 2, set one of the following flags on the Hypervisor Configuration Register (HCR):

- Trap FIQ Exceptions (bit 3)
- Trap IRQ Exceptions (bit 4)
- Trap Asynchronous Abort Exceptions (bit 5)


Note

Asynchronous aborts are not interrupts even though, like FIQs and IRQs, they generate asynchronous exceptions. Asynchronous aborts can be forwarded in a virtualized form but do not involve the GIC. This methodology is covered in OS monitor example.

The HCR is a secondary register under the primary CP15 register 1, which also controls other hypervisor-specific settings.

Trapping instructions and register access

This section describes the concept of trapping exceptions to Exception level 2. In this case, the processor traps the use of certain instructions, or attempts to access certain registers in an Exception level 0 or Exception level 1 process.

For example, an Armv8-R processor can be configured to trap Wait for Interrupt (WFI) and Wait For Event (WFE) instructions. These configurations enable the hypervisor to switch to another guest OS when the current guest OS enters a low power state. When a low power state is entered, the software waits for the next interrupt or event. Instead, an exception at Exception level 2 is raised and the hypervisor responds to this exception.

In another example, an Armv8-R processor can be configured to trap registers related to Memory Protection Unit (MPU) region creation. When a register is accessed, an exception is raised and the hypervisor responds to this exception.

Protected Memory System Architecture

The Protected Memory System Architecture (PMSA) in an Armv8-R processor allows spatial separation to occur in a system. Specifically, PMSA enables regions of memory to be defined and prevent access by other regions. For example, a guest OS is usually unable to access a region that is used for another guest OS. In addition, regions can be defined that require a higher privilege to access than others. For example, guest OS code cannot access a region that is defined for hypervisor code or data. The memory regions also have memory attributes, which are set according to what the region contains. For example, memory regions for peripherals such as device memory are not executable.

Virtual timers

Physical timers were a feature of earlier Arm-R series processors. Virtual timers are provided to work with the physical timers and are used to track virtual time. For example, there are two guest OSs: the first guest OS and second guest OS are both active for five seconds. After ten seconds of system time, the total virtual time for each guest OS is five seconds. The amount of time that each

guest OS is inactive is not calculated in the virtual time. Virtual time calculates the time using the guest OS and does not include the physical time it is inactive.

System timers use a 64-bit register to record the number of ticks that have occurred since the system started. Countdown timers are set to raise an interrupt after a specific number of ticks. These timers set another 64-bit register to a comparison time in the future. The interrupt is raised when the system time reaches the comparison time.

To enable complete virtualization, system timers and countdown timers can be physical or virtual.

Virtual machine IDs

Virtual machine IDs identify guest OSs and are assigned by the hypervisor. If IDs are exported to the bus, they can be used to filter access to peripherals. This protection, on a per device or per register basis, is obtained by programming an external System Memory Protection Unit (SMPU). The SMPU can filter access based on the virtual machine IDs. Virtual machine IDs can be a single entity in the system or distributed between the interconnect and the peripheral bridge.

Virtual machine IDs are also useful when debugging and can be used to limit Data Memory Barrier (DMB) and Data Synchronization Barrier (DSB) instructions.

Back up registers

An essential part of creating a virtual machine that is running a guest OS is being able to deactivate the virtual machine and then reactivate it later. When a virtual machine is deactivated, the virtual machine state must be saved so it can be restored later. When the virtual machine state is saved correctly, the guest OS can continue as if the virtual machine has not been deactivated. The content of the processor registers determines the state of a virtual machine. The hypervisor must back up all registers used by each guest OS when each guest OS becomes deactivated. When a guest OS becomes active again, the hypervisor can then restore the backed up registers for the guest OSs virtual machine. In the following figures, actions 1 and 2 continually cycle. Although only r0-5 is shown in the figure, more registers are backed up, including r0-r14:

Figure 2-3: Register backup Action1

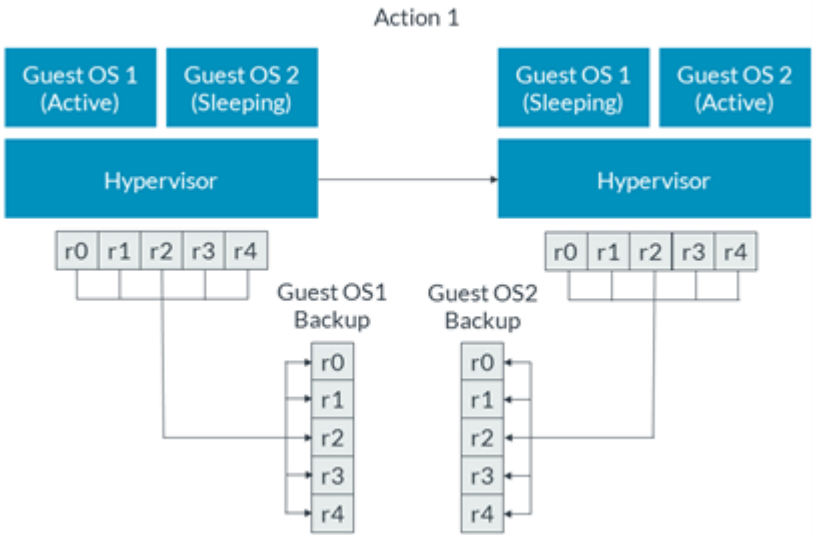
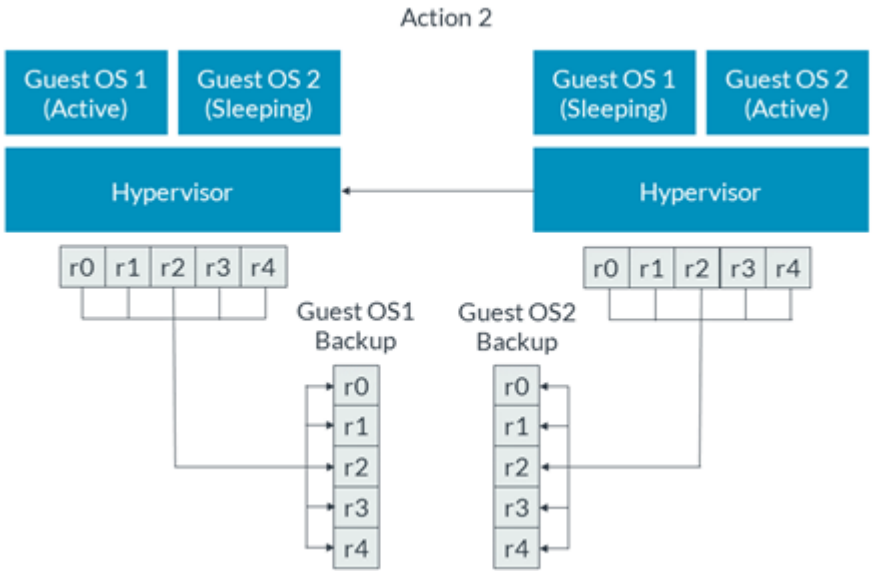


Figure 2-4: Register backup Action2



3 The Generic Interrupt Controller

The Generic Interrupt Controller (GIC) is responsible for routing interrupts to one or more cores and managing them throughout their lifecycle. This section of the guide introduces the GIC concepts that are used in the examples in this guide. For more detailed information about the GIC, see the Arm Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and version 4.0.

The GIC is architecturally independent from the Armv8-R architecture. Although other Armv8-R cores can support an alternative interrupt controller, the integrated GIC cannot be bypassed on Cortex-R52 cores.

The `Basic_GIC_Setup()` function contains generic code for configuring the GIC that is used in all examples in this guide. This function contains the global settings and the settings for an individual core, rather than the settings for specific interrupts.

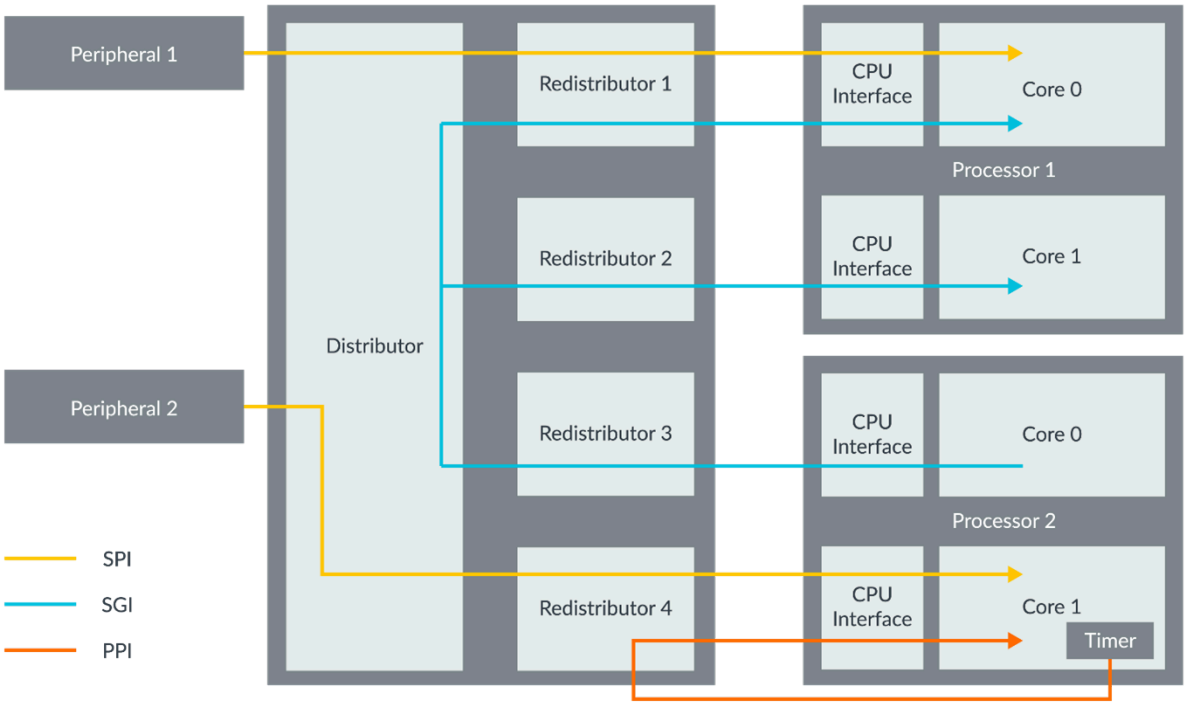
GIC interrupts

The GIC supports three types of interrupt:

- Shared Peripheral Interrupts (SPIs)
- Private Peripheral Interrupts (PPIs)
- Software Generated Interrupts (SGIs)

These interrupts differ in how they are generated and routed. Be careful not to confuse the types of interrupt from the perspective of the GIC with the differentiation between IRQs or FIQs. You can configure the GIC to signal any SPI, PPI, or SGI as either an IRQ or an FIQ. The following diagram shows an example of how the three types of interrupt are routed in the GIC:

Figure 3-1: GIC interrupts



All GIC interrupts have an ID, which enables them to be identified throughout their lifecycle. Where applicable, virtual interrupts often have the same ID as their physical counterparts. A physical counterpart does not need to exist for virtual interrupts to be created, and virtual SPIs, PPIs, and SGIs are all possible. Virtual interrupts can also be linked to a physical counterpart with a different ID.

Shared Peripheral Interrupts

SPIs are generated by peripherals that are external to the core, like a sensor. In a cluster system, SPIs must be routed to one target core using the GIC concept of affinity, which is explained in Guest OS switcher with virtual interrupts example. SPIs have an interrupt ID that ranges from 32-1019. Refer to the documentation on your system to find out which IDs are assigned to which peripherals.

Private Peripheral Interrupts

PPIs are generated by peripherals internal to a core and cannot be received by any other cores in a system. PPIs have interrupt IDs ranging from 16-31. You can find PPIs in the form of countdown timers in Simple guest OS switcher example.

Software Generated Interrupts

SGIs are not related to either an internal or external peripheral. Instead, SGIs are generated in the code by a write to a GIC register. Their primary use is inter-core communication. For example, if a core needs to know about a PPI on a second core, create an SGI that is targeted at the first core in

response to the PPI in the second core. SGIs have interrupt IDs ranging from 0-15. You can choose which IDs you assign to these custom messages.

GIC components

The GIC that is implemented on your system is made up of three different component types:

- A Distributor
- Redistributors
- CPU interfaces

The following table provides a description of each component type:

Component	Description	Register Groups	Mapping
Distributor	Provides global settings for affinity routing and enabling Group 0 interrupts (FIQs) and Group 1 interrupts (IRQs). Responsible for the SPI configuration including: • Whether a specific SPI is enabled or disabled • Whether a specific SPI is taken as an IRQ or an FIQ on the target core • The priority for a specific SPI • The target core for a specific SPI	GICD	Memory
Redistributor	Currently services one child CPU interface. Responsible for the PPI and SGI configuration including: • Whether a specific PPI or SGI is enabled on its child CPU interface • Whether a specific PPI or SGI is taken as an IRQ or an FIQ on its child CPU interface • The priority or a specific PPI or SGI Each core has its own set of Redistributor registers.	GICR	Memory
CPU interface	Controls the lifecycle of an interrupt as it is handled by a core. Creates SGIs and sets their target core or cores. Creates virtual interrupts and provides core-specific settings for: • Enabling Group 0 interrupts (FIQs) and Group 1 interrupts (IRQs) • A priority filter which decides which interrupts the core handles based on their priority • Interrupt preemption. Allows the core to resolve which interrupts can interrupt another, based on priority settings.	ICC, ICH, (ICV)	System

In many ways, the architecture of the GIC supports a restrictive functionality. GIC architecture provides a series of gates through which an interrupt must pass. For example, an interrupt can target a specific core. However, depending on how you set register bits in the Redistributor in the core, CPU interface, and the Distributor, the core might never handle the interrupt. In this case, an interrupt is not raised in a core as expected and the register setting prohibits it from reaching its destination.

GIC memory-mapped registers

The Distributor and Redistributor Registers are memory mapped. The Arm Generic Interrupt Controller Architecture Specification supplies the offsets for these registers. However, the address to which these offsets are relative is specific to each system.

CPU interface registers

IVC registers are the Exception level 1 equivalents of ICC registers. In an Exception level 1 process, they are accessed in the same way you access ICC registers. There is no difference in the instruction encoding. The ICV registers can be thought of as banked Exception level 1 registers. However, there is no way to directly access ICV registers at Exception level 2. ICV registers can only be manipulated at Exception level 2 by using specific ICH registers. ICH registers are also used to create virtual interrupts.

The Arm Generic Interrupt Controller Architecture Specification document also refers to the following legacy register groups: GICC, GICH, and GICV. These registers relate to GIC version 2.0 memory-mapped access, which is not supported on the Cortex-R52. Instead, version 3 system register access is used: ICC, ICH, and ICV.

Grouping in the GIC

In the GIC, interrupts are divided into two Groups: Group 0 and Group 1. Group 0 interrupts are handled as FIQs, and Group 1 interrupts are handled as IRQs. The GIC provides the flexibility that a specific interrupt, as defined by its interrupt ID, can be an IRQ or an FIQ. Whether it is an SPI, PPI, or SGI, interrupts can be either an IRQ or an FIQ.

Routing in the GIC

In an Armv8-R processor, the GIC architecture identifies a core using a 24-bit unique address. This address is not an address in memory, but a hierarchical addressing system where 8 bits are used for each level of the hierarchy. An example address is as follows:

```
<Level 2 [0…255]>.<Level 1 [0…255]>.<Level 0 [0…15]>
```

The levels of the hierarchy are also known as affinity levels. By specifying the affinity for an interrupt at each level, you can route it. The exact meaning of the levels is defined by the processor or SoC. The following code example could be used to represent the affinity levels:

```
<group of processors>.<processor>.<core>
```

A system does not have to use all three levels. For example, a relatively simple system based on a single four-core processor could use level 0 and define the cores as shown in the following code:

```
0.0.0
0.0.1
0.0.2
0.0.3
```

This system is the setup used in the four-core Cortex-R52 Fixed Virtual Platform supplied by Arm.

In addition to supplying cores with unique identifiers, the hierarchy addressing system allows interrupts to use affinity routing.

Because registers for configuring SPIs can only target one core, you specify the hierarchical address of the target core. The SPI then has an affinity for a single core. With SGIs, you can use the hierarchical addressing system to specify a group of cores for which the SGI has an affinity, and then target multiple cores within that group. Despite having eight bits available in the addressing system, Group 0 only allows 16 possible identifications. In practice, because of the way that SGIs target cores, level 0 is unlikely to be used to represent more than 16 cores. In the Interrupt Controller Software Generated Interrupt Group 1 Register (ICC_SGI1R), up to 16 bitflags are used to set the affinity for a core to either on or off.

At level 0, affinity is a binary term. A core is either targeted by an SPI or SGI, or it is not targeted.

The Cortex-R52 processor has a Distributor integrated into each processor, which cannot be bypassed. For this reason, the processor cannot see outside level 0. Theoretically, a system could be based on multiple Armv8-R series processors and have an external Distributor. This system could manage SPIs arriving for multiple processors and route SGIs between the processors.

Priorities in the GIC

Priorities are assigned to interrupts in the GIC. Up to eight bits can be available, depending on how the Armv8-R architecture has been implemented on the core. The Cortex-R52 has five bits, therefore interrupts can have a priority between 0 and 31, where 0 is the highest priority and 31 is the lowest.

In priority, IRQs and FIQs are unified. For example, a core handles an IRQ with a priority of 1 before an FIQ with a priority of 4. Another IRQ with a priority of 5 is handled last.

Related to priority is a lower priority interrupt that is handled by the core is put on hold while a higher priority interrupt is handled. If preemption is required, the core uses the priority bits to work out whether one interrupt can take priority over another that is already being handled. However, not all the priority bits have to be used when making this decision. The binary point is the point at which the bits are split, so that bits of higher significance are used in preemption decisions. The more significant bits are the group priority field, and the less significant bits are the sub priority field. All bits, including the subpriority field, are used when deciding the order in which the interrupts are handled.

The possible settings for the Cortex-R52 are as follows:

```
ggggg.
gggg.s
ggg.ss
gg.sss
g.ssss
.sssss (no preemption)
```

For example, if the setting `gg.sss` is chosen, interrupts are divided into four groups based on the possible values in the 2-bit group priority field. An interrupt with a priority of 2 and `0b00` in the group priority field could preempt an interrupt with a priority of 8 and `0b01` in the group priority field. However, an interrupt with a priority of 18 and `0b10` in the group priority field could not preempt an interrupt with a priority 19, which also has `0b10` in the group priority field.

You can also set a different binary point for IRQs and FIQs. For example, if the binary points for FIQs cannot preempt each other, IRQs can still preempt FIQs depending on how the IRQ binary point is set. If the IRQ binary point is set to no preemption, FIQs can still preempt IRQs. To prevent unwanted interrupt preemption, set the IRQ binary point to a high enough priority to ensure it cannot be preempted by an interrupt of another type.

Priority and preemption are not recognized across levels of privilege. Exception level 2 processes have a higher privilege than Exception level 1 processes. An interrupt can be raised and handled at Exception level 2 when an interrupt exception is handled at Exception level 1. This is the situation even if the Exception level 1 interrupt has the highest priority and prohibits preemption.

Interrupt lifecycle in the GIC

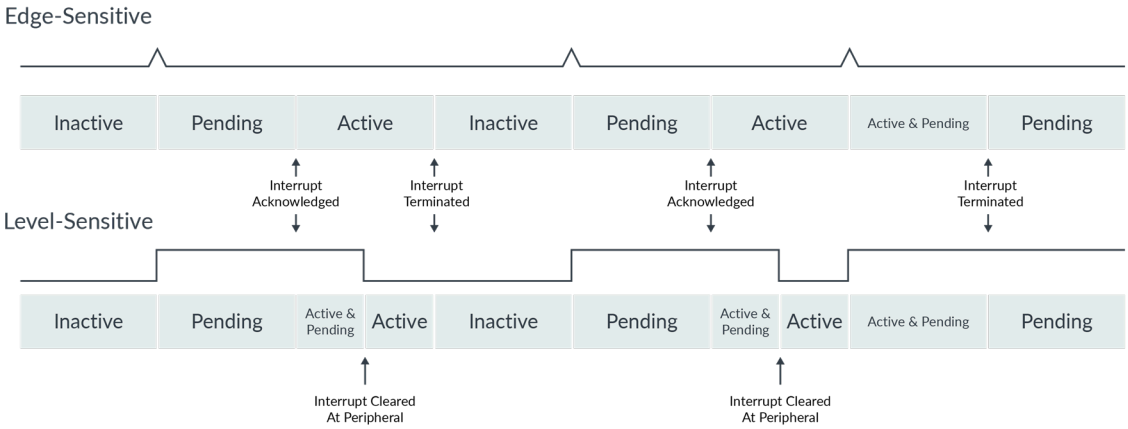
SPIs, PPIs, and SGIs have a lifecycle in the GIC. At any one time, the GIC stores one of four states for a specific interrupt ID:

- Inactive
- Pending
- Active and pending
- Active

Interrupts can be either level-triggered or edge-triggered, which affects the interrupt lifecycle. The following table describes the possible states for a specific interrupt ID, and the figure compares the lifecycle of a level-triggered and edge-triggered interrupt:

State	Description
Inactive	An interrupt with this interrupt ID is not currently asserted. No signal has been received from the source of the interrupt yet or any previous signals have been handled.
Pending	An interrupt which is asserted by the source. However, the interrupt has not been acknowledged. When an interrupt is in this state, it raises an exception at either Exception level 1 or Exception level 2 depending on how the core is configured.
Active and pending	The interrupt is acknowledged by the software, but one or more interrupts with the same interrupt ID are still pending. With level-triggered interrupts, the active interrupt and the pending interrupt can in fact be a single instance of the interrupt. This is because level-triggered interrupts remain pending until write back to the peripheral occurs. For example, to clear the interrupt, a peripheral register might be written to. Interrupts are acknowledged in the GIC by writing to either ICC_IAR0 or ICC IAR1. In the CPU interface, the only way an interrupt can move from active and pending to pending is by writing to ICC_EOIRO and ICC_EOIR1.
Active	The interrupt is acknowledged by the software. At this point, the software must react to the interrupt as appropriate. In a level-triggered interrupt, the active state signifies that the interrupt is cleared from the peripheral, and the level is dropped in response.

Figure 3-2: GIC interrupt example

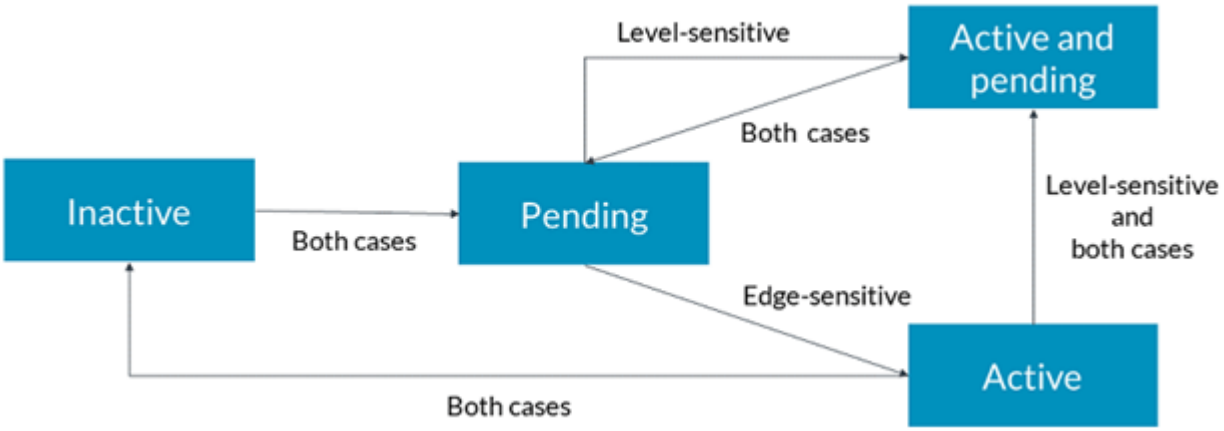


The first interrupt completes, and the state returns to inactive. In the level-triggered case, the level remains up until the interrupt is cleared at the peripheral. If a peripheral is edge-triggered, the level falls back almost immediately, and writing back to the peripheral is not required.

While the second interrupt is being handled, a third interrupt asserts. In both cases, the state changes to active and pending. This is the only way that an edge-triggered interrupt can have this state. When the second interrupt is terminated, the state returns to pending, because the third interrupt is still outstanding. The third interrupt can be acknowledged and handled in the same way the first and second interrupts were. Interrupts with the same ID are not asserted again when one has been acknowledged, until either ICC_EOIR0 or ICC_EOIR1 is called.

The following diagram summarizes the possible state changes that can occur:

Figure 3-3: Interrupt state changes



A separate virtual state is available for all interrupt IDs. Therefore, virtual interrupts that are generated in response to physical level-triggered interrupts might transition from pending to active when they are acknowledged. This transition occurs if the peripheral has already cleared the interrupt during the handling of the physical interrupt by the time that the virtual interrupt is acknowledged. This is the case in the Guest OS Switcher with Virtual Interrupts example, see Guest OS switcher with virtual interrupts example.

All PPI and SPI countdown timers used in these examples are predefined as level-triggered. When the interrupt that is associated with a countdown timer is triggered, the level remains up until the countdown value for the timer is reset. SGIs are always edge-triggered. The Interrupt Configuration Register 1 (GICR_ICFGR1) defines, when possible, whether PPIs are level-triggered or edge-triggered. The Interrupt Configuration Registers 2-61 (GICD_ICFGR2-61) define whether SPIs are level-triggered or edge-triggered. You can find more information about these registers in the Arm Generic Interrupt Controller Architecture Specification and the Arm Cortex-R52 Processor Technical Reference Manual.

4 Download and build the examples

To build and run the examples for this guide, follow these steps:

1. Download the Armv8-R Virtualization Examples. The examples are distributed as zipped Arm Development Studio projects.
2. Extract the examples into a folder on your computer.
3. Open Arm Development Studio.
4. Import all ten projects.
5. Build the Armv8-R Virtualization Common project. This directory creates a static library that all the other projects use. In this directory, `initialization.s` contains common initialization code including Generic Interrupt Controller (GIC) initialization code. The project also contains `clock_tick.c` and `clock_tick.h` to output a ticket clock to the console.
6. Build the Armv8-R Virtualization Simple guest OS switcher example. This project demonstrates virtualization using two simple guest OSs, register backups, using a physical countdown timer, trapping exceptions, and GIC Fast Interrupt Request (FIQ) and Private Peripheral Interrupts (PPI) configurations.
7. Build the Armv8-R Virtualization OS monitor example project. This project demonstrates how to monitor a simple RTOS, using a physical countdown timer to drive a simple clock task, trapping exceptions, how to generate a virtual interrupt using Hyp Configuration Register (HCR) flags, and GIC configuration.
8. Build the Armv8-R Virtualization Guest OS switcher with virtual interrupts example project. This project demonstrates virtualization with two simple guest OSs that both run a clock task, register backups, using a virtual countdown timer to drive the clock tasks in the guest OSs, trapping exceptions, how to generate a virtual system timer, virtual machine IDs, and GIC configuration.
9. Build the Armv8-R Virtualization SPIs and SGIs example project. This project demonstrates how to use multiple cores, using a countdown timer from a module that is external to the core, and GIC configuration with Shared Peripheral Interrupts (SPIs) and Software-generated Interrupts (SGIs).
10. Expand a project folder, for example Armv8-R Virtualization Example 1 - Simple Guest OS Switcher, to debug each example.
11. Double-click the .launch file. We wrote these examples using Arm Development Studio 2019.0 and the debug files must be updated to use the latest version of Arm Development Studio. Click OK to upgrade the debug configuration.
12. Click Debug to run the Development Studio IDE and connect to a Cortex-R52 FVP. The Fast Models window is displayed, and the Target Console shows the connection to the Cortex-R52 FVP.
13. Click Continue in the Debug Control panel or press F8 to restart the example execution. The result is displayed in the Target Console panel.

Study the examples

Some files have the same name in more than one project. These files have the same functionality but use different code at certain points. You can run a diff on files with the same name in each example to become familiar with how the examples work.

We recommend that you read about each example before moving on to the next example, and that you study the source code that is provided with the examples.

5 Simple guest OS switcher example

In this first of four examples, you learn about a simple guest OS switcher. In this example, we demonstrate how code running at Exception level 2 can switch between two different guest OSs. The two guest OSs in this example output Hello World messages and the code running at Exception level 2 functions as the hypervisor. The example code never drops to the Exception level 0 privilege level however, you can think of the `printf()` calls as tasks.

In this example, you learn about:

- Virtualization using two simple guest OSs
- Register backups
- Using a physical countdown timer
- Trapping exceptions
- GIC configurations: FIQs and PPIs

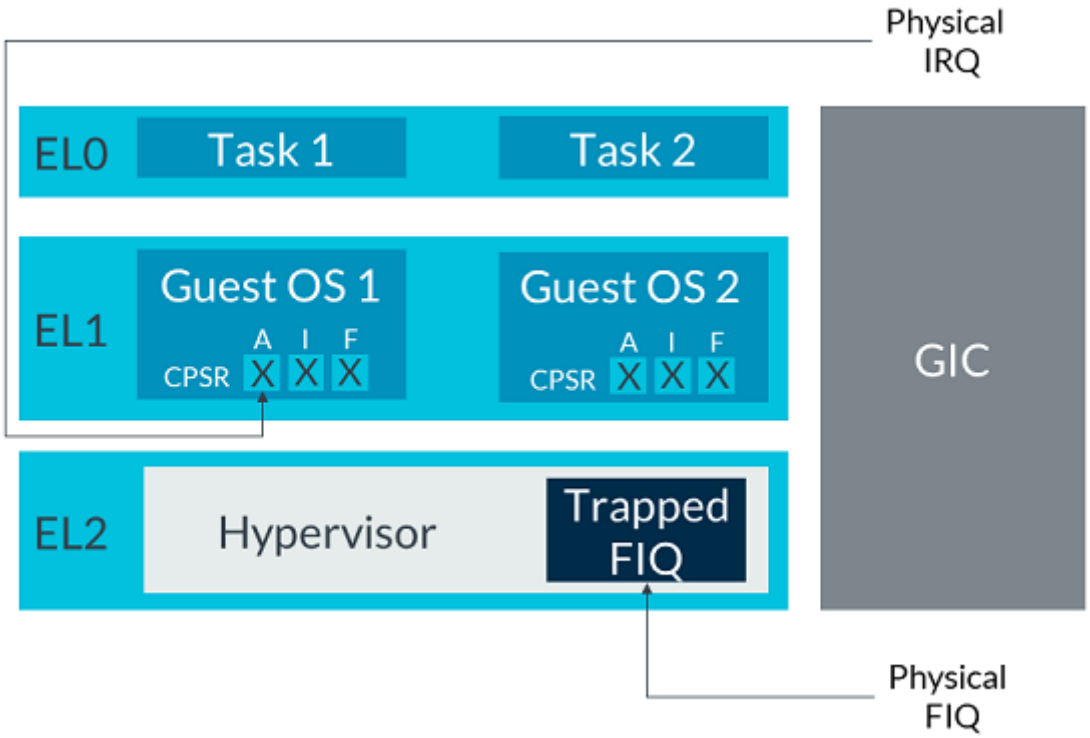
This example uses the following files:

- `hyp_start.s` contains hypervisor initialization code, interrupt handling code, and guest OS switching code
- `The initialization.s` file in the Armv8-R Virtualization Common directory contains essential code, including code for setting the Hyp Vector Base Address Register (HVBAR) to the Exception level 2 vector table and enabling floating-point functionality. A significant part of the code in `hyp_start.s` and `initialization.s` is dedicated to configuring the Generic Interrupt Controller (GIC). The GIC is responsible for forwarding IRQs and FIQs to the core and is explained more in Generic Interrupt Controller, see Guest OS switcher with virtual interrupts example. This code is required for the core to receive FIQs and IRQs.
- `guest_os1.s` and `guest_os2.s` embed the guest OS binaries. The guest OSs in this example are lightweight and are included as linked in binaries. Branching to the `__main()` global in these files initializes the standard library to allow `printf()` to be used to output a Hello World message. The code in these files sets Vector Base Address Register (VBAR) to the vector table for the guest OS, and initializes the stack for the different Exception level 1 modes.
- `main.c` contains C code, including code for message printing for the guest OSs
- `scatter.scat` describes the target memory map to the linker, which enables the linker to place the data and code for the example at the correct addresses in memory
- Armv8-R Virtualization Example 1 - Cortex-R52x1 FVP.launch is the Arm Development Studio IDE debug configuration for this example
- Armv8-R Virtualization Ex1 Guest OS 1 creates the first guest OS binary file this example uses
- Armv8-R Virtualization Ex1 Guest OS 2 creates the second guest OS binary file this example uses
- `guest_os1_start.s` and `guest_os2_start.s` contain initialization code for OS 1 and OS 2. All code runs at the Exception level 1 privilege level.

When the hypervisor switches from one guest OS to another, the restored guest OS registers are in the same configuration as when it last executed an instruction. After the guest OS is restored, it continues processing the previous instruction as if the switch never happened. In a virtualization scenario, each guest OS must operate without knowing that the other guest OS exists. If any guest OS registers are not successfully backed up and restored, the registers can become corrupted by the other guest OS.

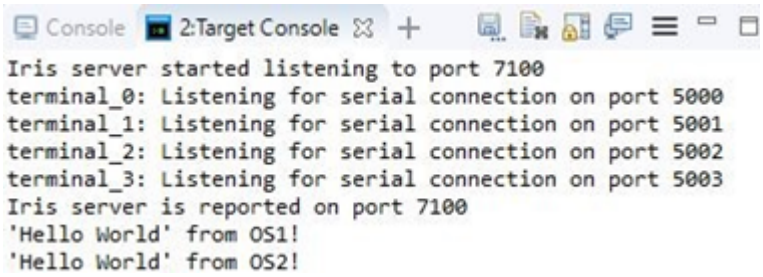
Armv8-R series processors can be configured so that physical interrupts are handled at Exception level 2 rather than Exception level 1. In this case, virtual interrupts are handled at Exception level 1. This allows the hypervisor to receive physical interrupts and decide what action to take. For example, the hypervisor can create a corresponding virtual interrupt. The hypervisor then returns to a guest OS to handle the virtual interrupt exception. In this example, no virtual interrupts are created and the physical interrupt, which occurs and is trapped every two seconds, switches to the other guest OS. The following diagram looks at the scenario demonstrated in this example:

Figure 5-1: Simple guest OS



The following screenshot shows the target console output for this example in the Arm Development Studio IDE:

Figure 5-2: Target console output



Initialize the guest OSs

In this example, an active guest OS is used to describe the OS that is running. The term sleeping guest OS refers to the guest OS that is not running and the register state is backed up.

Create a data structure for each guest OS

Storage space must be allocated to each guest OS, to back up registers when the guest OS is not active. The amount of storage space that is needed depends on which registers are backed up. In this example, the VBAR is backed up in addition to general-purpose registers, all banked registers, and FPU and SIMD registers.

The following code from hyp_start.s shows the memory that is allocated for guest OS 1 and an identical structure is used for guest OS 2:

```
// ------------------------------------------------------------
// Guest OS 1 data structure
// Stores registers for guest OS 1 when it is not running.
// ------------------------------------------------------------
guest_os1:
    .word 0     // R0
    .word 1     // R1
    .word 2     // R2
    .word 3     // R3
    .word 4     // R4
    .word 5     // R5
    .word 6     // R6
    .word 7     // LR_USR
    .word 8     // R7
    .word 9     // R8
    .word 10    // R9
    .word 11    // R10
    .word 12    // R11
    .word 13    // R12
    .word 14    // SP_USR
    .word 15    // SP_SVC
    .word 16    // LR_SVC
    .word 17    // SPSR_SVC
    .word 18    // SP_IRQ
    .word 19    // LR_IRQ
    .word 20    // SPSR_IRQ
    .word 21    // R8_FIQ
    .word 22    // R9_FIQ
    .word 23    // R10_FIQ
    .word 24    // R11_FIQ
    .word 25    // R12_FIQ
    .word 26    // SP_FIQ
    .word 27    // LR_FIQ
    .word 28    // SPSR_FIQ
    .word 29    // SP_UND
```

```
        .word 30    // LR_UND
        .word 31    // SPSR_UND
        .word 32    // SP_ABT
        .word 33    // LR_ABT
        .word 34    // SPSR_ABT
        .word 35    // ELR_HYP
        .word 36    // SPSR_HYP
        .word 37    // VBAR
        .quad 0     // FPU/SIMD - allocate two 8-byte quads each for registers q0-15
        .quad 1
        …
```

Because you cannot be sure which mode a guest OS is using when it switches to the other guest OS, all Exception level 1 banked registers must be backed up.

Notice that two Exception level 2 ELR_HYP and SPSR_HYP registers are also backed up because they contain information about a guest OS that must be restored when the guest OS becomes active again. Specifically, these two registers contain a value for the Program Counter (PC) and the CPSR at the time the guest OS stopped being active.

The following code shows how a small data structure is used to hold pointers to both data structures and when switching between guest OSs:

```
    guest_os_pointers:
        .word guest_os1 // Points to the active guest OS. Initialized to guest OS 1.
        .word guest_os2 // Points to the sleeping guest OS. Initialized to guest OS
2.
```

Populate the guest OS data structure

The guest OS data structures must be populated before they can be used.

The `init_guest_os()` function is multi-purpose and is called for both guest OSs. This function sets ELR_HYP and SPSR_HYP and saves the value for these registers to the fields that are allocated for them in the data structure.

The `init_guest_os()` function is first called for guest OS2 and then for guest OS1. After the second time that the function is called, ELR_HYP is set to branch to the beginning of the guest OS1 initialization. The branching occurs when the Exception Return (ERET) instruction is executed at the end of `EL2_Reset_Handler()`. The `init_guest_os()` function also sets SPSR_HYP to supervisor mode, so the ERET instruction also puts the processor into supervisor mode.

The ERET instruction is used several times in the example that we describe in this guide. The function of the ERET instruction is to return from an Exception level 2 exception by copying ELR_HYP into LR and copying SPSR_HYP into CPSR.

During normal program flow, ELR_HYP and SPSR_HYP are automatically populated when the handling of an Exception level 2 exception begins. Startup is different, but you can think of it like this: on an Armv8-R series processor, the software always begins in Hyp mode and the initial code is an Exception level 2 process. An exception has already been raised at the beginning of the program, but you must specify where and how to return from it before ERET is executed.

In this simple example, the ELR_HYP fields in the data structure are initialized to values that become available by the linker during compilation. These values are base addresses for the guest OS binaries, and they are defined in the scatter.scat files for the guest OSs.

Only the data structure for guest OS2 must be populated. This is because when guest OS 2 first becomes the active guest OS, the register values are taken from its data structure. Guest OS1 becomes active with the values that it requires already set in the registers by `init_guest_os()`. The initialization values for the guest OS1 data structure are never restored, even though `init_guest_os()` sets them.

The following code from `hyp_start.s` shows both `init_guest_os()` and the code that calls it for guest OS 2 and guest OS 1:

```
    // Entry points for the guest OS Vectors
    .equ GUEST_OS1_EL1_Vectors, Image$$GUEST_OS1$$Base    // 0x200000
    .equ GUEST_OS2_EL1_Vectors, Image$$GUEST_OS2$$Base    // 0x300000
    …
        // ** Initialize the guest OSs

        // Initialize guest OS 2
        LDR      r0, =GUEST_OS2_EL1_Vectors  // Guest OS 2 code starting point.
        LDR      r1, =guest_os2              // Data structure to back up registers
in.
        BL       init_guest_os
        DSB
        ISB

        // Initialize guest OS1
        LDR      r0, =GUEST_OS1_EL1_Vectors  // Guest OS 1 code starting point
        LDR      r1, =guest_os1              // Data structure to back up registers
in.
        BL       init_guest_os
        DSB
        ISB
    …
    // This function expects r0 and r1 to be set as follows:
    // r0: The point the code should start from when the guest OS first runs.
    // r1: The address of the data structure to back the registers up in.
    .global init_guest_os
    .type init_guest_os, "function"
    init_guest_os:

        // ELR_HYP
        STR      r0, [r1, #140]
        MSR      ELR_HYP, r0

        // SPSR_HYP
        MRS     r5, cpsr
        MOV     r6, #Mode_SVC
        BFI     r5, r6, #0, #5
        STR     r5, [r1, #144]
        MSR     SPSR_HYP, r5

        MOV     pc, lr
```

The value for VBAR is set in the guest OS initialization code in `guest_osx_start.s`, which runs the first time a guest OS becomes active. Therefore, VBAR is not set to an initial value in `init_guest_os()`.

Trapping exceptions

In this example, the guest OS is switched in response to the trapped interrupt and you configure the processor so that physical FIQs are trapped at Exception level 2.

The following code shows the Trap FIQ Exceptions flag being set on the HCR:

```
// Set bit 3 to route physical FIQ exceptions directly to EL2.
MRC     p15, 4, r0, c1, c1, 0      // Read HCR onto r0.
ORR     r0, r0, #0x8                // Set bit.
MCR     p15, 4, r0, c1, c1, 0      // Write r0 to HCR.
```

The diagram in Simple guest OS switcher, see The Generic Interrupt Controller shows what happens if a physical IRQ is raised. Because the Trap IRQ Exceptions flag is not set on the HCR, it is taken at Exception level 1 as a physical exception. However, the exception is masked and not processed unless the CPSR I flag is set to 0.

This example relies on a countdown timer that is used to generate a physical FIQ every two seconds. After initialization, the guest OSs are waiting for instructions in Exception level 1, which is seen in `main()` in main.c and `wait()` in guest_osx_start.s. When the timer is fired, the code jumps to EL2_FIQ_Handler function in `hyp_start.s`.

Countdown timers

A countdown timer on an Armv8-R series processor generate either an IRQ or an FIQ when it reaches 0. Each countdown timer consists of two registers: a register containing the current value of the countdown timer, and a register that controls whether interrupts are enabled for the timer. Countdown timers continue to decrement even if their interrupts are disabled.

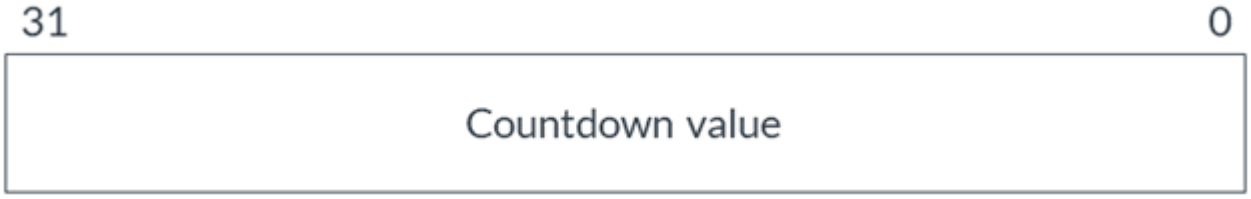
Countdown timers generate Private Peripheral Interrupts (PPIs). PPIs are described in Generic Interrupt Controller, see Guest OS switcher with virtual interrupts example. Like all interrupts, timer interrupts have a specific ID that identifies them as the reason behind an exception. In other examples in this guide, these IDs are also used to create virtual interrupts and to link the virtual interrupts to corresponding physical interrupts.

The following table lists the countdown timers and the registers that are used to control them:

Countdown timer	Notes	Interrupt ID	Related Registers
Exception level 2 physicalException level 2 physical	Not accessible from Exception level 1 or Exception level 0 processes	26	CNTHP_TVAL, CNTHP_CTL
Exception level 1 physical	Not accessible from Exception level 0 processes	30	CNTP_TVAL, CNTP_CTL
Virtual	Produces physical exceptions. Not accessible from Exception level 0 processes. Designed to be used by multiple guest OSs and be backed up and restored as appropriate. It is no different in functionality to the physical timers.	27	CNTV_TVAL, CNTV_CTL

The following diagram and table shows the layout for the Counter-timer Hyp Physical Timer TimerValue (CNTHP_TVAL), Counter-timer Physical Timer TimerValue (CNTP_TVAL), and Counter-timer Virtual Timer TimerValue (CNTV_TVAL):

Figure 5-3: Countdown Timer Value



Name	Bits	Function	Architecture reference
Countdown value	31:0	Set to the number of ticks that must occur before an interrupt is generated. This register updates and always reads the current value of the countdown timer. The length of a single tick varies depending on the system you are using. For example, the frequency of the countdown timer ticks on Fixed Virtual Platforms is 100MHZ. Therefore, you would set this register to 100000000 to receive an interrupt one second later, providing the timer is enabled.	TimerValue

To use the CNTHP_TVAL use MRC and MCR as shown in the following code:

```
MRC p15, 4, <Rt>, c14, c2, 0 ; Read CNTHP_TVAL into a general register
MCR p15, 4, <Rt>, c14, c2, 0 ; Write a general register into CNTHP_TVAL
```

To use the CNTP_TVAL use MRC and MCR as shown in the following code:

```
MRC p15, 0, <Rt>, c14, c2, 0 ; Read CNTP_TVAL into a general register
MCR p15, 0, <Rt>, c14, c2, 0 ; Write a general register into CNTP_TVAL
```

To use the CNTV_TVAL use MRC and MCR as shown in the following code:

```
MRC p15, 0, <Rt>, c14, c3, 0 ; Read CNTV_TVAL into a general register
MCR p15, 0, <Rt>, c14, c3, 0 ; Write a general register into CNTV_TVAL
```

The following diagram and table shows the layout for the Counter-timer Hyp Physical Timer Control (CNTHP_CTL), Counter-timer Physical Timer Control (CNTP_CTL), and Counter-timer Virtual Timer Control (CNTV_CTL):

Figure 5-4: Counter-timer

Name	Bits	Function	Architecture reference
Reserved	31:3	Do not set.	
Countdown Value Reached	2	A read-only flag that is set to `0b1` when the Countdown Value of the corresponding CNT*_TVAL register has reached 0.	ISTATUS
Mask Interrupt	1	Set to `0b1` to prevent interrupts from being generated when the Countdown Value of the corresponding CNT*_TVAL register has reached 0. Setting this flag does not stop the Countdown Value Reached flag being set when the Countdown Value of the corresponding CNT*_TVAL register reaches 0.	IMASK
Enabled	0	Set to `0b1` to enable the countdown timer. If this flag is set to `0b0`, the Countdown Value of the corresponding CNT*_TVAL register continues to decrement, but the Countdown Value Reached flag is never set.	ENABLE

The following code from `initialization.s` shows how the CNTFRQ register is set:

```
.equ SYSTEM_COUNTER_FREQUENCY, 0x5F5E100      // = 100MHz
…
    LDR r0, =SYSTEM_COUNTER_FREQUENCY
    MCR p15, 0, r0, c14, c0, 0  // Write r0 to CNTFRQ
```

For more information about countdown timers, see the section Generic Timer in the Arm Cortex-R52 Processor Technical Reference Manual.

Use timers in Fixed Virtual Platforms

When you use timers in Fixed Virtual Platforms, you set 2 bits at a specific address, as shown in the following code from `initialization.s`:

```
.equ FAST_MODELS_COUNTER_CONTROL,   0xAA430000
…
    LDR r0, =FAST_MODELS_COUNTER_CONTROL
    LDR r1, [r0]
    ORR r1, r1, 0x3
    STR r1, [r0]
```

Switch guest OSs

In this example, the guest OSs are switched using a call to `switch_guest_os()` in the Exception level 2 FIQ exception handler. First, we look at the Exception level 2 FIQ exception handler code:

```
.type EL2_FIQ_Handler, "function"
EL2_FIQ_Handler:
    // Push registers that are going to be used in this
    // handler onto the stack. These will be backed up in switch_guest_os().
    PUSH {r0-r6, lr}

    // Acknowledge the interrupt by reading the interrupt ID from the
    // Interrupt Controller Interrupt Acknowledge Register 0 (ICC_IAR0)
    MRC p15, 0, r6, c12, c8, 0

    // Do the bulk of the switch to the other guest OS.
    BL       switch_guest_os

    // Inform the processor that the processing of the timer interrupt
    // has been completed. This is achieved by writing the interrupt ID
    // to the Interrupt Controller End Of Interrupt Register 0 (ICC_EOIR0).
    MCR p15, 0, r6, c12, c8, 1     // Interrupt ID (26) is stored in r6.
```

```
        // Set timer up again just before switching to the other guest OS.
        LDR r0, =TICK_VALUE
        MCR p15, 4, r0, c14, c2, 0      // Write r0 to CNTHP_TVAL
        ISB

        // Pop registers that have been used in this
        // handler off the stack. The values on the stack
        // were updated in switch_guest_os() and are now from the
        // guest OS being restored.
        POP {r0-r6, lr}

        // ERET will update PC and CPSR from ELR_HYP and SPSR_HYP.
        // The program flow will branch into the guest OS being restored.
        // Values for ELR_HYP and SPSR_HYP were backed up when the
        // guest OS being restored went to sleep. These values have now been
        // reinstated in ELR_HYP and SPSR_HYP ready for use with ERET.
        ERET
```

The first thing you do is put the general-purpose registers 0-6 and the link register on the stack. This is because they are used in `EL2_FIQ_Handler()` and `switch_guest_os()`. A snapshot of these registers must be taken at the point that the code enters `EL2_FIQ_Handler()`. The snapshot is then backed up to the active guest OS data structure in `switch_guest_os()`.

Before the call to `switch_guest_os()`, you must read the interrupt ID from the Interrupt Controller Interrupt Acknowledge Register 0 (ICC_IAR0), which is a GIC register. This line of code works together with the Interrupt Controller End Of Interrupt Register 0 (ICC_EOIR0). The ICC_EOIR0 is a GIC register that is called directly after `switch_guest_os()`. Reading from ICC_IAR0 acknowledges the interrupt and informs the GIC that processing of the interrupt has begun. Writing to ICC_EOIR0 deactivates the interrupt after processing so it is not called again. This process is covered in more detail in Guest OS switcher with virtual interrupts example. If you debug the code, you can check R6 and find it is set to 26, which is the interrupt ID of the Exception level 2 physical timer.

There are two final things to do before calling ERET and branching into the guest OS that has been restored:

1. Start the countdown timer again by setting CNTHP_TVAL.
2. Write the restored values for R0-6 and LR from the stack and into the registers. After `switch_guest_os()` has been called, the values on the stack are from the guest OS about to be restored. A snapshot of the stack values was taken the previous time `EL2_FIQ_Handler()` was called.

ERET branches to the value which was backed up for the PC the previous time that `EL2_FIQ_Handler()` was called. ERET also restores the Exception level 1 processor mode that was backed up into SPSR_HYP during the same call. This repeats the process that occurred when guest OS 1 was run for the first time.

Now, examine what happens in `switch_guest_os()` in the following code example:

```
    // Note: this function expects r0-6 and lr have been pushed on the stack.
    .global switch_guest_os
    .type switch_guest_os, "function"
    switch_guest_os:

        // Load the pointers of guest OS data structures.
```

```
        LDR    r2, =guest_os_pointers
        LDR    r0, [r2]          // Load address of active guest OS into r0.
        LDR    r1, [r2, #4]      // Load address of sleeping guest OS into r1.

        // Swap the pointers ready for next time.
        STR    r1, [r2]          // Sleeping guest OS becomes the active guest OS.
        STR    r0, [r2, #4]      // Active guest OS becomes the sleeping guest OS.
        …
```

This code fetches the pointers to the active guest OS and sleeping guest OS data structures into general-purpose registers. For the remainder of the function, these pointers are used to back up and restore data. However, before this happens, the pointers are swapped inside the guest_os_pointers data structure, ready for the next time that the function is called.

The first registers that `switch_guest_os()` handles are R0-6 and LR, as shown in the following code:

```
        MOV    r3, sp         // Load stack pointer to r3.
        MOV    r4, #8         // Set counter to 8. Note that
                              // r0-6 + lr (R14_USR) have been put on the stack.
    1:
        LDR    r5, [r3]       // Copy register for the active guest OS from the
stack.
        STR    r5, [r0], #4   // Back it up it so the active guest OS can sleep.
        LDR    r5, [r1], #4   // Load register value for the sleeping guest OS into
r5.
        STR    r5, [r3], #4   // Place it on the stack ready to be put in the
register.
        SUB    r4, r4, #1     // Decrement counter.
        CMP    r4, #0x0
        BNE    1b
        …
```

This code works with the pointers to the active and sleeping guest OSs, which are already loaded into R0 and R1. As the code loops, values that are popped off the stack are backed up for the active guest OS and values for the sleeping guest OS are put onto the stack. At the end of the `EL2_FIQ_Handler()`, these values are restored from the stack into the actual registers.

The rest of the function follows the same principle without looping or using the stack. For example, the following code shows how SPSR_HYP and VBAR are handled:

```
        // SPSR_HYP - backup/restore ready for use with ERET
        MRS    r2, SPSR_HYP
        STR    r2, [r0], #4
        LDR    r2, [r1], #4
        MSR    SPSR_HYP, r2

        // VBAR - Vector Base Address Register
        MRC    p15, 0, r2, c12, c0, 0
        STR    r2, [r0], #4
        LDR    r2, [r1], #4
        MCR    p15, 0, r2, c12, c0, 0
```

On Armv8-R series processors, you can use MRS and MSR instructions to move banked registers to and from general-purpose registers. However, with Armv8-A series processors, they cannot be used to move coprocessor registers.

6 OS monitor example

In this example, you learn how to set up timers and interrupts on a system, virtualize FIQs and IRQs, and how the GIC can virtualize exceptions.

This example includes just one OS that simulates a scenario in which the Exception level 2 process functions as a monitor rather than a hypervisor. The monitoring scenario is introduced in Introduction to virtualization. This example also demonstrates instruction execution and one method of forwarding virtual interrupts.

This example works with both FIQs and IRQs. FIQs are trapped at Exception level 2, but IRQs are not. IRQs behave as if there is no virtualization and are handled directly at Exception level 1 by the OS. In a real world scenario, a monitor responds to FIQ exceptions being raised and the OS receives IRQs as usual.

This example uses the following files:

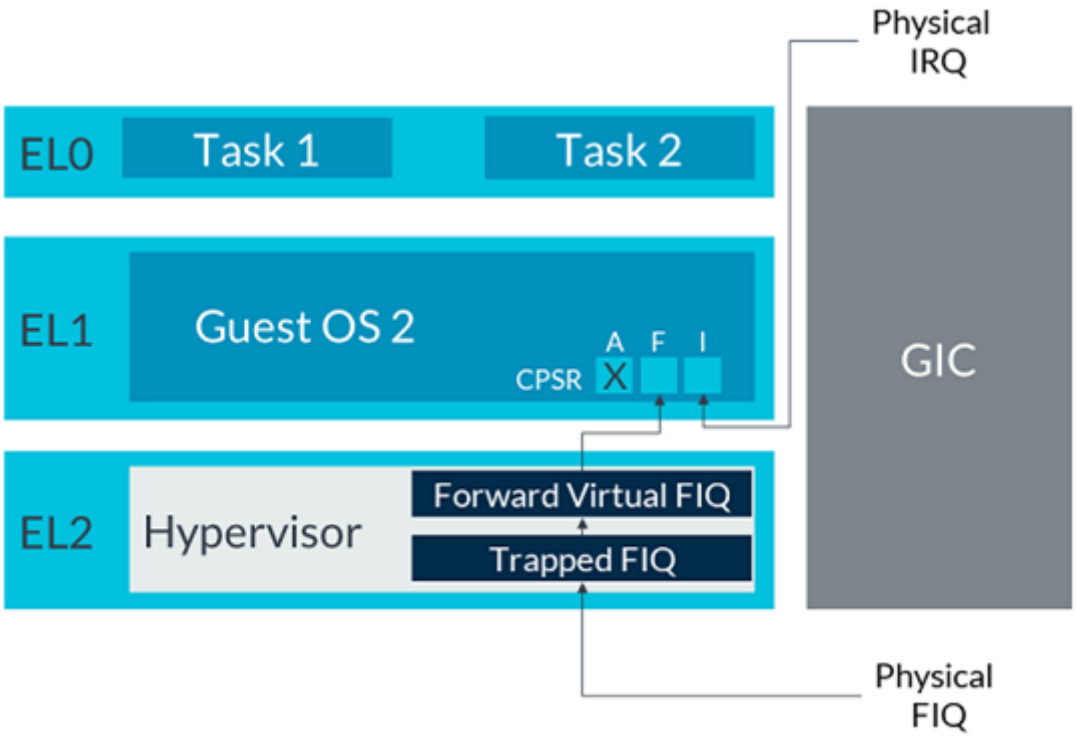
- `monitor_start.s`: This file contains monitoring initialization and interrupt handling code.
- `main.c`: This file contains C code, including code for message printing for the monitor.
- `scatter.scat`: This file describes the target memory map to the linker, which enables the linker to place the data and code for the example at the correct addresses in memory.
- `Armv8-R Virtualization Example 2 - Cortex-R52x1 FVP.launch`: This file is the Arm Development Studio IDE debug configuration for this example project.
- `clockTick.c`: This file is in the Armv8-R Virtualization Common directory.

This example can be broken down into the following process:

1. Essential configuration at Exception level 2 including GIC setup. Branch to `__main()` to initialize the standard library and then call `main()`. The branch to the `__main()` initializes the standard library. If you want to use `printf()` from the monitor, add another step to initialize the standard library. A separate standard library with its own address is used for the OS, and the code of the OS initializes it independently. If you want to print to the console from both the monitor and the guest OS binary, you need a separate instance of the standard library for each.
2. Change from Hyp mode to Supervisor mode, so that the program can demonstrate instruction trapping, specifically a WFI instruction, in an Exception level 1 process. The hypervisor is configured to trap WFI instructions, and therefore `EL2_HypModeEntry_Handler()` is called when the WFI instruction is executed in Supervisor mode. Although this example is superficial, trapping instruction execution and register access is a powerful technique. It enables Exception level 2 software to monitor various situations.
3. Jump to `EL2_HypModeEntry_Handler()` because of a WFI instruction trap.
4. Branch to the OS and complete the setup of the OS.

The following diagram shows the scenario that this example demonstrates:

Figure 6-1: OS monitor example



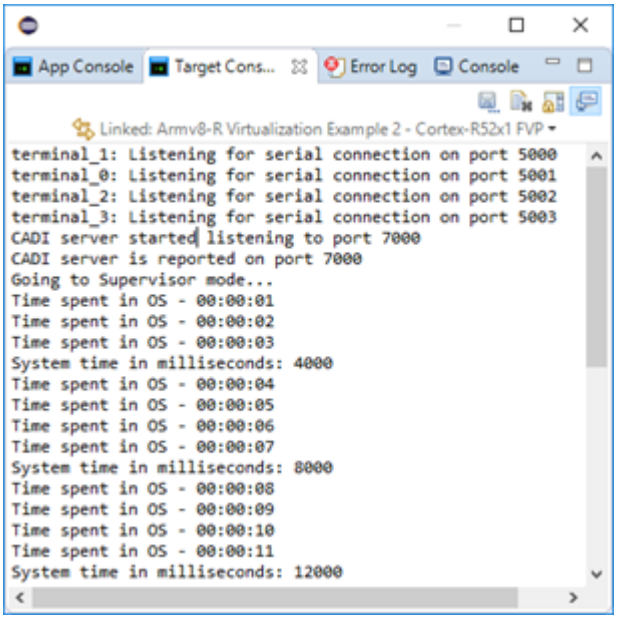
The following diagram shows the scenario that this example demonstrates: Although there is only one guest OS in this example, its functionality is more sophisticated than either of the guest OSs that are used in Simple guest OS switcher example. This guest OS includes code to handle both physical IRQs and virtual FIQs:

- Physical IRQs fire every millisecond. These IRQs are used for a clock that outputs the time since startup in hours, minutes, and seconds.
- Virtual FIQ exception handler outputs the time since startup in milliseconds, and the virtual FIQ itself is forwarded from a physical FIQ, which is raised every four seconds. The job that the virtual FIQ exception handler does is trivial and could be done in the monitor code when the physical FIQ is handled. However, the FIQs are virtualized just to demonstrate one method of forwarding exceptions.

Both the physical IRQs and physical FIQs are generated from countdown timers.

The following screenshot shows the Target Console output for this example in the Arm Development Studio Integrated Development Environment:

Figure 6-2: Target Console output



Trap WFI instructions

In this example, you learn how a WFI instruction execution can be trapped at Exception level 2.

To trap WFI instructions, set the Trap WFI Instructions flag (bit 13) on the HCR. The HCR contains several settings that specifically trap the execution of instructions and the access of registers. We recommend that you become familiar with the possibilities that these settings provide.

When an instruction execution or a register access is trapped, a Hyp Trap exception is raised in response. This exception is also raised when Hyp mode is reentered, because of an HVC instruction being executed in an Exception level 1 process. Therefore, it is also referred to as the Hyp Mode Entry exception.

The following code from `monitor_start.s` shows the HCR being configured to trap WFI Instructions:

```
        // Set bit 3 to route physical FIQ exceptions directly to EL2.
        // Set bit 13 to trap WFI instructions to EL2.
        MOV     r0, #0x2008                 // Set bits.
        MCR     p15, 4, r0, c1, c1, 0      // Write r0 to HCR.
```

The jump to the Hyp Trap exception occurs immediately after the WFI instruction in `wait()` is executed. This jump might be prevented if there is a pending interrupt or debug event. You can find wait() in monitor_start.s.

Considering that the Hyp Trap exception can be raised for several reasons, how do you know why it has been called? When the OS handles the virtual FIQ exception, the software also temporarily returns to Hyp mode. You can use the Hyp Syndrome Register (HSR) to find the reason for the Hyp Trap exception.

In this example, the Exception Class setting (bits 26-31) is used to find the reason behind the exception. The following code shows the relevant code in `EL2_HypModeEntry_Handler()`:

```
    .type EL2_HypModeEntry_Handler, "function"
    EL2_HypModeEntry_Handler:
        //Back up the two registers used here.
        PUSH {r0, r1}

        //Find out reason Hyp mode was entered.
        MRC p15, 4, r0, c5, c2, 0
        AND r0, r0, #0xFC000000              // Mask off the exception reason.
        CMP r0, #(WFI_EXCEPTION << 26)
        BEQ turn_off_wfi_trapping_and_goto_os // Check if this is a result of
                                              // a WFI instruction being executed.
    …
    turn_off_wfi_trapping_and_goto_os:
        // Turn off the "Trap WFI Instructions" flag in the Hyp Configuration
Register.
        MRC p15, 4, r1, c1, c1, 0           // Read HCR.
        AND r1, r1, #0xFFFFDFFF               // Set bit 13 to 0.
        MCR p15, 4, r1, c1, c1, 0           // Write back to HCR.

        // Drop into the OS.
        MRS r0, cpsr
        MOV r1, #Mode_SVC
        BFI r0, r1, #0, #6
        MSR spsr_hyp, r0
        LDR r0, =OS_EL1_Vectors
        MSR elr_hyp, r0

        //Restore the two registers used here from the stack.
        POP {r0, r1}

        ERET
```

If it is determined that the Hyp Trap exception occurred because of a trapped WFI instruction, WFI trapping is turned off and the code branches to `EL1_Reset_Handler()` in the OS. In this example, it is not necessary to trap WFI instructions when it has been used to demonstrate the return to Exception level 2 mode.

In this example, the other reason the Hyp Trap exception can occur is that an HVC instruction is executed in the virtual FIQ handler. The HVC instruction is executed here so that the code can change to Exception level 2. The code then turns off the HCR Forward Virtual FIQ Exceptions flag to prevent repeatedly raising a virtual FIQ exception at Exception level 1.

For more information about the HCR settings, refer to the section Hyp Configuration Register in the Arm Cortex-R52 Processor Technical Reference Manual.

Allow the OS to receive IRQs and FIQs

One of the most important lines of code in os_start.s is:

```
    CPSIE if
```

This instruction disables the CPSR IRQ and FIQ masks flags, so that physical IRQs and virtual FIQs can be received in the OS. The masking flags do not differentiate. Depending on how the trapping

flags are set on the HCR, physical or virtual interrupt exceptions can be masked. In this example no interrupts, physical or virtual, are raised at Exception level 1, so you do not need to mask flags.

Use IRQs to count the time

The HCR Trap IRQ Exceptions flag is not set, so IRQs are handled at Exception level 1. They are used to call a `clockTick()` function, which outputs the time that has elapsed since the program started. The Exception level 1 physical countdown timer has an interrupt ID of 30 and is used to generate the IRQ exception. The following code shows the `EL1_IRQ_Handler()`:

```
    .type EL1_IRQ_Handler, "function"
    EL1_IRQ_Handler:
        // Push the registers used in the handler onto the IRQ stack.
        // This includes r0-r3, r12, and lr, which are mandated by the
        // Arm Architecture Procedure Call Standard (AAPCS).
        // This is required because external C functions are called in the handler.
        // Also pushed are r4 and r5, which are used as workhorse registers.
        PUSH {r0-r5, r12, lr}

        // Acknowledge the interrupt by reading the physical interrupt ID from the
        // Interrupt Controller Interrupt Acknowledge Register 1 (ICC_IAR1)
        MRC p15, 0, r4, c12, c12, 0 // Store in r4.

        // AAPCS mandates 8-byte alignment at all external boundaries
        // (separately compiled or assembled files).
        // Check if the sp is 8-byte aligned.
        AND r5, sp, #4              // r5 contains either 0 (if sp is already
aligned)
                                    // or 4 (if sp is misaligned) after this AND.
        SUB sp, sp, r5              // Subtract either 0 or 4 to align to 8 bytes.

        // Branch to clockTick() C function. All C subroutines called
        // conform to the AAPCS standard.
        LDR r0, =clockTick
        BLX r0

        // Update countdown timer.
        LDR r0,  =OS_TICK_VALUE
        MCR p15, 0, r0, c14, c2, 0  // Write r0 to CNTP_TVAL.

        ADD sp, sp, r5              // Restore sp to its previous alignment.

        // Inform the core that the processing of the EL1 physical timer interrupt
        // has been completed. This is achieved by writing the interrupt ID
        // to the Interrupt Controller End Of Interrupt Register 1 (ICC_EOIR1).
        MCR p15, 0, r4, c12, c12, 1 // Interrupt ID 30 stored in r4.

        POP {r0-r5, r12, lr}        // Pop the registers to restore the context.
        SUBS pc, lr, #4             // Perform legacy lr adjustment, necessary for
some
                                    // EL1 modes, and return to previous mode.
```

The `clockTick()` function calls _clockTick(). The following code from `clock_tick.c` shows _clockTick():

```
    void _clockTick(char* message)
    {
        static unsigned char seconds = 0;
        static unsigned char minutes = 0;
        static unsigned char hours = 0;

        static unsigned int ms = 0;

        ms++;
```

```
        // check if a second has elapsed.
        if (ms==1000)
        {
            ms = 0;
            // update timer
            seconds++;
            if (seconds == 60)
            {
                seconds = 0;
                minutes++;
                if (minutes == 60)
                {
                    minutes = 0;
                    hours++;
                    if (hours == 24)
                    {
                        hours = 0;
                    }
                }
            }
            printf("%s", message);
            printf(" - ");
            if (hours >= 10)
            {
                printf("%d",hours);
            }
            else
            {
                printf("0%d",hours);
            }
            printf(":");
            if (minutes >= 10)
            {
                printf("%d",minutes);
            }
            else
            {
                printf("0%d",minutes);
            }
            printf(":");
            if (seconds >= 10)
            {
                printf("%d\n",seconds);
            }
            else
            {
                printf("0%d\n",seconds);
            }
        }
    }
```

The `_clockTick()` function expects to be called every millisecond, and tracks the amount of time it has been called using the static C variable ms.

Forward physical FIQs as virtual FIQs

When a physical FIQ is raised at Exception level 2, you can forward it to the OS as a virtual FIQ exception by setting a flag on the HCR. The HCR has three flags for forwarding virtual exceptions, and they correspond to the three flags that are used to trap exceptions at Exception level 2:

- Forward Virtual FIQ Exceptions (bit 6)
- Forward Virtual IRQ Exceptions (bit 7)
- Forward Virtual Asynchronous Abort Exceptions (bit 8)

In this guide, this methodology for forwarding asynchronous exceptions by setting an HCR flag is referred to as the quick and easy method. For more information, see The quick and easy method.

In this example, the Forward Virtual FIQ Exceptions flag is used, which is set in EL2_FIQ_Handler(). The flag is turned off when each virtual FIQ that it generates is handled at Exception level 1. For this reason, the flag must be set again each time that EL2_FIQ_Handler() is called. Turn off the flag to prevent the virtual FIQ from being raised repeatedly. The following code from EL2_FIQ_Handler() shows the Forward Virtual FIQ Exceptions flag turned on:

```
    // Set FIQ forwarding
    MRC p15, 4, r1, c1, c1, 0       // Read Hyp Configuration Register
    ORR r1, r1, #0x40
    MCR p15, 4, r1, c1, c1, 0       // Write Hyp Configuration Register
```

The virtual interrupt is raised when the ERET instruction is called at the end of EL2_FIQ_Handler() and the processor changes to supervisor mode. You can test this by stepping through the code with a debugger.

The HCR Forward Virtual FIQ Exceptions flag is turned off in `EL2_HypModeEntry_Handler()`. The jump to `EL2_HypModeEntry_Handler()` occurs in `EL1_FIQ_Handler()` when the HVC instruction is called. The following code shows the relevant code in `EL2_HypModeEntry_Handler()`:

```
.type EL2_HypModeEntry_Handler, "function"
EL2_HypModeEntry_Handler:
    //Back up the two registers used here.
    PUSH {r0, r1}

    //Find out reason Hyp mode was entered.
    MRC p15, 4, r0, c5, c2, 0
    AND r0, r0, #0xFC000000               // Mask off the exception reason.
    CMP r0, #(WFI_EXCEPTION << 26)
    BEQ turn_off_wfi_trapping_and_goto_os // Check if this is a result of
                                          // a WFI instruction being executed.

    // The exception was not raised because of a WFI instruction.
    // Therefore, it must be from an E1_FIQ_Handler.
    // Turn off the "Forward Virtual FIQ Exceptions" flag
    // in the Hyp Configuration Register.
    // This stops the exception being raised again and again.
    MRC p15, 4, r1, c1, c1, 0           // Read HCR.
    AND r1, r1, #0xFFFFFFBF               // Set bit 6 to 0.
    MCR p15, 4, r1, c1, c1, 0           // Write back to HCR.

    //Restore the two registers used here from the stack.
    POP {r0, r1}

    ERET

turn_off_wfi_trapping_and_goto_os:
…
```

System timers

An Armv8-R series processor includes two system timers: a physical system timer and a virtual system timer.

In this example, you use the physical system timer to output the time that the OS has been running in milliseconds. The physical system timer value is read from the Counter-timer Physical Count register (CNTPCT). This 64-bit register stores the ticks that have occurred since the system started. Because only one OS runs in the example, you can use CNTPCT to record the time that the OS has run for. However, in a scenario involving virtualization with more than one OS, you must use the Counter-timer Virtual Count register (CNTVCT). This scenario is covered in Guest OS switcher with virtual interrupts example.

The following figure and table shows the layout for the CNTPCT and CNTVCT:

Figure 6-3: CNTPCT and CNTVCT layout

Name	Bits	Function	Description	Architecture reference
System Count	63:0	CNTPCT	The number of ticks that the system has been running for	Physical or Virtual count value
System Count	63:0	CNTVCT	The number of ticks that the virtualized system (like a guest OS) has been running for	Physical or Virtual count value

The length of a single tick varies, depending on the system that you are using. For example, the frequency of the system timer ticks on Fixed Virtual Platforms is 100MHz. The register is read-only.

On an Armv8-R series processor, the MRRC instruction is used to read from a 64-bit register. To read from CNTPCT, use MRRC as follows:

```
MRRC p15, 0, <Rt>, <Rt2>, c14 ; Read CNTPCT into two 32-bit general registers
```

To read from CNTVCT, use MRRC as follows:

```
MRRC p15, 1, <Rt>, <Rt2>, c14 ; Read CNTVCT into two 32-bit general registers
```

In this example, the value for CNTPCT is obtained and printed to the console in the OS Exception level 1 process. The following process explains how the value is obtained:

1. The `EL1_FIQ_Handler()` function in os_start.s is called to handle the virtual FIQ exception.
2. The handler calls the `systemTimeInMilliseconds()` C function.
3. In `systemTimeInMilliseconds()`:
   - `getCNTPCT()` is called to retrieve the physical system timer value.
   - The system timer value is divided by a system-specific value to get the time in milliseconds.
   - `printf()` is called to output the system time in milliseconds to the console.

The following code shows `getCNTPCT()`:

```
    .global getCNTPCT
    .type getCNTPCT, %function
    getCNTPCT:                          // Return: low and high word
        MRRC p15, 0, r0, r1, c14        // Read 64-bit CNTPCT into r0 (low word)
                                        // and r1 (high word)
        BX lr
```

Spurious interrupts

In the `EL1_FIQ_Handler()` function from this example:

- The Interrupt Controller Virtual Interrupt Acknowledge Register 0 (ICV_IAR0) is read from
- The Interrupt Controller Virtual End Of Interrupt Register 0 (ICV_EOIR0) is written to

If you check the value that is read into r8 it is not 26, which is the Exception level 2 physical countdown timer ID. The value in r8 is 1023, which is the code for a spurious interrupt ID. This tells you that this exception was not raised in response to a real virtual interrupt. The GIC was never aware of it, and the read from ICV_IAR0 to acknowledge the virtual FIQ and the write to ICV_EOIR0 to terminate the virtual FIQ are not required. This means that the quick and easy method for forwarding virtual FIQs and IRQs never creates any virtual exceptions.

The quick and easy method

The quick and easy method of forwarding virtual asynchronous exceptions can be used to raise an exception at Exception level 1 in response to an interrupt being trapped at Exception level 2. This means that the quick and easy method has the same effect as creating a virtual interrupt using the GIC, which also results in an exception being raised at Exception level 1.

Compared with using the GIC, the quick and easy method has limitations:

- A corresponding physical interrupt must first be trapped. An Exception level 2 process can program the GIC at any time to create a virtual interrupt.
- There is no way to identify or prioritize the virtual interrupt unless, for example, there is only one FIQ being received. Maintaining control of the situation might become increasingly difficult as more FIQs are virtualized.

The quick and easy method is suited for virtualizing asynchronous aborts. The GIC cannot create aborts, and they are not identifiable in the way FIQs and IRQs are. After an abort, you might want to clean up or do some debug at both Exception level 2 and Exception level 1. To perform a debug, use the Forward Virtual Asynchronous Abort Exceptions flag.

7 Guest OS switcher with virtual interrupts example

In this example, you learn how to use the GIC to create virtual interrupts, and about virtual system timers and virtual machine IDs. This example also shows an increase in the complexity of the guest OS switching routine.

This example uses the following files:

- `hyp_start.s`: This file contains hypervisor initialization code, interrupt handling code, and guest OS switching code. All code runs at the Exception level 2 privilege level.
- `main.c`: This file contains C code, including code for message printing for the monitor.
- `guest_os1.s and guest_os2.s`: These files describe the target memory map to the linker, which enables the linker to place the data and code for the example at the correct addresses in memory.
- `scatter.scat`: This file describes the target memory map to the linker, which enables the linker to place the data and code for the example at the correct addresses in memory.
- `Armv8-R Virtualization Example 3 - Cortex-R52x1 FVP.launch`: This file is the Arm Development Studio IDE debug configuration for this example project.
- `Armv8-R Virtualization Ex3 Guest OS 1`: This file creates the first guest OS binary file this project uses.
- `Armv8-R Virtualization Ex3 Guest OS 2`: This file creates the second guest OS binary file this project uses.

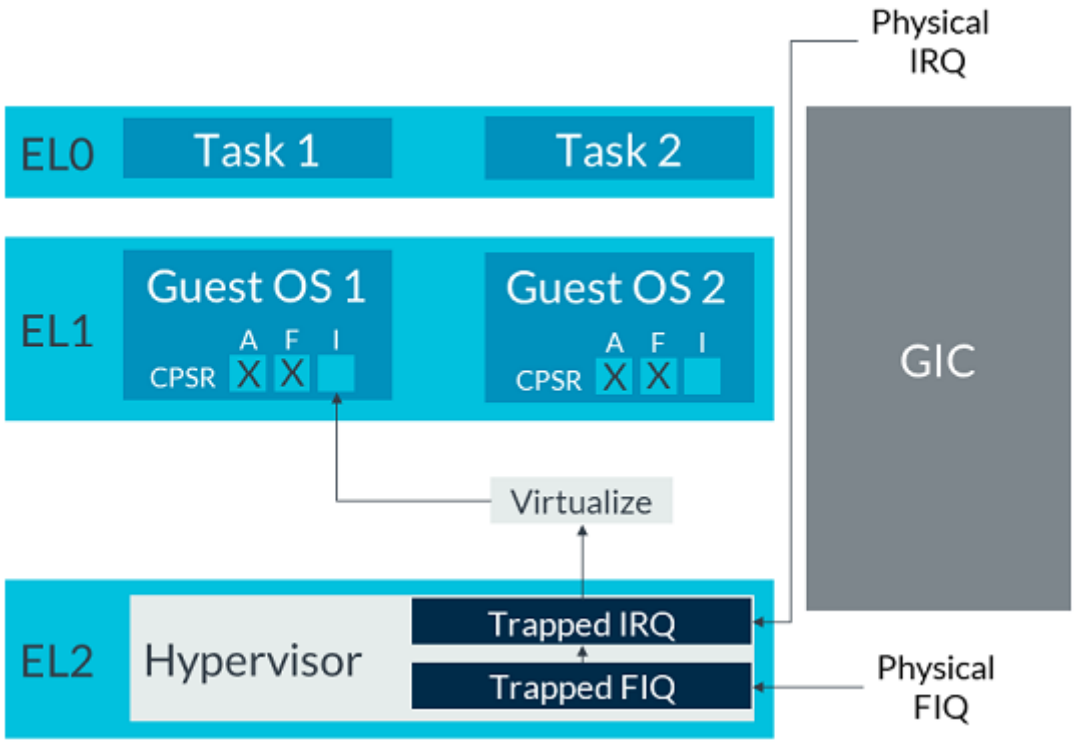
This example extends Simple guest OS switcher example to use virtual interrupts. Functionality from the single OS in OS monitor example is added to both guest OSs in this example. The physical IRQs controlling this functionality in OS monitor example are virtual IRQs for this example.

The C function from OS monitor example, which displays a millisecond count in hours, minutes, and seconds is added to each guest OS. The result is that second ticks are displayed first for guest OS 1. After four seconds, the Exception level 2 hypervisor process switches to guest OS 2, and second ticks are displayed for guest OS 2. Four seconds later, guest OS 1 becomes active again with the second count picking up from where it left off. However, this example differs from OS monitor example, because the interrupts calling `_clockTick()` for each OS are virtual, rather than physical. OS monitor example uses a single OS and only uses physical interrupts.

A second piece of functionality from OS monitor example is also virtualized: the virtual system timer is used to track how long each guest OS has been running.

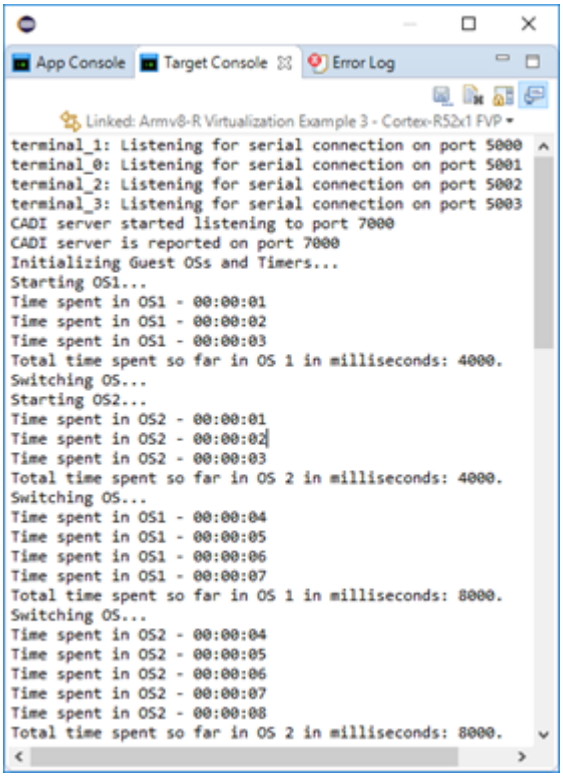
The following figure shows the scenario in this example:

Figure 7-1: Guest OS switcher example



The following screenshot shows the Target Console output for this example in the Arm Development Studio Integrated Development Environment:

Figure 7-2: Guest OS Target Console output



Ensure the code runs for core 0

Before you explore the GIC configuration code, look at `All_Cores_Except_0_To_Sleep()`, which selects core 0 as the active core and puts the other cores to sleep. The function reads from the Multiprocessor Affinity Register (MPIDR), which contains the ID of the core. The function uses the hierarchical addressing system that is covered in the section Routing in the GIC in Generic Interrupt Controller.

To read from the MPIDR use MRC as follows:

```
    MRC p15, 0, <Rt>, c0, c0, 5 ; Read MPIDR into a general register
```

The following code is from `All_Cores_Except_0_To_Sleep()`:

```
        // Put any Core other than 0 to sleep.
        // Code should only run on Core 0 in this example.
        // It is possible to find out which Core is currently in use by checking the
        // Multiprocessor Affinity Register (MPIDR). At affinity level 0, the cores
are
        // identified numerically from 0 to 3.
        MRC p15, 0, r0, c0, c0, 5            // Read MPIDR into r0.
        AND r0, r0, #0xFF                     // Isolate bits used for affinity
level 0.
        CMP r0, #0x0
        BEQ boot                             // If bits 0-7 are all 0 this is Core
0.
    sleep:
```

```
        // For Cores other than 0.
        WFI
        B sleep
    boot:
        MOV pc, lr
```

The code is run as part of the first three examples in this guide. It queries the Affinity Level 0 setting, to find out which core the software is running on. In any core other than 0, it pauses on a WFI instruction. The code does not need to route anything, so it can ignore the Affinity Level 1 and Affinity Level 2 settings.

Distributor and Redistributor register memory-mapping

The registers for the Distributor and Redistributor components are memory-mapped. Refer to the Arm Cortex-R52 Processor Technical Reference Manual to find the base address for these components. You can then apply the offsets that are found in the Arm Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0. The base addresses are specifically for the Cortex-R52, as shown in the following code:

```
    // The base address of the Distributor registers.
    .equ DISTRIBUTOR_BASE_ADDRESS,          0xAF000000
    …
    // The base address of Core 0 Redistributor registers.
    .equ CORE0_REDISTRIBUTOR_BASE_ADDRESS,      0xAF100000
    …
    // The base address of Core 0 Redistributor SGI/PPI registers.
    .equ CORE0_REDISTRIBUTOR_SGI_PPI_BASE_ADDRESS,    0xAF110000
```

Observe the following about these addresses:

- A second Redistributor base address is required for registers dealing specifically with the SGIs and PPIs
- The Redistributor base addresses are specific to core 0. Other cores have other Redistributor base addresses.

Configure the Distributor

The Distributor is configured using the GIC Distributor Control Register (GICD_CTLR). The following code is from `Basic_GIC_Setup()` and demonstrates GICD_CTLR configuration:

```
    // Redistributor register offsets
    .equ GICD_CTLR_OFFSET,                  0x0
    …
        // Use GIC Distributor Control Register (GICD_CTLR) to enable interrupts
        // and affinity routing. If interrupts are not switched on nothing can
happen.
        // Note that Armv8-R processors only support a single security state
        // and this is reflected in the limited settings for this register.
        LDR      r0, =DISTRIBUTOR_BASE_ADDRESS
        ADD      r1, r0, #GICD_CTLR_OFFSET           // Load address of GICD_CTLR
into
                                                     // r1. ADD just for
completeness
                                                     // as offset for GICD_CTLR is
0.
        MOV      r2, #0x1                            // Enable Group 0 interrupts.
        ORR      r2, #0x2                            // Enable Group 1 interrupts.
        STR      r2, [r1]                           // Store value in GICD_CTLR.
```

```
        DSB      SY
```

After globally enabling both FIQs and IRQs, the next step is to keep looping until the Register Writes Pending flag is set to `0b0`, as shown in the following code:

```
        MOV r3, #0x80000000
    GICD_CTLR_wait:
        LDR r2, [r1]
        AND r2, r2, r3                      // Test whether bit 31 is set.
        CMP r2, r3
        BEQ GICD_CTLR_wait                  // Continue polling until it is not
set.
```

When the Register Writes Pending flag is not set, it guarantees that the Distributor register settings are visible to all components that are controlled by the Distributor. For more information, see the section Distributer Control Register in the Arm Cortex-R52 Processor Technical Reference Manual.

Wake up the child core of a Redistributor

A Redistributor must first wake up the core that it services. In this example, this is core 0 and is achieved using the GIC Redistributor Wake Register (GICR_WAKER) for core 0.The following code is from `Basic_GIC_Setup()` and demonstrates using the GICR_WAKER to wake up child cores:

```
    .equ GICR_WAKER_OFFSET,                 0x14
    …
        // Mark core 0 as awake in the Core 0 Redistributor and (if required)
        // mark core 1 as awake in the Core 1 Redistributor.
        MRC p15, 0, r0, c0, c0, 5            // Read MPIDR into r0.
        AND r0, r0, #0xFF                    // Isolate bits used for affinity
level 0.
        // Core ID now in r0.
        CMP r0, #0x1
        BEQ core1                       // If bits 0-7 are set to 1 this is core 1.
        // Assume core 0 if here - good enough for these examples.
        LDR      r0, =CORE0_REDISTRIBUTOR_BASE_ADDRESS
        B all
    core1:
        LDR      r0, =CORE1_REDISTRIBUTOR_BASE_ADDRESS
    all:
        // Use the GIC Redistributor Wake Register (GICR_WAKER) to do this.
        // Set bit 1 to 0 o indicate the core is not in (or entering) low power
state.
        ADD      r1, r0, #GICR_WAKER_OFFSET
        MOV      r2, #0x0
        STR      r2, [r1]
        DSB      SY

        // Now wait until bit 2 is set to 0, which indicates that the CPU interface
        // (and therefore core) connected to the Resdistributor is awake.
        MOV      r3, #0x4
    GICR_WAKER_wait:
        LDR      r2, [r1]
        AND      r2, r2, r3                  // Test whether bit 2 is set.
        CMP      r2, r3
        BEQ      GICR_WAKER_wait             // Continue polling until it is not
set.
```

In the preceding code, the Child Core Sleeping flag is set to `0b0` to wake the core. The next step is to keep looping until the Child CPU Interface Asleep flag is not set, which confirms that the child CPU interface and the core of the Redistributor are awake.

For more information, see the section Redistributor Wake Register in the Arm Cortex-R52 Processor Technical Reference Manual.


Note

This code also wakes core 1 in SPIs and SGIs example. All examples in this guide use core 0.

Enable FIQs and IRQs for a core

FIQs and IRQs can be enabled or disabled for each core. To do this, use the Interrupt Controller Interrupt Group 0 Enable register (ICC_IGRPEN0) for FIQs, and the Interrupt Controller Interrupt Group 1 Enable register (ICC_IGRPEN1) for IRQs.To use the ICC_IGRPEN0 register, use MRC and MCR as follows:

```
MRC p15, 0, <Rt>, c12, c12, 6 ; Read ICC_IGRPEN0 into a general register
MCR p15, 0, <Rt>, c12, c12, 6 ; Write a general register into ICC_IGRPEN0
To use the ICC_IGRPEN1 register use MRC and MCR as follows:
MRC p15, 0, <Rt>, c12, c12, 7 ; Read ICC_IGRPEN1 into a general register
MCR p15, 0, <Rt>, c12, c12, 7 ; Write a general register into ICC_IGRPEN1
```

The following code is from `Basic_GIC_Setup()` and enables FIQs and IRQs for a core:

```
        MOV r0, #0x1                    // Set bit 1 to enable interrupts.
        MCR p15, 0, r0, c12, c12, 6     // Write r0 to ICC_IGRPEN0.
        ISB
        MCR p15, 0, r0, c12, c12, 7     // Write r0 to ICC_IGRPEN1.
        ISB
```

For more information, see the section Interrupt Controller Interrupt Group 0 Enable Register in the Arm Cortex-R52 Processor Technical Reference Manual.

Set the priority filter for a core

The priority filter of a core acts as a barrier, and filters out interrupts that do not have a high enough priority. The priority filter is set using the Interrupt Controller Interrupt Priority Mask Register (ICC_PMR). To use the ICC_PMR, use MRC and MCR as follows:

```
MRC p15, 0, <Rt>, c4, c6, 0 ; Read ICC_PMR into a general register
MCR p15, 0, <Rt>, c4, c6, 0 ; Write a general register into ICC_PMR
```

The following code is from `Basic_GIC_Setup()` and sets the priority filter to 31 so that the core handles all interrupts it receives:

```
        MOV r0, #0xFF           // Set priority filter so the core handles
everything.
                                // Set all 8 priority filter bits even though the
```

```
                                // Cortex-R52 only use the most significant 5 bits.
        MCR p15, 0, r0, c4, c6, 0    // Write r0 to ICC_PMR.
        ISB
```

For more information, see the section Interrupt Controller Interrupt Priority Mask Register in the Arm Cortex-R52 Processor Technical Reference Manual.

This code only sets the priority filter for exceptions or interrupts raised at Exception level 2. This code does not set the priority filter for exceptions at Exception level 1, which is indicated in the ICC_ prefix to the register. To set the priority filter for exceptions that are raised at Exception level 1, swap to an Exception level 1 mode and then use the Interrupt Controller Virtual Interrupt Priority Mask Register (ICV_PMR). The procedure for this is the same as setting the ICC_PMR. Alternatively, when using all ICC_ registers you can remain at Exception level 2 and use the Interrupt Controller Virtual Machine Control Register (ICH_VMCR) in this example. This example uses the second approach.

Set the binary point for FIQs and IRQs

The binary point is used to determine which interrupts, if any, can preempt others. The binary point is linked to the priority filter of a core. Binary points can be set independently for FIQs and IRQs, using the Interrupt Controller Binary Point Register 0 (ICC_BPR0) and Interrupt Controller Binary Point Register 1 (ICC_BPR1). To use the ICC_BPR0 use MRC and MCR as follows:

```
    MRC p15, 0, <Rt>, c12, c8, 3 ; Read ICC_BPR0 into a general register
    MCR p15, 0, <Rt>, c12, c8, 3 ; Write a general register into ICC_BPR0
```

To use the ICC_BPR1 use MRC and MCR as follows:

```
    MRC p15, 0, <Rt>, c12, c12, 3 ; Read ICC_BPR1 into a general register
    MCR p15, 0, <Rt>, c12, c12, 3 ; Write a general register into ICC_BPR1
```

Preemption is not used in the examples in this guide. That is, at Exception level 2 one interrupt cannot preempt another. In this guide, the handlers that are responsible for switching between guest OSs and FIQ handlers and the IRQ handlers that are responsible for timer ticks cannot preempt another interrupt.

For reasons beyond the scope of this guide, you cannot set the binary point for IRQs to .sssss , or no preemption. This means that IRQs can always potentially interrupt FIQs that have a setting of 1 in the most significant priority bit (bit 5). To prevent this issue, either:

- Use the g.ssss setting for IRQs and make sure that IRQs always have bit 5 set to 1. In this case, all IRQs have a priority in the range 16-31 or
- Set the ICC_CTLR Use Single Binary Point Register flag. When this flag is set, the ICC_BPR0 sets the binary point for both FIQs and IRQs.

This example uses the flag setting method. When the flag is set, the following code from `Basic_GIC_Setup()` switches off preemption for FIQs and IRQs:

```
    // Specify which bits from the priority bits are used as the group priority
        // field. The group priority field is used to determine preemption.
 Preemption
        // is when a lower priority interrupt already being handled by the core gets
        // put on hold while a higher priority interrupt is handled. The point at
 which
        // the more significant bits are used for the group priority field is known
 as
        // the binary point.
        // The following registers are used to set the binary point:
        // 1. Interrupt Controller Binary Point Register 0 (ICC_BPR0)
        // 2. Interrupt Controller Binary Point Register 1 (ICC_BPR1)
        MOV r0, #0x7                // Set to no preemption (.sssss) for Group 0.
                                    // In other words, no bits are used for
                                    // preemption for Group 0.
                                    // Set to one bit used for preemption (g.ssss)
                                    // for Group 1.
        MCR p15, 0, r0, c12, c8, 3  // Write r0 to ICC_BPR0 to set this for Group 0.
        ISB
        MCR p15, 0, r0, c12, c12, 3 // Write r0 to ICC_BPR1 to set this for Group 1.
        ISB

MOV pc, lr
```

For more information, see the section Interrupt Controller Binary Point Register in the Arm Cortex-R52 Processor Technical Reference Manual.

Configure individual physical interrupts

In this example, two interrupts are configured: the Exception level 2 physical countdown timer, an FIQ, and the virtual countdown timer.

Specify how to raise an SGI or PPI

To specify whether to raise an SGI or PPI as an FIQ or an IRQ, use the Interrupt Group Register 0 (GICR_IGROUPR0). The following code sets the timer interrupts up for this example:

```
    .equ GICR_IGROUPR0_OFFSET,                    0x80
    …
    // The base address of Core 0 Redistributor SGI/PPI registers.
    .equ CORE0_REDISTRIBUTOR_SGI_PPI_BASE_ADDRESS,    0xAF110000
    …
        // Specify interrupt 26 to be a Group 0 interrupt and therefore an
        // FIQ on a v8-R series processor.
        // Specify interrupt 27 to be a Group 1 interrupt and therefore an
        // IRQ on a v8-R series processor.
        // Note that both interrupts are PPIs and are configured through registers
 in the
        // SGI/PPI part of the Redistributor interface.
        // To set the group to which different SGI/PPIs belong to, use the
        // Interrupt Group Register 0 (GICR_IGROUPR0)
        LDR      r0, = CORE0_REDISTRIBUTOR_SGI_PPI_BASE_ADDRESS
        ADD      r1, r0, #GICR_IGROUPR0_OFFSET
        MOV      r2, #(1 << 27)  // Set bit 27 to 1 so interrupt 27 is in Group 1.
                                 // All other SGIs/PPIs are considered Group 0,
                                 // but only interrupt 26 is used in this example.
        STR      r2, [r1]        // Store information in GICR_IGROUPR0.
        DSB      SY
```

For more information, read about how individual interrupts are configured in Simple guest OS switcher example and OS monitor example, and refer to the section Interrupt Group Register 0 in the Arm Cortex-R52 Processor Technical Reference Manual.

Set the individual priorities for SGIs and PPIs

To set the priorities for individual SGIs or PPIs, use the Interrupt Priority Registers (GICR_IPRIORITYR0-7). The following code sets the priority for the timers that are used in this example:

```
   .equ GICR_IPRIORITYR_OFFSET,                    0x400 // The base address of
eight
                                                         // registers holding the
                                                         // priority for 32
interrupts
   …
       LDR      r0, = CORE0_REDISTRIBUTOR_SGI_PPI_BASE_ADDRESS
   …
       // Set the priority for interrupt 26 and 27 on Core 0 using the
       // Interrupt Priority Registers (GICR_IPRIORITYR).
       // These eight registers hold the priorities for the 16 SGIs and 16 PPIs.
       // One byte is used per interrupt.
       ADD      r1, r0, #GICR_IPRIORITYR_OFFSET
       ADD      r1, r1, #(6*4)  // Access register 6 (interrupts 24-27)
       LDR      r2, =0x1878FFFF // Set interrupt 26 to have a priority of 15.
                                // Set interrupt 27 to have a higher priority (3).
                                // We do not use interrupts 24 and 25 so we just
set
                                // them to the lowest priority (31).
       STR      r2, [r1]
```

In this case, the priority of the Exception level 2 physical timer is set to 15, and the priority of the virtual timer is set to 3. This is because the IRQs that are raised by the virtual timer must be handled before any FIQs raised by the physical timer. By setting these priorities, the Exception level 2 physical IRQ from one guest OS must be handled before the process of switching to the other guest OS begins. This process is more complicated because virtual interrupts are involved. To learn more, see [[Backing up Exception level 2 physical interrupt state].

For more information, refer to the section Interrupt Priority Registers 0-7 in the Arm Cortex-R52 Processor Technical Reference Manual.

Enable and disable SGIs and PPIs

To specify whether an SGI or PPI is enabled or disabled, use the Interrupt Set-Enable Register 0 (GICR_ISENABLER0). The following code enables the Exception level 2 physical timer and the virtual timer:

```
   .equ GICR_ISENABLER0_OFFSET,                    0x100
   …
       LDR      r0, = CORE0_REDISTRIBUTOR_SGI_PPI_BASE_ADDRESS
   …
       // Enable interrupt 26 and 27 on Core 0 using the
       // Interrupt Set-Enable Register 0 (GICR_ISENABLER0).
       // This register holds the enabling flags for the 16 SGIs and 16 PPIs.
       ADD      r1, r0, #GICR_ISENABLER0_OFFSET
       MOV      r2, #(3 << 26) // Set bits 26 and 27 to 1.
       STR      r2, [r1]       // Store information in GICR_ISENABLER0.
```

For more information, refer to the section Interrupt Set-Enable Register 0 in the Arm Cortex-R52 Processor Technical Reference Manual.

Prepare to use virtual interrupts

The next step in our example is to prepare a virtual interrupt to send in response to a physical IRQ from the Virtual Timer. Registers from both the hypervisor CPU interface and the Exception level 2 CPU interface are used to achieve this.

Enabling the hypervisor CPU interface

First, enable the hypervisor CPU interface using the Interrupt Controller Hyp Control Register (ICH_HCR). To use the ICH_HCR, use MRC and MCR as follows:

```
MRC p15, 4, <Rt>, c12, c11, 0 ; Read ICH_HCR into a general register
MCR p15, 4, <Rt>, c12, c11, 0 ; Write a general register into ICH_HCR
```

The following code enables the hypervisor CPU interface:

```
MRC p15, 4, r0, c12, c11, 0
ORR r0, r0, #0x1            // Set bit 0 to enable.
MCR p15, 4, r0, c12, c11, 0
```

For more information, refer to the section Interrupt Controller Hyp Control Register in the Arm Cortex-R52 Processor Technical Reference Manual.

Maintenance interrupts

Maintenance interrupts are PPIs that have a maintenance ID of 25. It is possible, for example, to specify that maintenance interrupts are FIQs and then trap them in `EL2_FIQ_Handler()`. Because the use of maintenance interrupts is often related to management of more than four different virtual interrupts in a system, they are not included in this example. To use maintenance interrupts, lear about the Interrupt Controller Maintenance Interrupt State Register (ICH_MISR). You can interrogate the ICH_MISR to find out the reason behind a maintenance interrupt. To learn more about this register, see the section Interrupt Controller Hyp Maintenance Interrupt Status Register Exception level 2 in the Arm Cortex-R52 Processor Technical Reference Manual.

Set up a virtual interrupt

You must set up virtual interrupt with an interrupt ID of 27, which is called in response to the physical IRQ that is generated by the virtual timer. The Interrupt Controller List Registers ICH_LR0-3 and ICH_LRC0-3 are used to achieve this. The Cortex-R52 processor supports four ICH_LRs and four ICH_LRCs. To use the ICC_LRn use MRC and MCR as follows:

```
MRC p15, 4, <Rt>, c12, c12, n ; Read ICC_LRn into a general register
MCR p15, 4, <Rt>, c12, c12, n ; Write a general register into ICC_LRn
```

To use the ICC_LRCn, use MRC and MCR as follows:

```
MRC p15, 4, <Rt>, c12, c14, n ; Read ICC_LRCn into a general register
MCR p15, 4, <Rt>, c12, c14, n ; Write a general register into ICC_LRCn
```

In this example, only the ICH_LR0 and ICH_LRC0 must be configured. This is because there is only one virtual interrupt to set up. The following code shows the configuration of ICH_LR0 and ICH_LRC0:

```
        MOV r0, #27                  // Set the interrupt ID.
        MCR p15, 4, r0, c12, c12, 0  // Write r0 to ICH_LR0.
        ISB

        MOV r0, #27            // Set bits 0-9 to the interrupt ID.
                               // Bits 16 to 23 set the interrupt priority,
                               // which in this case is set to 0 (the highest
priority)
        ORR r0, r0, #(1 << 28) // Set bit 28 to indicate that this is a Group 1
                               // interrupt and therefore an IRQ.
        ORR r0, r0, #(1 << 29) // Set bit 29 to indicate that this interrupt maps
                               // directly to a hardware interrupt.
        MCR p15, 4, r0, c12, c14, 0  // Write r0 to ICH_LRC0
        ISB
```

In summary, the virtual interrupt:

- Has an interrupt ID of 27
- Has a priority of 0, which is the highest priority. The Exception level 2 physical interrupts in this example still take priority over the virtual interrupts even though their priorities are lower. These interrupts have a priority greater than 0.
- Is an IRQ
- Is directly related to a hardware interrupt. In this case, it is related to the physical IRQ that is generated by the virtual timer.

When configured, you must set the virtual interrupt to pending to use it. For more information about the lifecycle of interrupts, refer to Interrupt lifecycle in the GIC and the section Hypervisor Control System Registers in the Arm Cortex-R52 Processor Technical Reference Manual.

Set the Exception level 2 CPU interface with virtual interrupts

When an interrupt has been handled, write to the write-only Interrupt Controller End Of Interrupt Register (ICC_EOIR) to deactivate it. You can see an example in Switch guest OSs. Deactivating the interrupt is the default behavior of the ICC_EOIR. However, if you deactivate a physical interrupt that is linked to a virtual interrupt, this also deactivates the virtual interrupt. To avoid deactivating the virtual interrupt, set the EOI Register Writes Only Drop Interrupt Priority flag on the Interrupt Controller Control Register (ICC_CTLR). The following code from hyp_start.s shows the two flags that are set in the ICC_CTLR:

- The EOI Register Writes Only Drop Interrupt Priority flag is set. Writing to the ICC_EOIR drops the priority of the physical interrupt.
- The Use Single Binary Point Register flag is set, so that ICC_BPR0 defines the binary point for both FIQs and IRQs.

```
      MRC p15, 0, r0, c12, c12, 4 // Read ICC_CTLR into r0.
      ORR r0, r0, #0x3             // Set bit 1 and 0.
      MCR p15, 0, r0, c12, c12, 4 // Write r0 to ICC_CTLR.
```

To learn more, see the section Interrupt Controller Control Register (Exception level 1) in the Arm Cortex-R52 Processor Technical Reference Manual.

Configure the virtual CPU interface

The next step in our example is to configure the virtual CPU interface, which controls the lifecycle of virtual interrupts. The virtual CPU interface consists of the ICV_ registers, which are banked ICC_ registers for Exception level 1. Because the ICV_ registers use the same instruction encoding as ICC_ registers, you must set them using the following ICH_ registers if you are in an Exception level 2 process:

- Interrupt Controller Virtual Machine Control Register (ICH_VMCR)
- Interrupt Controller Hyp Active Priorities Group 0 Register 0 (ICH_AP0R0)
- Interrupt Controller Hyp Active Priorities Group 1 Register 0 (ICH_AP1R0)

The ICH_VMCR is designed so that multiple flags and settings can be specified and covers several ICV_ registers.

To use the ICH_VMCR, use MRC and MCR as follows:

```
   MRC p15, 4, <Rt>, c12, c11, 7 ; Read ICH_VMCR into a general register
   MCR p15, 4, <Rt>, c12, c11, 7 ; Write a general register into ICH_VMCR
   The following code shows how the virtual CPU interface is configured using the
ICH_VMCR:
       // Use ICH_VMCR to:
       // 1. Enable the interrupts for Group 1. There is no need to turn on Group 0
       // interrupts as virtual FIQs are not used.
       // 2. Set the priority filter. A value of 31 indicates the lowest
       // possible priority and a value of 0 indicates the highest.
       // 3. Set the binary point for Group 1 (IRQs). Determines preemption.
       MOV r0, #0x2              // Set bit 2 to enable virtual IRQs.
       ORR r0, r0, #(0xFF << 24) // Set bits 24 to 31 to 255. This means the core
                                 // handles all virtual IRQs whatever their
priority.
       ORR r0, r0, #(0x0 << 18)  // Bits 18 to 20 are set to 0 to
                                 // specify no preemeption (.sssss) for virtual
IRQs.
       MCR p15, 4, r0, c12, c11, 7  // Write r0 to ICH_VMCR.
       ISB
```

In summary, the virtual machines for both guest OSs:

- Do not filter out any virtual interrupts
- Do not allow preemption for virtual interrupts
- Deactivate virtual IRQs when ICV_EOIR1 is written to
- Use a separate register for IRQ binary points and FIQ binary points
- Have virtual IRQs enabled
- Have virtual FIQs enabled, even though none exist in this example)

Although both guest OSs are initialized in the same way in this example, they could be configured with different startup settings.

These registers also have a secondary function: they enable information about the current state of the CPU interface of a guest OS to be backed up when a hypervisor switches to another guest OS. In this example, code to initialize, back up, and restore all three registers has been added to `init_guest_os()` and `switch_guest_os()`. The guest OS data structures have also been updated to store a copy of these registers for each guest OS.

For more information, refer to the following sections of the Arm Cortex-R52 Processor Technical Reference Manual:

- Interrupt Controller Virtual Machine Control Register
- Interrupt Controller Hyp Active Priorities Group 0 Register 0

In this section of the guide, the exception handlers are modified to handle the scenario in this example.

Modify the exception handlers

`EL2_FIQ_Handler()` switches between the two guest OSs and in this example, is called every four seconds rather than every two seconds. The version of `switch_guest_os()` this example calls is more sophisticated and is covered in Switch guest OSs.

`EL2_FIQ_Handler()` calls a C function that logs the time in milliseconds spent so far in the active guest OS. To ensure that the output is accurate, the handler calls timeInOSInMilliseconds() just before switching to the other guest OS. The timeInOSInMilliseconds() output is the same functionality in OS monitor example. However, in this example, `EL2_FIQ_Handler()` does not generate a virtual exception and drop into an Exception level 1 process to output the message.

A branch is made to `__main()` in `hyp_start.s`. The `__main()` function initializes the standard library for the hypervisor Exception level 2 process, so that `EL2_FIQ_Handler()` can call `printf()`. This example initializes the standard C library for the hypervisor Exception level 2 process and the two guest OS Exception level 1 processes. The code branches directly back into hyp_start.s afterwards and completes the initialization of the system.

`EL2_IRQ_Handler()` creates virtual exceptions using the GIC. In this guide, the method to forward FIQ and IRQ exceptions is referred to as the GIC method. The virtual exception that `EL2_IRQ_Handler()` creates every millisecond allows both guest OSs to function like the OS in OS monitor, and the guest OS responds to virtual IRQs.

In this example, the millisecond countdown timer is reset in `EL2_IRQ_Handler()` and uses the virtual countdown timer (interrupt ID 27).

Each guest OS in this example has an EL1_IRQ_Handler() that is similar to the EL1_IRQ_Handler() used for the OS in OS monitor. The `EL1_IRQ_Handler()` in this example does not reset a counter timer. This is because this reset is performed in the `EL2_IRQ_Handler()`.

The following line of code is also added for each guest OS, to ensure that the OSs receive the virtual IRQs:

```
        CPSIE i
```

GIC registers used in the handlers

Several registers are used in the handlers to acknowledge interrupts, deprioritize interrupts, and to deactivate both FIQs and IRQs. These registers contain a single Interrupt ID setting, and all have the same layout:

- ICC_IAR0
- ICC_IAR1
- ICC_EOIR0
- ICC_EOIR1
- ICC_DI

To read from ICC_IAR0, use MRC as follows:

```
    MCR p15, 0, <Rt>, c12, c8, 0 ; Read ICC_IAR0 into a general register
```

To read from ICC_IAR1, use MRC as follows:

```
    MCR p15, 0, <Rt>, c12, c12, 0 ; Read ICC_IAR1 into a general register
```

To write to ICC_EOIR0, use MCR as follows:

```
    MCR p15, 0, <Rt>, c12, c8, 1 ; Write a general register into ICC_EOIR0
```

To write to ICC_EOIR1, use MCR as follows:

```
    MCR p15, 0, <Rt>, c12, c12, 1 ; Write a general register into ICC_EOIR1
```

To write to ICC_DIR, use MCR as follows:

```
    MCR p15, 0, <Rt>, c12, c11, 1 ; Write a general register into ICC_DIR
```

For more information, see the section CPU Interface Registers in the Arm Cortex-R52 Processor Technical Reference Manual.

Switch guest OSs

The `switch_guest_os()` function in this example puts ten registers on the stack. This is because `EL2_FIQ_Handler()` also calls the `timeInOSInMilliseconds()` C function and `switch_guest_os()` is more complex. In this example, more registers are backed up when the handler is entered, and the

guest OS values in two general registers are put on the stack immediately. These two registers are then available for the handler and the functions that it calls.

The following table lists the additional registers that are added to the backup and restore process:

Register	Notes
GICR_ISACTIVER0	Contains information about which physical SGIs and PPIs are currently active or active and pending. In this example, this information is recorded for the virtual timer (interrupt ID 27). More information about why this register is backed up can be found in Back up Exception level 2 physical interrupt state.
GICR_ISPENDR0	Contains information about which physical SGIs and PPIs are pending, or active and pending. In this example, this information is recorded for the virtual timer (interrupt ID 27). More information about why this register is backed up can be found in Back up Exception level 2 physical interrupt state.
ICC_AP1R0	Contains the active priorities for the physical IRQs. The active priorities are any physical interrupts that become active between the time ICC_IAR1 is called and the time ICC_EOIR1 is called. However, because the `EL2_IRQ_Handler()` cannot be preempted in this example, it is not expected that this register shows any active physical IRQs when guest OSs are switched in the `EL2_FIQ_Handler`. Also, the GIC is configured so the ICC_EOIR1 only drops the priority to the lowest level (the idle priority) until the linked virtual interrupt terminates. You should not expect to see bit 31 set in this register because the bits in this register are set when an interrupt is acknowledged using ICC_IAR0 or ICCIAR1. The bits are not set by any other method, for example an ICC_EOIR1 priority drop. For completeness, this register is still backed up and restored.
CNTVCT	Contains the virtual system time, which is the number of ticks that have elapsed since the guest OS started running. The restoration process is more complex because this register is read-only. To restore the register, subtract the backed-up value for CNTVCT from the physical system time then write the result to the Counter-timer Virtual Offset Register (CNTOFF). You can find more information in Use the virtual system timer.
CNTV_TVAL	Contains the current value for the virtual countdown timer. Countdown timers work by setting a compare value that is compared with the physical or virtual system timer. In this case, the comparison is made with CNTVCT therefore, CNTVCT must be restored before CNTV_TVAL.
ICH_LR0	Contains the interrupt ID of the virtual countdown timer, which is responsible for the only virtual interrupt in this example. The value of this register does not change throughout this example although it is still backed up and restored. If other virtual interrupts are required, the Cortex-R52 processor provides three more slots for virtual interrupts. According to the GICv3 specification, processors can theoretically have up to 16 slots for virtual interrupts. However, the exact number of slots available is processor-specific. If more virtual interrupts are required than slots are available, the slots usage must be managed and the content of the ICH_LR* registers changes over time. In this case, back up and restore the registers.
ICH_LRC0	Contains details about the virtual interrupt that is identified by ICH_LR0. This includes information on its state, its priority, whether it is an IRQ or an FIQ, and whether it is related to a real physical interrupt. The interrupt state which the ICH_LRC* registers record changes as a guest OS runs. Therefore, backing up and restoring these registers is essential.
ICH_VMCR	Contains the settings for the ICV_* registers. It is essential to back up this register because the ICV_* register settings can change when the OS handles virtual interrupts
ICH_AP0R0	Contains the active priorities for the virtual FIQs. The active priorities are the priorities of any virtual FIQs which have been acknowledged using ICV_IAR0 but have not been terminated using ICV_EOIR0. There are no virtual FIQs used in this example. However, for completeness, this register is still backed up and restored.
ICH_AP1R0	Contains the active priorities the virtual IRQs. The active priorities are the priorities of any virtual IRQs which have been acknowledged using ICV_IAR1 but have not been terminated using ICV_EOIR1. It is important to back up this register because virtual IRQs are used in this example. In addition, physical interrupts can preempt virtual interrupts. In this case, that means the code execution could jump to `EL2_FIQ_Handler()` in between the ICV_IAR1 read and the ICV_EOIR1 write in `EL1_IRQ_Handler()`.
VSCTLR	Identifies the guest OS currently active. The register is only accessible from an Exception level 2 process.

Back up Exception level 2 physical interrupt state

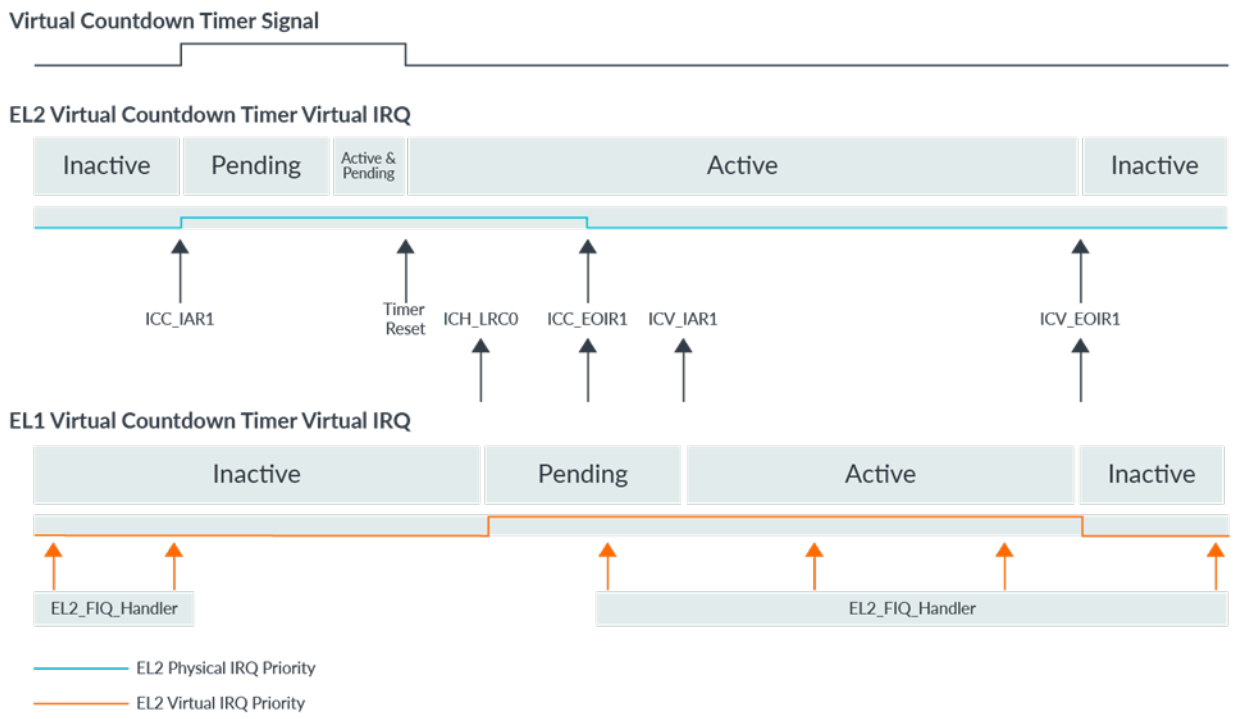
You can use the Interrupt Set-Pending Register 0 (GICR_ISPENDR0) to set an SGI or PPI to pending and use the Interrupt Set-Active Register 0 (GICR_ISACTIVER0) to set an SGI or PPI to active. Both registers can also be used to check the status of the interrupt. If GICR_ISPENDR0 and GICR_ISACTIVER0 are both set for an SGI or PPI, the state is active and pending.

Why do you need to know about state of physical interrupts that are raised at Exception level 2? In this example, the GIC is configured at Exception level 2 to only drop the priority of an interrupt when either ICC_EOIR0 or ICC_EOIR1 is written to. With a virtual timer (interrupt ID 27), the EL1_IRQ_Handler() terminates the physical interrupt. At this point, Exception level 2 processes ignore priority and preemption settings for Exception level 1, which are set using ICV_* registers. The `EL2_FIQ_Handler()` function responsible for switching guest OSs can interrupt `EL1_IRQ_Handler()` in mid flow or before it even starts. When this happens, the status of the Exception level 2 virtual timer IRQ remains set to active. This can be a problem, because the next guest OS cannot raise IRQs at Exception level 2. The solution is to back up and restore the state of the Exception level 2 virtual timer IRQ for guest OSs.

Two other registers are also of interest: GICR_ICPENDR0 and GICR_ICACTIVER0. These registers are used to clear the state of the SGIs and PPIs after they have been backed up. This backup provides a clean sheet on which to restore the state of the next guest OS. When you clear the state of the SGI and PPI, clear pending states first then active states.

The following diagram compares the states of the physical and virtual interrupts along a timeline:

Figure 7-3: Physical and virtual interrupts states



This diagram includes the priorities of the interrupts and shows how the register writes affect these priorities. The orange arrows indicate the points at which `EL2_FIQ_Handler()` can preempt the process.


Note

If you reset the timer in EL1_IRQ_Handler() before ICV_EOIR1 is written to, the timer may be inaccurate. If `EL2_FIQ_Handler()` interrupts `EL1_IRQ_Handler()`, the following code is used. This code handles countdown timer overrun and protects against unwanted effects.

The following code shows how these registers are used in `switch_guest_os()`:

```
        // GICR_ISACTIVER0 and GICR_ISPENDR0 contain the state of
        // the Virtual Timer Physical IRQ.

        LDR     r4, =CORE0_REDISTRIBUTOR_SGI_PPI_BASE_ADDRESS
        // GICR_ISACTIVER0
        ADD     r5, r4, #GICR_ISACTIVER0_OFFSET
        LDR     r2, [r5] // Grab value for active guest OS.
        // Only the bit for the Virtual Timer PPI (Interrupt ID 27) is backed up.
        AND     r2, r2, 0x08000000
        STR     r2, [r0], #4
        LDR     r3, [r1], #4 // Load value for sleeping guest OS.

        // GICR_ISPENDR0
        ADD     r5, r4, #GICR_ISPENDR0_OFFSET
        LDR     r2, [r5] // Grab value for active guest OS.
        // Only the bit for the Virtual Timer PPI (Interrupt ID 27) is backed up.
        AND     r2, r2, 0x08000000
        STR     r2, [r0], #4
        LDR     r2, [r1], #4 // Load value for sleeping guest OS.

        // Push sleeping guest OS GICR_ISPENDR0 and GICR_ISACTIVER0
        // values to the stack.
        // This preserves them while the state for the Virtual Timer Physical IRQ
        // is cleared.
        PUSH {r2, r3}

        // Clear pending, active and pending, and active states from
        // the Virtual Timer Physical IRQ.
        // This is done by writing to bit 27 of GICR_ICPENDR0
        // and GICR_ICACTIVER0.
        // The order is important. Writing to GICR_ICPENDR0
        // causes the IRQ to become:
        // 1. Inactive if it is pending.
        // 2. Active if it is active and pending.
        // Writing to GICR_ICACTIVER0 causes the IRQ to become
        // inactive if it is active.
        // Doing the cleardown in this order always returns the
        // state of the Virtual Timer Physical IRQ to inactive.
        ADD       r5, r4, #GICR_ICPENDR0_OFFSET
        MOV       r2, #0x08000000
        STR       r2, [r5]

        ADD       r5, r4, #GICR_ICACTIVER0_OFFSET
        MOV       r2, #0x08000000
        STR       r2, [r5]

        POP {r2, r3} // r3 -> GICR_ISACTIVER0, r2 -> GICR_ISACTIVER0

        // Read backed up value into registers in the following order:
        // GICR_ISACTIVER0, GICR_ISPENDR0
        // This is the opposite order to using the GICR_IC* registers
        // and enables all four possible states to be restored.
```

```
        ADD     r5, r4, #GICR_ISACTIVER0_OFFSET
        STR     r3, [r5]

        ADD     r5, r4, #GICR_ISPENDR0_OFFSET
        STR     r2, [r5]
```

For more information, see the section Redistributor Registers (GICR) in the Arm Cortex-R52 Processor Technical Reference Manual.

Use the virtual system timer

The virtual system timer is calculated using the following formula: virtual system timer = physical system timer – offset.

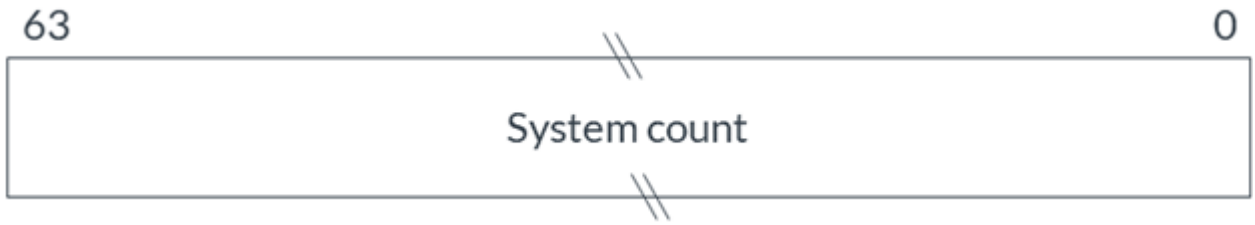
For 64-bit registers, the formula is:

CNTVCT = CNTPCT - CNTVOFF

Counter-timer Virtual Count Register = Counter-timer Physical Count Register - Counter-timer Virtual Offset Register

Only the CNTVOFF register is writeable. You must calculate the offset to set the virtual system timer. The following figure and table shows the layout for the CNTVOFF register:

Figure 7-4: System Count Layout



Name	Bits	Function	Architecture reference
Offset From Physical Timer	0:63	Specify the offset of the virtual system timer from the physical system timer.	Virtual offset

To use CNTVOFF, use MRRC and MCRR as follows:

```
    MRRC p15, 4, <Rt>, <Rt2>, c14 ; Read CNTVOFF into two 32-bit general registers
    MCRR p15, 4, <Rt>, <Rt2>, c14 ; Write two 32-bit general registers into CNTVOFF
```

In a system with multiple guest OSs, to maintain a valid virtual system timer when OSs are switched, follow these steps:

1. Store the virtual countdown timer (CNTV_TVAL) for the active guest OS on the stack.
2. Back up the virtual system time (CNTVCT) for the active guest OS to memory.
3. Retrieve the virtual system time for the sleeping guest OS from memory.
4. Calculate the offset for the sleeping guest OS as follows: Offset = physical system time (CNTPCT) - virtual system time for the sleeping guest OS

5. Restore the virtual system time for the sleeping guest OS by writing the calculated offset to CNTVOFF.
6. Back up the virtual countdown timer for the active guest OS after popping it off the stack. It is important to prevent the virtual countdown timer from underflowing before it can be backed up. When a countdown timer finishes, it wraps around to the maximum value and keeps counting down. If this happens, you must back up the starting value for the countdown timer. This backup prevents the first countdown from being too long after a guest OS switch. For more information, see the code that follows these steps.
7. Restore the virtual countdown timer (CNTV_TVAL) for the sleeping guest OS.
8. Change the sleeping guest OS to be the active guest OS.

The following code, taken from switch_guest_os(), shows you how to back up the starting value of the virtual countdown timer:

```
        // Step 1: Push CNTV_TVAL onto the stack to prevent
        // corruption.
        MRC p15, 0, r2, c14, c3, 0
        PUSH {r2}

        // CNTVCT - Counter-timer Virtual Count Register
        // Step 2:
        MRRC p15, 1, r2, r3, c14
        STR    r2, [r0], #4
        STR    r3, [r0], #4

        // Step 3:
        // CNTV_TVAL becomes corrupted when CNTVCT is restored in the
        // following code:
        LDR    r2, [r1], #4
        LDR    r3, [r1], #4

        // r2 and r3 now contain CNTVCT for the guest OS being restored.
        // However, it cannot just be restored as it is a read-only register.
        // It is instead used to write a value to the
        // Counter-timer Virtual Offset Register (CNTVOFF), which indirectly
        // restores CNTVCT.

        // Step 4a:
        //Extract the current system timer value from the
        // Counter-timer Physical Count Register (CNTPCT).
        MRRC p15, 0, r4, r5, c14    // Read 64-bit CNTPCT into r4 (low word)
                                    // and r5 (high word)

        // Step 4b: Subtract CNTVCT from CNTPCT to get the value
        // for CNTVOFF.
        // Note: Use SUBS and SBC for 64-bit subtraction.
        SUBS r4, r4, r2
        SBC r5, r5, r3

        // Step 5: Update CNTVOFF. This ensures the correct value can be read
        // from CNTVCT.
        MCRR p15, 4, r4, r5, c14

        // Step 6:
        // CNTV_TVAL - Counter-timer Virtual Timer TimerValue register
        // Important: Setting CNTV_TVAL writes a value to the
        // Counter-timer Virtual Timer CompareValue register (CNTV_CVAL).
        // This is compared with the CNTVCT to decide when to fire an interrupt.
        // Therefore you *must* update CNTVOFF (restore CNTVCT) before CNTV_TVAL.

        // First check CNTV_CTL to see if the timer has finished.
        // This can feasibly occur while the guest OS switch is happening
```

```
        // depending on the time the process takes.
        // If CNTV_TVAL finishes, it "wraps around" to an invalid,
        // large value. However, an IRQ is not pointlessly generated as
        // bit 2 of CNTV_CTL is set at the beginning of the FIQ handler.
        MRC    p15, 0, r3, c14, c3, 1
        AND    r4, r3, #0x4
        CMP    r4, #0x4
        BEQ    timer_finished
         // Get CNTV_TVAL from the stack - it will be a sensible value.
        POP {r2}
        B store_CNTV_TVAL
    timer_finished:
        // Timer finished - use "start" value as the actual timer value has
        // "wrapped" around and is not useful.
        LDR    r2, =VIRTUAL_TICK_VALUE
    store_CNTV_TVAL:
        STR    r2, [r0], #4
        // Step 7:
        LDR    r2, [r1], #4
        MCR    p15, 0, r2, c14, c3, 0
```

In `initializeGuestOSsAndStartTimers()` CNTVOFF is set to CNTPCT. This ensures that time spent in the Exception level 2 process initializing the system is not counted as time spent in guest OS 1. The CNTVCT for guest OS 2 is set to 0 in `init_guest_os()`. Applying the formula, this means that CNTVOFF also gets set to CNTPCT when guest OS 2 first runs. Time spent initializing the system and during the first run of guest OS 1 is discounted. In a system where a significant amount of time is spent in Exception level 2 processes, other than during initialization, you must back up the CNTVCT value for the active guest OS when an Exception level 2 process begins.

Specify the virtual machine ID of an OS

There are 8 bits available to specify a virtual machine ID for a guest OS in the Virtualization System Control Register (VSCTLR). For example, in a real world situation, the hypervisor can use the virtual machine ID to decide if a peripheral can be accessed depending on the active guest OS. In this example, the virtual machine ID is used with the virtual system timer to output a message about the milliseconds that have elapsed since the OS started.

The following function, which is called from `timeInOSInMilliseconds()` in `main.c`, retrieves the VSCTLR Virtual Machine ID setting:

```
    .global getVirtualID
    .type getVirtualID, "function"
    getVirtualID:
        // Read from the Virtualization System Control Register (VSCTLR)
        MRC p15, 4, r1, c2, c0, 0
        AND r1, r1, #0xFF0000      // Isolate the Virtual ID in r0
        MOV r0, r1, LSR #16

        BX      lr
```


Note

You can only access the VSCTLR from an Exception level 2 process. A guest OS does not need to be aware of its Virtual Machine Identifier (VMID).

8 SPIs and SGIs example

In this example you learn about using the GIC with SPIs and SGIs, and working with memory-mapped registers in C. This example also explains:

- How one core can generate an SGI that is received on another core
- About all the types of GIC interrupt supported by the Armv8-R architecture: SPIs, PPIs, and SGIs

Download a multi-core Fixed Virtual Platform (FVP) to run this example. You can use the FVP_BaseR_Cortex-R52x2 or FVP_BaseR_Cortex-R52x4 Fixed Virtual Platforms. These platforms are not supplied with the DS Ultimate Edition and must be purchased separately. For more information and to download a Fixed Virtual Platform, see Fixed Virtual Platforms.

This example uses the following files:

- `start.s`: This file contains initialization code which both core 0 and core 1 use. All code runs at the Exception level 2 privilege level. This file contains the initialization and handler code and is designed for both cores to use.
- `main.c`: This file contains C code, including code for message printing. This file is also designed for both cores and contains the `main()` function, which core 0 uses. Core 1 uses its own main function `core1_main()`.

Core 0 initializes the standard C library. The initialization step is only required once. Core 1 is then able to use the standard C library. For the cores to use the library functions like `printf()`, in a multi-threaded environment, the library must use the functions that `MP_Mutexes.c` implements.

- `guest_os1.s and guest_os2.s`: These files describe the target memory map to the linker, which enables the linker to place the data and code for the example at the correct addresses in memory.
- `scatter.scat`: This file describes the target memory map to the linker, which enables the linker to place the data and code for the example at the correct addresses in memory.
- `Armv8-R Virtualization Example 4 - Cortex-R52x1 FVP.launch` : This file is the Arm Development Studio IDE debug configuration for this example project.

In this example, C code is used for processes that Arm assembly code handled in previous examples. Depending on your preference, you can transfer Arm assembly code tasks to C, using this example as a starting point.

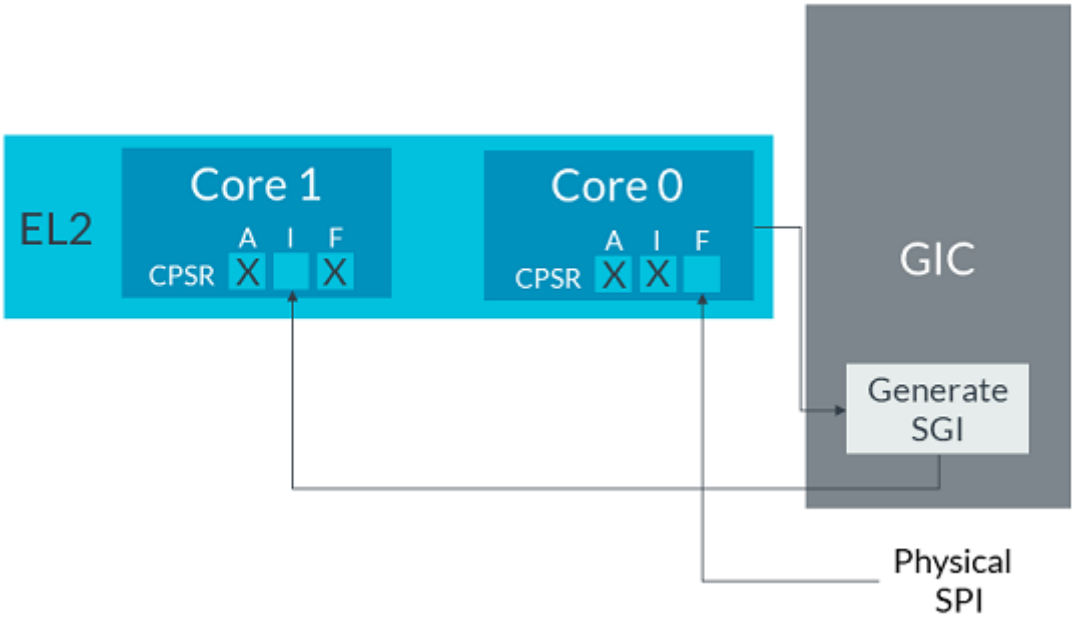
This example is simpler than the previous examples in this guide because it does not contain any hypervisors or OSs. The entire code is made up of one Exception level 2 process. However, the code is run on more than one core.

After initialization, this example can be broken down into the following steps:

1. An external timer generates an SPI every second, which is raised as an Exception level 2 FIQ on core 0.
2. The Exception level 2 FIQ handler running on core 0 reports that the SPI was received.
3. The Exception level FIQ handler running on core 0 generates an SGI targeted at core 1.
4. The SGI is raised as an Exception level 2 IRQ on core 1.
5. The Exception level 2 IRQ handler running on core 1 reports that the SGI was received.

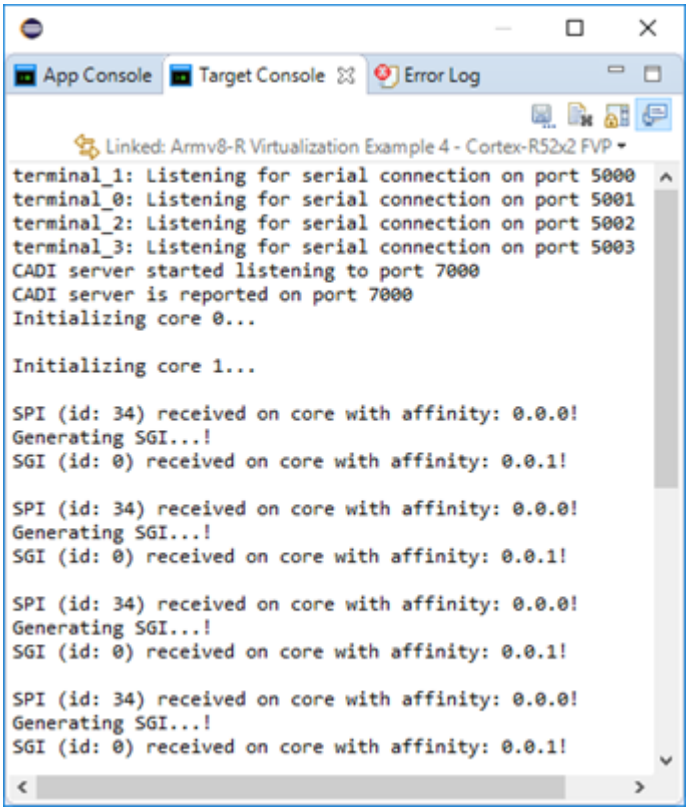
The following figure shows the scenario in this example:

Figure 8-1: SPI and SGI scenario



The following figure shows the Target Console output for both cores 0 and 1 in the Arm Development Studio Integrated Development Environment:

Figure 8-2: Target Console output



Use multiple cores

Unlike the previous examples, this example does not call `All_Cores_Except_0_To_Sleep()`. Instead, it runs a modified version of the code that allows both core 0 and core 1 to complete the boot process.

Each core has its own Redistributor. Because Redistributor registers are memory-mapped, you must know where they are located for each Redistributor. The code in `Basic_GIC_Setup()` that deals with the Redistributor is built to work with core 0 and core 1.

When the basic initialization has completed, the code branches to complete the initialization for either core 0 or core 1.

Core 0 initialization

To initialize core 0, follow these steps:

1. Set the HCR Trap FIQ Exceptions flag.
2. Disable the CPSR FIQ masking flag. This is required because an Exception level 2 process is running when FIQ exceptions are raised and at all other times in this example. In the other examples, disabling the masking flags at Exception level 2 was not required. The reason was because Exception level 1 processes were running when FIQ exceptions were raised. This meant that unless the exception was the result of a virtual FIQ, control jumped directly to the

Exception level 2 FIQ handler. In other words, in Exception level 1 processes, the CPSR FIQ masking flag is ignored for physical FIQs when the HCR Trap FIQ Exceptions flag is set.

The code branches to `__main()`, the C standard library is initialized, and the SPI external timer is set up.

Core 1 initialization

To initialize core 1, follow these steps:

1. Configure the core 1 Redistributor to receive physical SGIs as IRQs. In this example, an ID of 0 is chosen for the SGI, but any ID between 0 and 15 can be used. The process involves the GICR_IGROUPR0, GICR_IPRIORITYR, and GICR_ISENABLER0. For more information about configuring individual physical interrupts, see [Guest OS Switcher with Virtual Interrupts].
2. Set the HCR Trap IRQ Exceptions flag.
3. Disable the CPSR IRQ masking flag. This is required because an Exception level 2 process is running when IRQ exceptions are raised, and at other times in this example.

Use SPIs generated from the Arm Dual-Timer Module

The Cortex-R52 FVP provides an Arm Dual Timer Module (SP804). This module is an Advanced Microcontroller Bus Architecture (AMBA) compliant System on Chip (SoC) peripheral, and connects to the Advanced Peripheral Bus (APB). The module is physically independent from the Cortex-R52 processor, and contains two countdown timers that can both produce an SPI. Like all SPIs, these SPIs can be targeted at one specific core. In this example, you configure the GIC Distributor to send an FIQ to core 0 in response to SPIs from the Dual Timer module using C code.

In version 3.0 of the GIC, SPIs have an interrupt ID range of 32-1019. Specifically, the Cortex-R52 has a range of 32-991. However, it can still service over 900 interrupts from external peripherals. This example uses timer 1 from the Dual Timer Module. Timer 1 has an interrupt ID of 34. In GIC.h, a C struct is used to map all the GIC Distributor registers. Padding is used so that each register has the correct offset from the base address for the Distributor registers. You can compare the C struct with the GIC Distributor register table in theArm Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0.

Configuring the GIC Distributor for SPIs is like configuring a GIC Redistributor for SGIs or PPIs:

- The Interrupt Group Registers 1-30 (GICD_IGROUPR1-30) specify whether an SPI is received as an FIQ or IRQ.
- The Interrupt Priority Registers 8-247 (GICD_IPRIORITYR8-247) specify the priority of an SPI.
- The Interrupt Set-Enable Registers 1-30 (GICD_ISENABLER1-30) specify whether an SPI is enabled.

Use the Interrupt Routing Registers 32-991 (GICD_IROUTER32-991) to specify which core to send an SPI to. These registers are 64-bit and use the hierarchical addressing system described in [Generic Interrupt Controller]. The following code sets the SPI timer up in the Distributor:

```
void setupSPITimer(void)
{
    // Set up SPI by initializing the required GIC Distributor registers.
    volatile struct GIC_Distributor* dist =
```

```
            (volatile struct GIC_Distributor*)0xAF000000;

        // Specify interrupt 34 to be a Group 0 interrupt and therefore an
        // FIQ on a Cortex-R52 processor. Group 1 interrupts are IRQs.
        // To set the group to which different SPIs belong to, use the
        // Interrupt Group Registers (GICD_IGROUPR)
        dist->IGROUPR[1] &= 0xFFFFFFFB;         // Set bit 3 to 0. Interrupt 34
will be
                                                // in Group 0. All other interrupts
                                                // will retain their previous
setting.

        // Set the priority for interrupt 34 on core 0 using the
        // Interrupt Priority Registers (GICD_IPRIORITYR)
        dist->IPRIORITYR[34] = 0x1F;            // Set the priority of the
Interrupt 34
                                                // to 31.

        // Enable interrupt 34 on Core 0 using the
        // Interrupt Set-Enable Registers (GICD_ISENABLER).
        dist->ISENABLER[1] = 0x4;               // Set bit 3 to 1.

        // ** Route the SPI FIQ to core 0 using the
        // Interrupt Routing Registers (GICD_IROUTER).

        // Get affinity for core.
        // This will be core 0 and it is expected that all the affx
        // variables remain 0.
        // However, in a system with more than one processor getting these variables
        // would be an essential step.
        // The values for aff1 and aff2 might not be 0 and would have to
        // be obtained.
        int aff0 = 0;
        int aff1 = 0;
        int aff2 = 0;
        getAffinity(&aff0, &aff1, &aff2);

        // The Affinity 0 setting (bits 0-7) is set to the aff0 variable.

        aff1 <<= 8;  // Shift the Affinity 1 (bits 8-15) setting in the aff1
variable.
        aff2 <<= 16; // Shift the Affinity 2 (bits 16-23) setting in the aff2
variable.

        // Note that on the Cortex-R52:
        // The flag that routes an interrupt to all cores (bit 31) is
        // never used and is just set to 0 here.
        // The Affinity 3 setting (bits 32-39) is never used and
        // is just set to 0 here.
        unsigned long long iRouter = 0xFFFFFF;
        iRouter = iRouter && aff0 && aff1 && aff2;
        dist->IROUTER[34] = iRouter;
    }
```


Note

If you move this example to your own system from the Fixed Virtual Platform, check that the interrupt ID for any SoC timers on your system is 34.

The `getAffinity()` function is written in assembly and is called from both assembly and C code. The function retrieves Affinity Level 0, Affinity Level 1 and Affinity Level 2, in [Guest OS Switcher

with Virtual Interrupts] settings from the Multiprocessor Affinity Register (MPIDR), as shown in the following code:

```
.global getAffinity
.type getAffinity, "function"
getAffinity:
    // This function assumes the following:
    // r0, r1, r2: pointers to 32-bit int

    // Read MPIDR into r3.
    MRC p15, 0, r3, c0, c0, 5

    // Conform to AAPCS standard and preserve r4, r5, and r6.
    PUSH {r4-r6}

    // Load affinity level 0 (bits 0 - 7) into r4.
    MOV r4, r3
    AND r4, r4, 0xFF

    // Load affinity level 1 (bits 8 - 15)  into r5.
    MOV r5, r3
    AND r5, r5, 0xFF00
    LSR r5, 8

    // Load affinity level 2 (bits 16 - 23) into r6.
    MOV r6, r3
    AND r6, r6, 0xFF0000
    LSR r6, 16

    STR      r4, [r0]   // r4 into (int*) r0
    STR      r5, [r1]   // r5 into (int*) r1
    STR      r6, [r2]   // r6 into (int*) r2

    POP {r4-r6}

    BX      lr
```


Note

The 64-bit Interrupt Routing Registers (GICD_IROUTER32-991), Interrupt Controller Software Generated Interrupt Group 0 Register (ICC_SGI0R), and Interrupt Controller Software Generated Interrupt Group 1 Register (ICC_SGI1R) have an Affinity Level 3 setting that is not used. The Cortex-R52 processor does not support a Level 3 affinity. The 32-bit MPIDR does not have an Affinity Level 3 setting. In this example, all Affinity Level 3 settings are set to 0.

The Cortex-R52 has an internal Distributor, which means that it cannot route SPIs or SGIs to other processors. However, the MPIDR still reports its position in a cluster system. Affinity Level 1 and Affinity Level 2 settings that are read from MPIDR contain valid information, which must be written into Affinity Level 1 and Affinity Level 2 settings when routing SPIs and SGIs.

For more information, see the section Distributor Registers (GICD) in the Arm Cortex-R52 Processor Technical Reference Manual.

Set Arm Dual-Timer Module registers

For timer 1 from the Arm Dual-Timer Module to work, registers specific to it and unrelated to the Cortex-R52 must be set. A C struct in dual_timer.h is defined to map all the Arm Dual-

Timer Module registers. The struct is used in enableSPITimer() to initialize timer 1, and in SPI34FIQHandler() to clear the interrupt from the timer and begin the countdown again.

Use an interrupt service routine vector table

In this example, two C functions, or service routines, are called from the handlers in `start.s`:

- `SPI34FIQHandler()` is called from `EL2_FIQ_Handler()` in core 0
- `SGI0IRQHandler()` is called from `EL2_IRQ_Handler()` in core 1

In `start.s`, both C functions are referenced in the GIC_InterruptVectorTable rather than directly.

The GIC_InterruptVectorTable is an array of function pointers that are defined in GIC.c. In a more complicated system receiving multiple SPIs, PPIs, and SGIs, using a C array like this would enable you to write clean and efficient FIQ and IRQ handlers.

Configure an SGI

Sending an SGI is like targeting an SPI. To send an SGI, you can use the 64-bit ICC_SGI0R or ICC_SGI1R. An SGI that is sent using ICC_SGI0R generates an FIQ, and an SGI sent using ICC_SGI1R generates an IRQ. The following code from `EL2_FIQ_Handler()` shows how to send an SGI as an IRQ:

```
        // ** Send SGI
        // Use the Interrupt Controller Software Generated Interrupt Group 1
Register
        // (ICC_SGI1R), which is a 64-bit register.
        //
        // Get affinity for core.
        // This will be core 0 and it is expected that
        // values of 0 are returned for all affinity levels.
        // However, in a system with more than one processor this would be an
essential
        // step and the values for Aff1 and Aff2 would have to be obtained.

        // Note: getAffinity() is designed to be called by C code. Therefore,
variables
        // to receive the affinity levels must be assigned.
        // In other words things are set up as if it is being called from C.

        LDR    r3, =affinity_variables
        MOV    r0, r3            // Load address of affinity0.
        ADD    r3, r3, 4
        MOV    r1, r3            // Load address of affinity1.
        ADD    r3, r3, 4
        MOV    r2, r3            // Load address of affinity2.

        // Call getAffinity function
        BL getAffinity

        // affinity2, affinity1 and affinity0 should now contain the affinity for
        // this core. 0.0.0 is expected in this system.
        // affinity 0 can be discarded as the cores required (in this case only core
1)
        // are set using the target list.
        // affinity1 and affinity2 *must* be used in a multi-processor system
        // when routing SGIs.
        // Even though the Cortex-R52 cannot route SGIs outside of itself, these
        // two values will reflect its "position" in a multi-processor system.
        // They therefore must be respected or SGI routing will fail.

        // Set lower 31 bits for ICC_SGI1R in r0.
```

```
        MOV    r0, #0x0         // Initializes the Interrupt ID to use for this SGI
                                // (bits 24-27) because ID 0 has been chosen
                                // for this SGI.
        ORR    r0, r0, #0x2     // Set bit 1 of the target list (bits 0 to 4) to
                                // to direct this SGI to core 1 only.
        LDR    r2, =affinity_variables
        LDR    r3, [r2, #4]     // Load affinity1 into r3.
        ORR    r0, r0, r3, LSL #16  // Set bits 16-23 to affinity1.

        // Set upper 31 bits for ICC_SGI1R in r1.
        MOV    r1, #0x0         // Sets bit 40 to 0 - This turns off a flag
                                // that determines whether to send the SGI
                                // to every core (and override the target list).
                                // The Affinity 3 setting (bits 48-55) is also set
 to 0
                                // but the setting is not relevant to the Cortex-
R52.
        LDR    r3, [r2, #8]     // Load affinity2 into r3.
        ORR    r1, r1, r3       // Set bits 32-39 to affinity2.

        MCRR p15, 0, r0, r1, c12 // Write r0 and r1 to ICC_SGI1R.
```

`getAffinity()` is used to get the affinity settings for the core running the code. You must have the correct values for affinity level 1 and affinity level 2 to correctly route the interrupts to the cores. However, setting affinity level 0 is different when using the ICC_SGI1R, as opposed to the GICD_IROUTERs, as explained in Use SPIs generated from the Arm Dual-Timer Module.

Although the Cortex-R52 is limited to four cores, the GIC architecture allows for an SGI to be routed to up to 16 cores. If you are using another processor based on the Armv8-R architecture that also implements the GIC, you can route an SGI to up to 16 cores if necessary.

For more information, see the section CPU Interface Registers in the Arm Cortex-R52 Processor Technical Reference Manual.

9 Related information

Here are some resources that are related to the material in this guide:

- Arm Community
- Arm Cortex-R52 Processor Technical Reference Manual
- Arm Cortex-R Series Programmer's Guide

Generic Interrupt Controller:

- Arm Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0
- GICv3 and GICv4 Software Overview

SPIs and SGIs example:

- Arm Dual-Timer Module (SP804) Technical Reference Manual

10 Next steps

Based on what you learned in this guide, you can try the following tasks:

- Although there was no virtualization in this example, SPIs and SGIs can both be virtualized. An SPI or an SGI can be received by a core running a hypervisor and multiple guest OSs. A virtual interrupt can then be generated for the SPI or SGI and forwarded as required to the guest OSs. For example, you can set up a virtual interrupt with an ID of 34 and use this as a virtual Dual-Timer Module interrupt.
- Extend this example to create virtual guest OSs for core 0 and core 1 and then forward either virtual SPIs or SGIs to them
- Adjust this example so that core 0 receives the SGI as well. You could also use the FVP_BaseR_Cortex-R52x4 Fixed Virtual Platform and experiment with sending the SGIs to cores 2 and 3.
- Route an SGI to every core. To do this, set the Target All Cores flag of the ICC_SGI0R or ICC_SGI1R. When this is done, all affinity settings are ignored.
- Create C structures for the Redistributor registers. For example, the code that allows core 1 to receive an SGI with an interrupt ID of 0 could be moved to core1_main().
- Port a hypervisor or similar technology with the Protected Memory System Architecture (PMSA)

To learn more about the Armv8-R architecture, read our series of Cortex-R guides.