FTF 2016
TECHNOLOGY FORUM

HANDLING EXCEPTIONS IN MULTICORE ARM®v8 QorIQ PROCESSORS

FTF-DES-N1849

CHUCK CORLEY
SENIOR PRINCIPAL ENGINEER
FTF-DES-N1849
MAY 16, 2016

AGENDA

- References
- Overview
- Security Model and Exception Levels in ARMv8
  - Concept of banked registers by EL
- Interrupt Types
- GICv3 (Generic Interrupt Controller) Programmers’ model
  - Configuring the Distributor
  - Configuring the Re-Distributor
  - Configuring the CPU Interface
- Configuring the Core
- Configuring specifically the PPI (Private Peripheral) and SGI (Software Generated) Interrupts
- The Generic Timer
  - Enabling time counters
  - Enabling timer signals (interrupts)
- Example Code

Caveat – Detail Which Can Be Ignored for Now

Unless indicated otherwise, this [presentation ~~manual~~] describes the GICv3 architecture in a system with affinity routing, System register access, and two Security states, enabled. This means that:

- GICD_CTLR.ARE_NS == 1.
- GICD_CTLR.ARE_S == 1.
- GICD_CTLR.DS == 0.

We will set these.

For operation in AArch64 state:

- ICC_SRE_EL1.SRE == 1, for both the Secure and the Non-secure copy of this register.
- ICC_SRE_EL2.SRE == 1.
- ICC_SRE_EL3.SRE == 1.

NOTE: These are not the reset values.

I.e., we will have to set these.

REFERENCES

Important References

Generally it requires consulting several ARM documents to figure out how to do anything. Three in particular are important here:

- GICv3 Software Overview (DAI 0492A) – really useful recent application note on the topic
- ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0 (ARM IHI 0069A) – necessary for GIC register definitions and operation
- ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile (ARM DDI 0487A.g) – necessary for CPU register definitions and operation – Appendix J11.2 Alphabetical index of AArch64 registers and system instructions is invaluable

OVERVIEW

Overview

Frankly, I found implementing exceptions in ARMv8 using the GIC-500 challenging. And explaining it is even more daunting. Let me start at a high level.

- ARMv8 has two asynchronous interrupt input “pins” – IRQ and FIQ. Numerous interrupt sources can be or’ed onto either of those pins at the direction of the Generic Interrupt Controller (GIC). The two interrupt sources are more aptly now called Group0 and Group1 as the IRQ and FIQ differentiation has been re-purposed.
- To accommodate virtualization, ARMv8 has four levels of exception EL0 – EL3 for user apps, OSes, Hypervisor and Secure Monitor.
- User apps and OSes can be secure or non-secure; Hypervisor can only be non-secure; the Secure Monitor can only be secure. Hence, we will speak of being secure or non-secure and of being at an exception level of EL0, EL1, EL2, or EL3
- The GIC has to be programmatically configured for the ARM core to receive an interrupt.

Overview (cont'd)

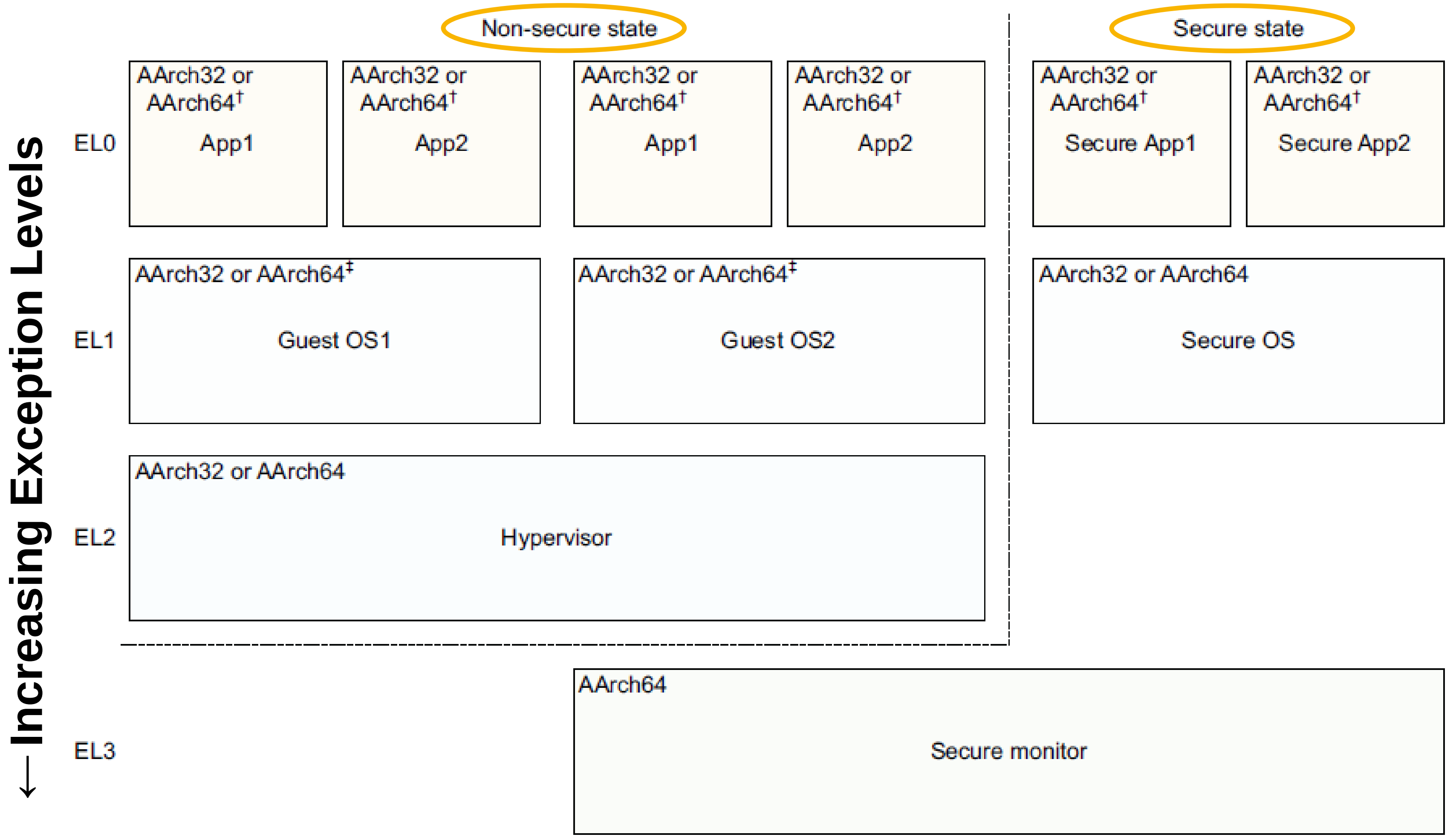
Once the GIC is programmed, exception handling is similar to other architectures – Power Architecture for example.

- Enabled interrupts of high enough priority and security cause execution to begin at one of several entry vectors depending on entry Exception Level (EL), security state, and type of exception (synchronous, IRQ, FIQ, or system error)
- The hardware saves some state (like return address and machine status) automatically on entry to the exception.
- The exception handler routine(s) must:
  - Save additional state (like registers to be overwritten)
  - Identify the type of interrupt
  - Handle the interrupt
  - Signal end-of-interrupt or clear it to avoid repeating the ISR
  - Restore saved registers
  - Return from interrupt instruction (eret) will restore machine state and branch to return address.

This is not so hard and will not be addressed in this presentation

SECURITY MODEL AND EXCEPTION LEVELS IN ARMV8

ARMv8 Security Model and Exception Levels



† AArch64 permitted only if EL1 is using AArch64

‡ AArch64 permitted only if EL2 is using AArch64

Registers May Be Banked by Exception Level

	EL0	EL1 (S)	EL1 (NS)	EL2	EL3	
SP = Stack Ptr	**SP_EL0**	**SP_EL1**		**SP_EL2**	**SP_EL3**	SPSel=?
ELR = Exception Link Register		**ELR_EL1**		**ELR_EL2**	**ELR_EL3**	(PC)
Saved/Current Process Status Register		**SPSR_EL1**		**SPSR_EL2**	**SPSR_EL3**	(CPSR)
System Interrupt Enable Register		**ICC_SRE_EL 1(S)**	**ICC_SRE_EL 1(NS)**	**ICC_SRE_EL 2**	**ICC_SRE_EL 3**	See later slide
EL1 Physical Timer Compare Value register		**CNTP_CVAL _EL0(S)**	**CNTP_CVAL _EL0(NS)**			See later slide
EL1 Physical Timer Value register		**CNTP_TVAL_ EL0(S)**	**CNTP_TVAL_ EL0(NS)**			See later slide
EL1 Physical Timer Control register		**CNTP_TVAL_ EL0(S)**	**CNTP_TVAL_ EL0(NS)**			See later slide

Acknowledgement:
Richard Grisenthwaite, ARMv8Technology Preview, 2011 ARM TechCon

INTERRUPT TYPES

Interrupt Types

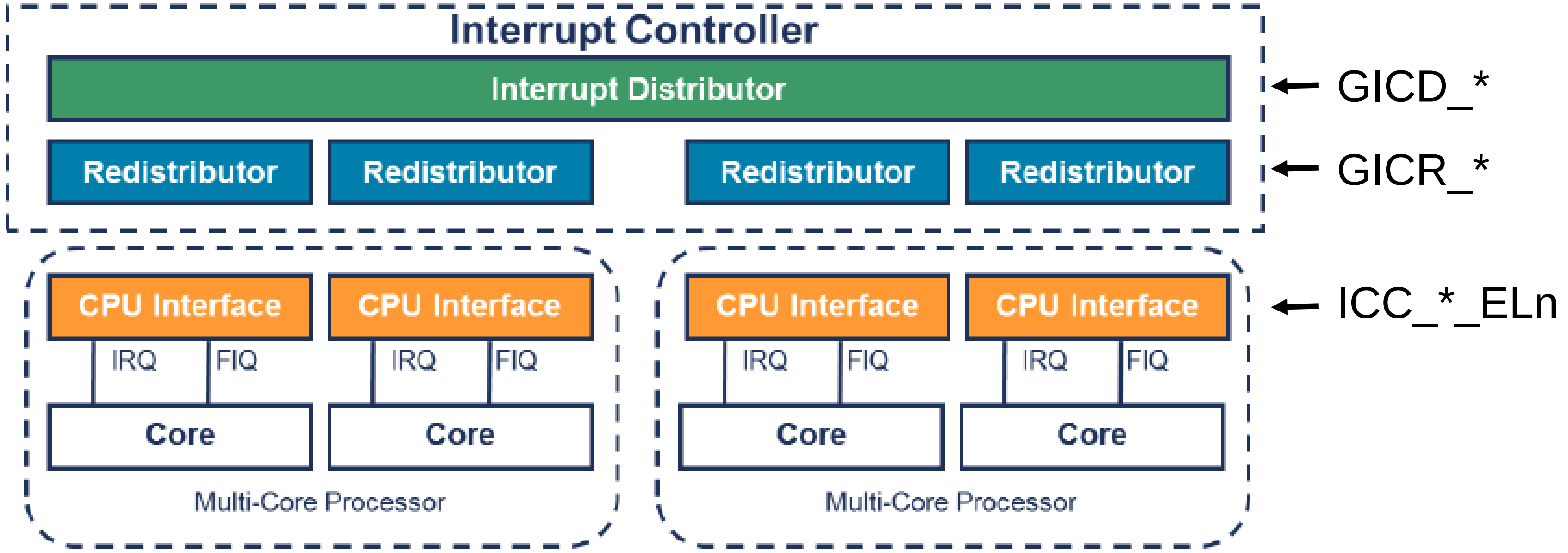
- SGI (Software Generated Interrupt) INTID 0 – 15 per core
  - SGIs are typically used for inter-processor communication, and are generated by a write to an SGI register in the GIC
- PPI (Private Peripheral Interrupt) INTID 16 – 31 per core
  - This is peripheral interrupt that targets a single, specific core.
  - An example of a PPI
- SPI (Shared Periphe
  - This is a global peripheral interrupt that can be routed to a specified core, or to one of a group of cores.
- LPI (Locality-specific Peripheral Interrupt) INTID 8192+
  - LPIs are new in GICv3, and they are different to the other types of interrupt in a number of ways. In particular, LPIs are always message-based interrupts, and their configuration is held in tables in memory rather than registers.

Not addressed in this presentation
See GICv3 Software Overview

GICV3 PROGRAMMERS' MODEL

GICv3 Programmers' Model

- The register interface of a GICv3 interrupt controller is split into three groups:
- Distributor interface (GICD_* registers).
- Redistributor interface. (GICR_* registers).
- CPU interface (ICC_*_ELn registers).
  - Software must enable the System register interface before using these registers. This is controlled by the SRE bit in the ICC_SRE_ELn registers, where "n" specifies the Exception level (EL1-EL3).

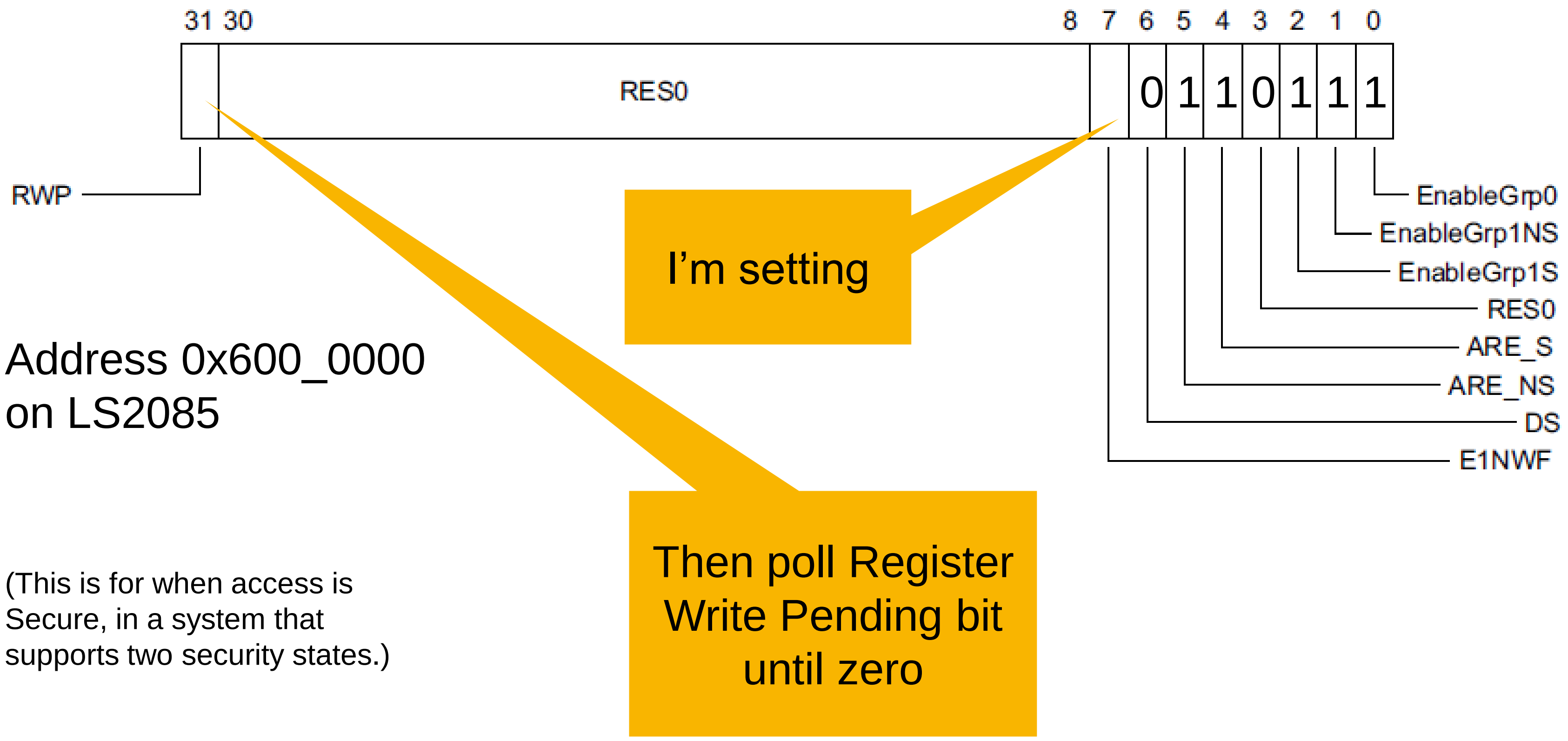


QorIQ LS2085A QDS CodeWarrior Memory Map

Address	Use
0x01e3_18a0	Power Management unit in CCSR space
0x02ed_0000	Global Generic Reference Timer in CCSR
0x0600_0000	GICD regs in CCSR space
0x0610_0000	GICR CPU 0 RD_base
0x0611_0000	GICR CPU 0 SGI_base
0x0620_0000	GICR CPU 1 RD_base
0x0621_0000	GICR CPU 1 SGI_base
0x8000_0000+	Exception Vectors (set in linker control file)

Configuring the GIC

The Distributor control register (GICD_CTLR) must be configured to enable the interrupt Groups and to set the routing mode.

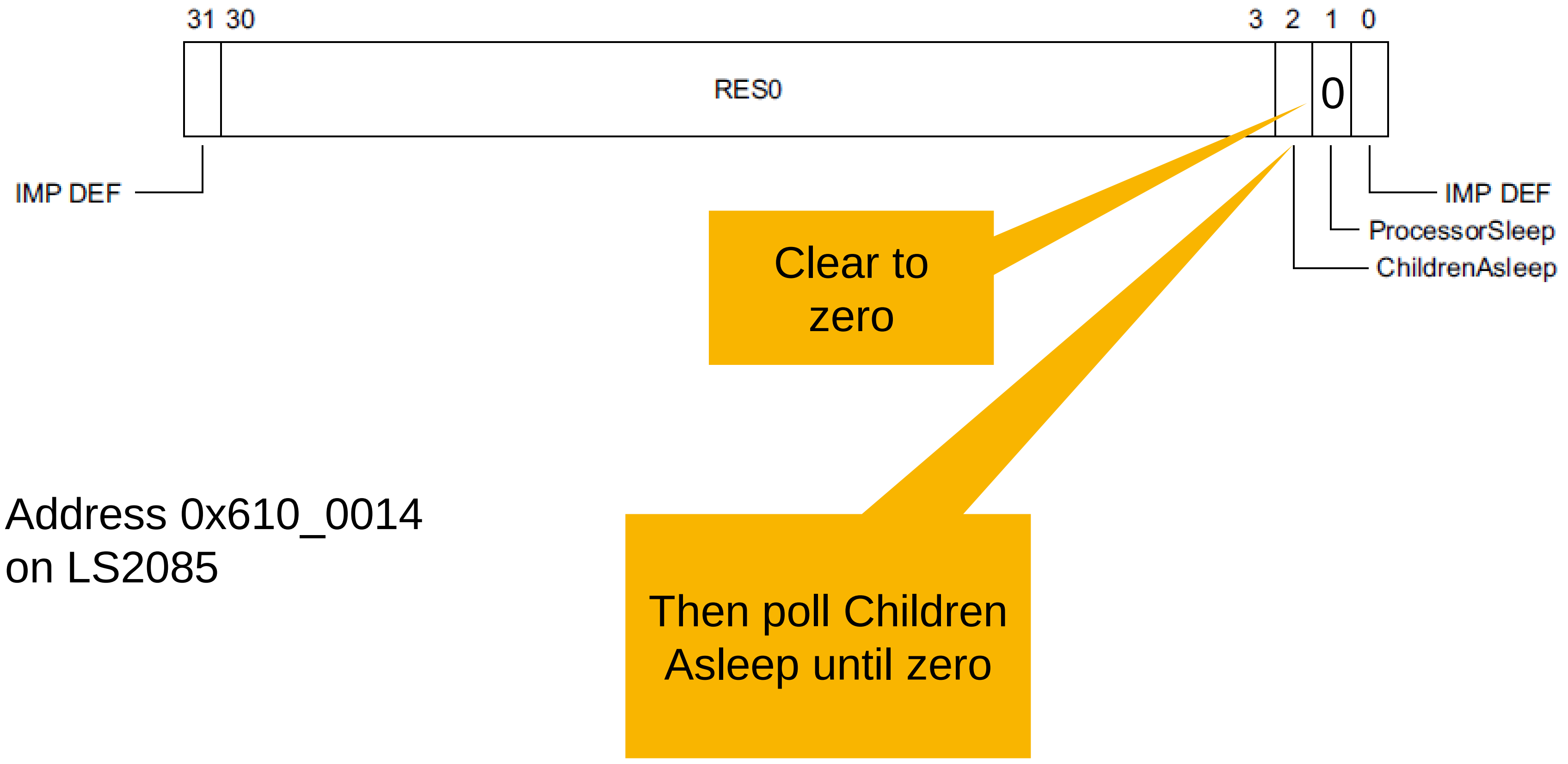


Address 0x600_0000 on LS2085

(This is for when access is Secure, in a system that supports two security states.)

Redistributor Configuration

Have to wake-up the connected core via GICR_WAKER register.



Address 0x610_0014
on LS2085

GIC System Register Access Can Be via CPU Registers

The GIC System register interface is managed by Exception level, using the following AArch64 System registers:

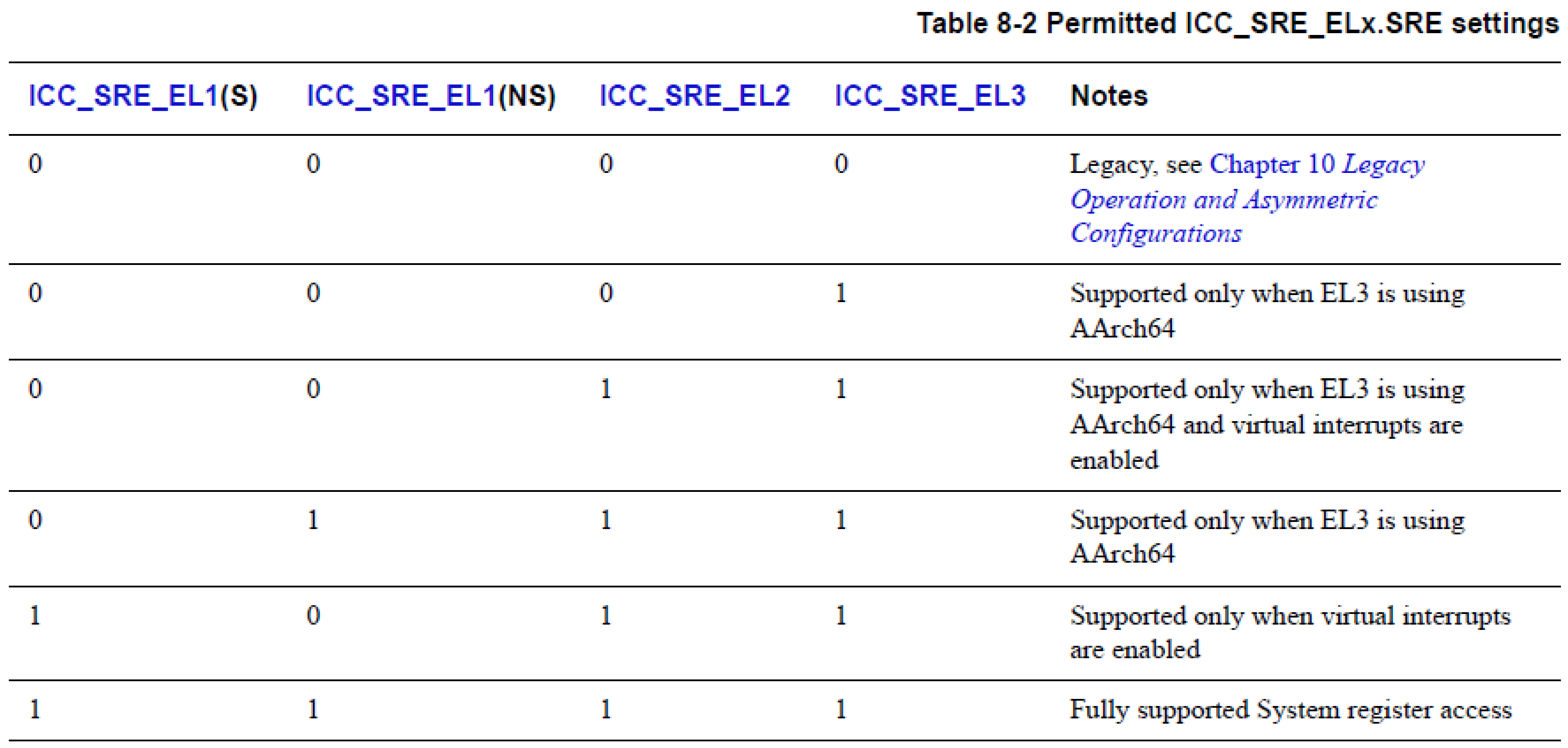
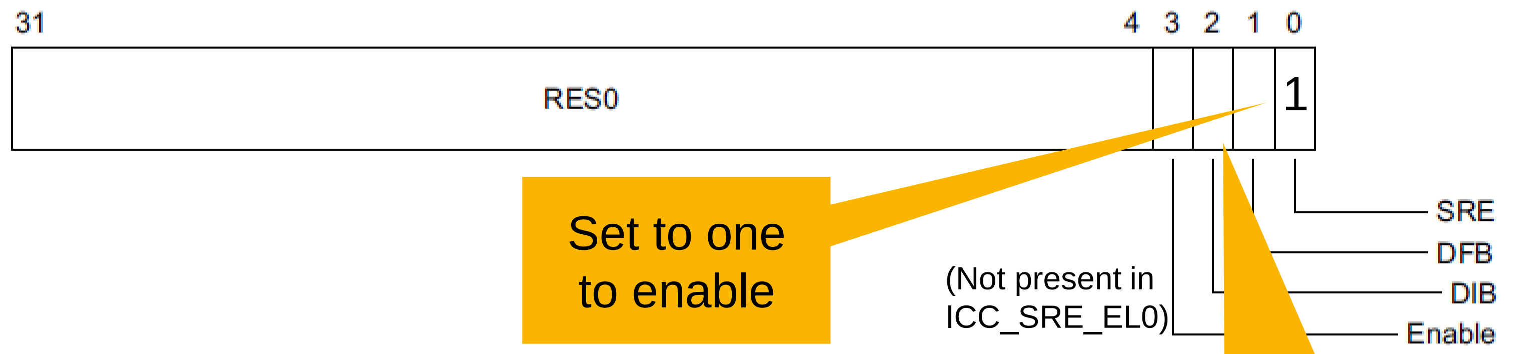


Table 8-2 Permitted ICC_SRE_ELx.SRE settings

ICC_SRE_EL1(S)	ICC_SRE_EL1(NS)	ICC_SRE_EL2	ICC_SRE_EL3	Notes
0	0	0	0	Legacy, see Chapter 10 *Legacy Operation and Asymmetric Configurations*
0	0	0	1	Supported only when EL3 is using AArch64
0	0	1	1	Supported only when EL3 is using AArch64 and virtual interrupts are enabled
0	1	1	1	Supported only when EL3 is using AArch64
1	0	1	1	Supported only when virtual interrupts are enabled
1	1	1	1	Fully supported System register access

CPU Interface Configuration – Register Access

Software must first enable access to the CPU interface registers, by setting the SRE bit in the ICC_SRE_ELn registers.



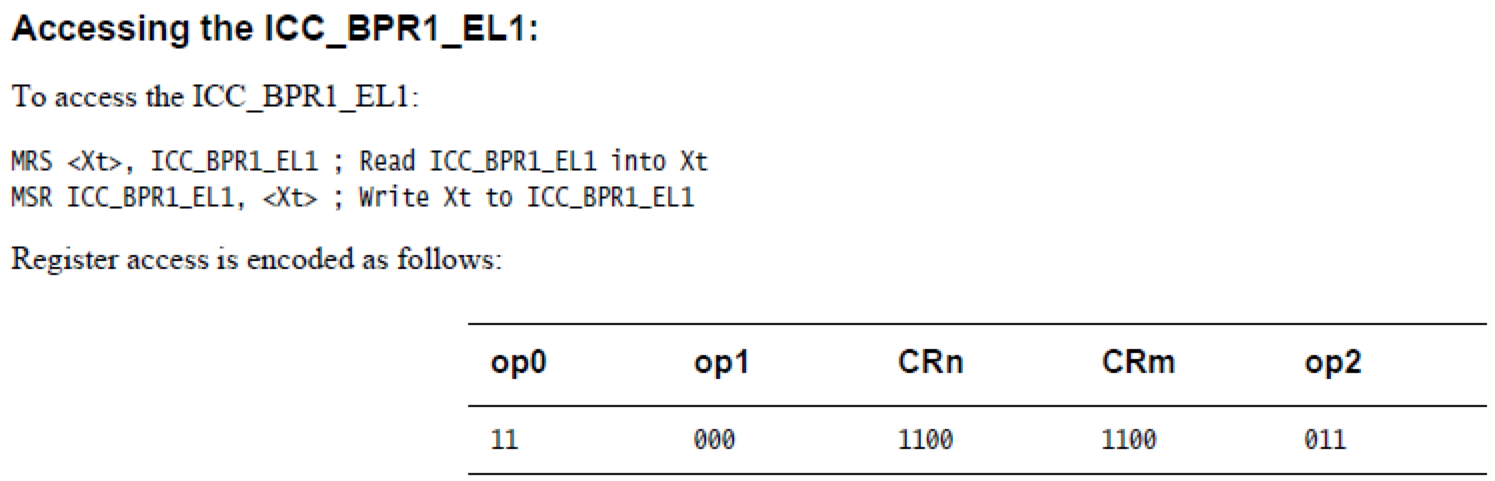
Set to one to enable

```
/* Let SCR_EL3.NS = 0 then: */
asm  (msr icc_sre_el3, %0” : : “r” (0x1));
asm  (msr icc_sre_el1, %0” : : “r” (0x1));
/* then let SCR_EL3.NS = 1 and:
asm  (msr icc_sre_el2, %0” : : “r” (0x1));
asm  (msr icc_sre_el1, %0” : : “r” (0x1));
```

Less sure about these but I think ARM is recommending DIB = 0 at EL3

CPU Register Access – Example

When the compiler fails to recognize the register name, the information below allows encoding as “implementation specific” register op0_op1_CRn_CRm_op2 or S3_0_C12_C12_3



Accessing the ICC_BPR1_EL1:

To access the ICC_BPR1_EL1:

```
MRS <Xt>, ICC_BPR1_EL1 ; Read ICC_BPR1_EL1 into Xt
MSR ICC_BPR1_EL1, <Xt> ; Write Xt to ICC_BPR1_EL1
```

Register access is encoded as follows:

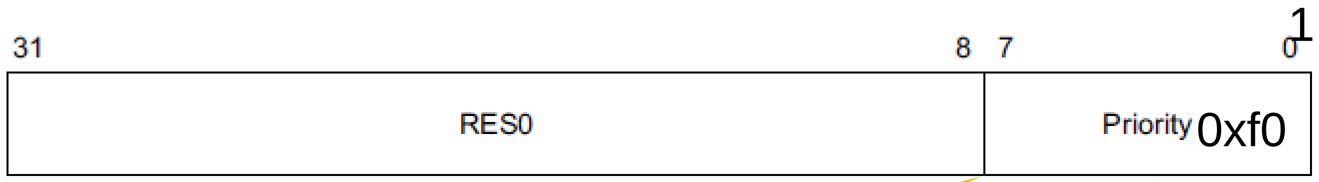
op0	op1	CRn	CRm	op2
11	000	1100	1100	011

in my experience, some are recognized as icc_*_el* and some are not. Perhaps there is a *.h file somewhere that defines them that I am not including.

CPU I/F Config – Priority Mask Register

The Priority Mask sets the priority an interrupt must have in order to be forwarded to the core.

The highest priority is 0x00; lowest is 0xf8 (GIC500 only implements 5 bits). i.e. interrupt has to be numerically lower than the value in PMR.



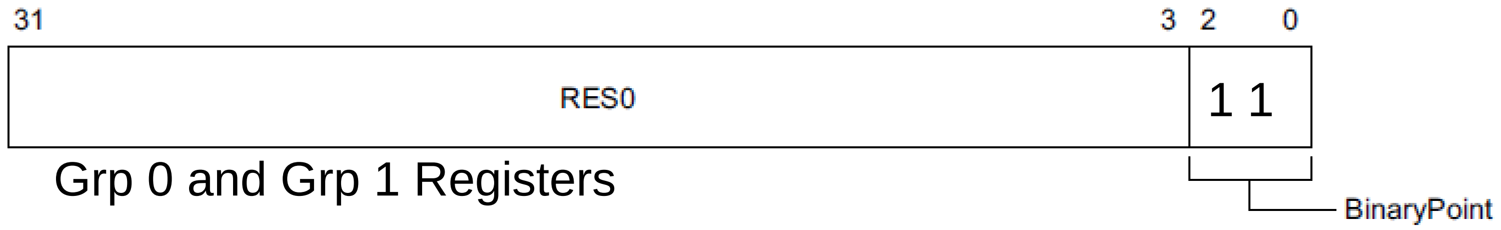
What priority interrupt will the CPU accept?

Binary pointer (on next slide) determines what level will pre-empt current interrupt

```
#define icc_pmr_el1 S3_0_C4_C6_0
asm  (msr icc_pmr_el1, %0” : : “r” (0xf0));
```

CPU I/F Config – Binary Point Register

The Binary Point register is used for priority grouping and preemption. The value controls how the 8-bit interrupt priority field is split into a group priority field, that determines interrupt preemption, and a subpriority field.



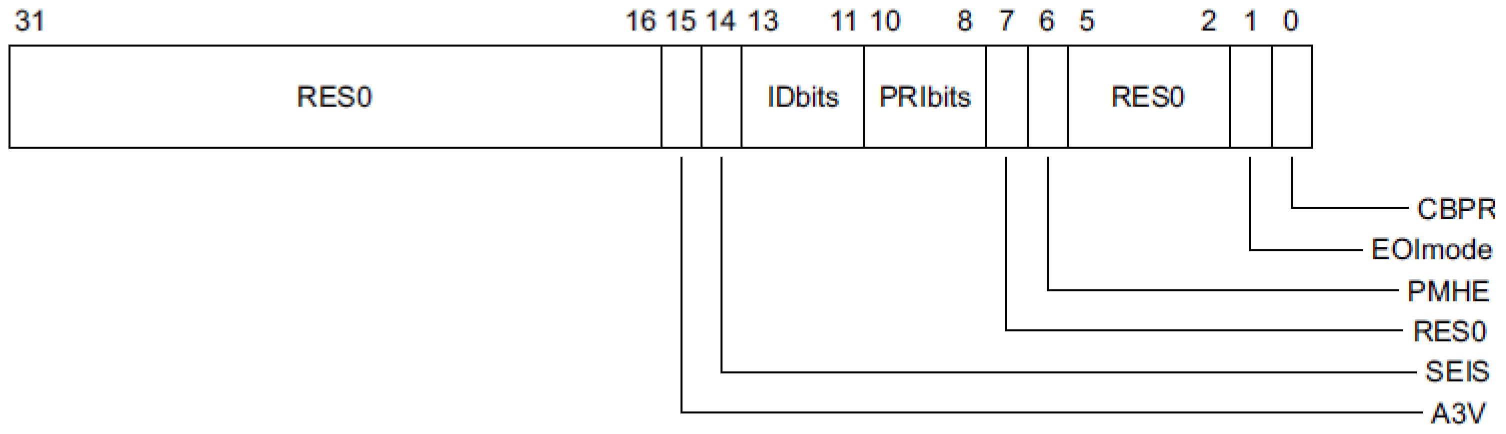
Grp 0 and Grp 1 Registers

```
#define icc_bpr0_el1
S3_0_C12_C8_3
asm  (msr icc_bpr0_el1, %0" :
: "r" (0x3));
#define icc_bpr1_el1
S3_0_C12_C12_3
asm  (msr icc_bpr1_el1, %0" :
: "r" (0x3));
```

Binary point value	Group priority field	Subpriority field	Field with binary point
0	[7:1]	[0]	ggggggg.s
1	[7:2]	[1:0]	gggggg.ss
2	[7:3]	[2:0]	ggggg.sss
3	[7:4]	[3:0]	gggg.ssss
4	[7:5]	[4:0]	ggg.sssss
5	[7:6]	[5:0]	gg.ssssss
6	[7]	[6:0]	g.sssssss
7	No preemption	[7:0]	.ssssssss

CPU I/F Config – Set EOI Mode

The ICC_CTLR_ELn registers control aspects of the behavior of the GIC CPU interface and provides information about the features implemented.



In my case, ICC_CTLR_EL1.EOImode = 0 and ICC_CTLR_EL3.EOImode = 0 says a write to an End of Interrupt register also deactivates the interrupt.

```
msr icc_ctlr_el3, %0” : : “r” (0));
msr icc_ctlr_el1, %0” : : “r” (0));
```

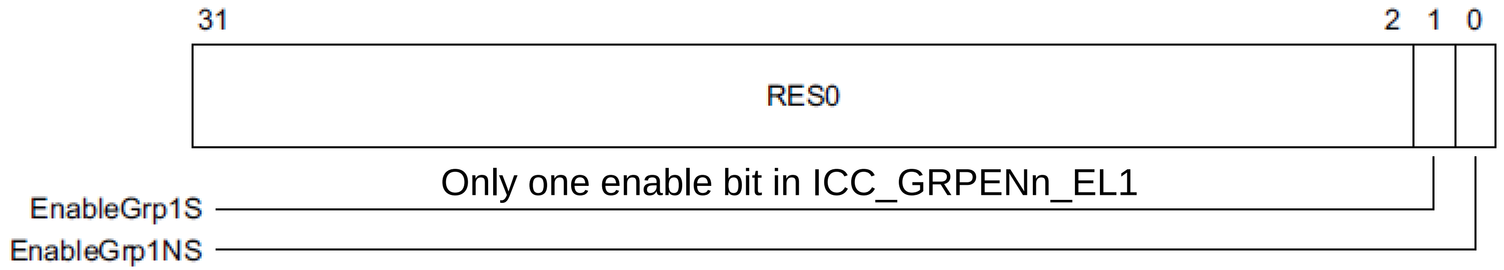
CPU I/F Config – Enable Signaling of Each Group

The signaling of each interrupt group must be enabled before interrupts of that group will be forwarded by the CPU interface to the core.

- To enable signaling software must write to ICC_IGRPEN1_EL1 register for Group 1 interrupts and ICC_IGRPEN0_EL1 registers for Group 0 interrupts.

ICC_IGRPEN1_EL1 is banked by Security state.

- This means that ICC_GRPEN1_EL1 controls Group 1 for the current Security state. At EL3, software can access both Secure Group 1 interrupt and Non-secure Group 1 interrupt enables using ICC_IGRPEN1_EL3.



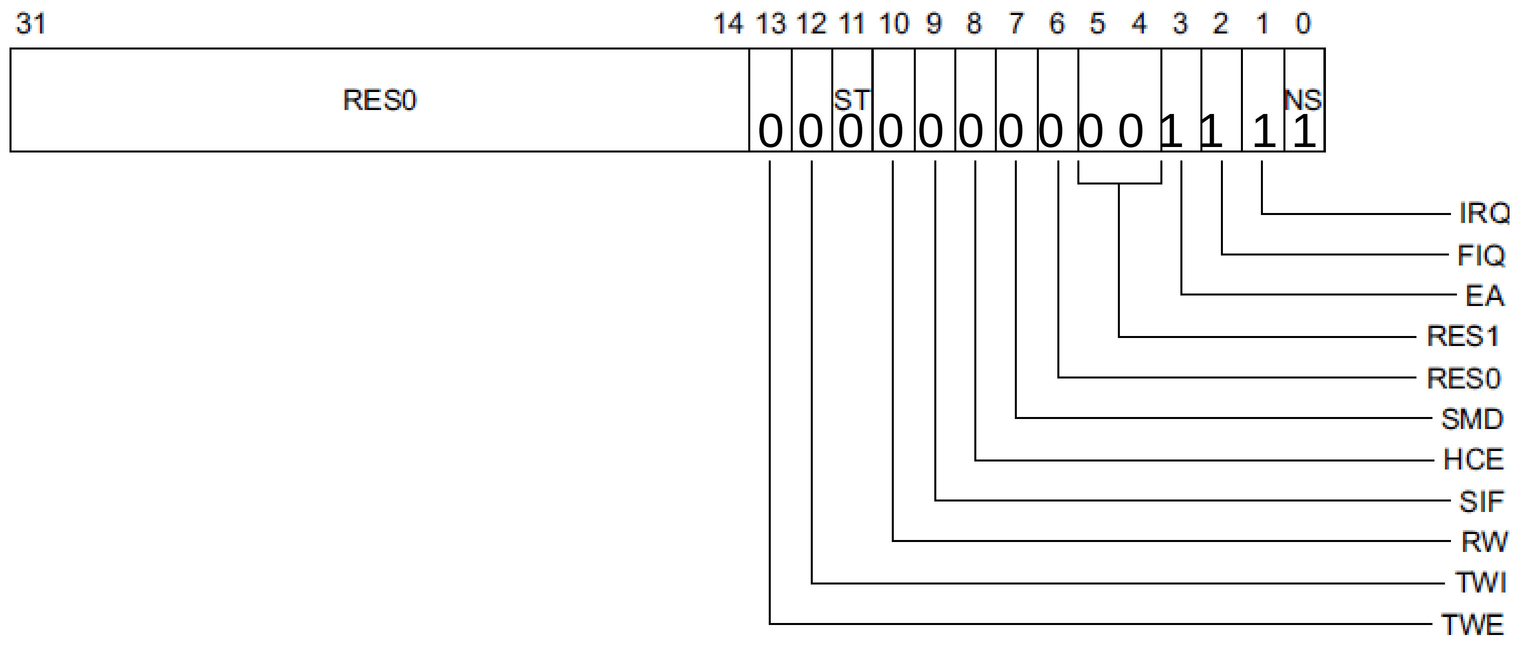
```
asm  ( msr icc_grpen0_el1, %0” : : “r” (1));     // S3_0_C12_C12_6
asm  ( msr icc_grpen1_el1, %0” : : “r” (1));     // S3_0_C12_C12_7
asm  ( msr icc_grpen1_el3, %0” : : “r” (3));     // S3_6_C12_C12_6
```

CONFIGURING THE CORE

Core Configuration

Some configuration of the core (or cores) is also required to allow them to receive and handle interrupts. In a CodeWarrior project core configuration is largely performed in the start.S file.

- Routing Controls – start.S sets SCR_EL3 as shown below



Core Configuration (cont'd)

- VectorTable – start.S sets VBAR_EL3 register to point to the vector tables LS3_vectors in exception.S as positioned in memory by the linker control file aarch64elf.x (0x8000_0000).
- Interrupt masks - The core also has exception mask bits in PSTATE. When these bits are set, interrupts are masked. These bits are set at reset. Our code must enable by clearing.
  - **The exception mask bits**
    - **D** Debug exception mask bit. See
    - **A**, **I**, **F** Asynchronous exception mask bits:
      - A SError interrupt mask bit.
      - I IRQ interrupt mask bit.
      - F FIQ interrupt mask bit.
- An exception routed to a lower EL that the current one is always masked
- An exception routed to a higher EL is never masked
- CAUTION: since SCR_EL3 and HCR_EL2 are UNKNOWN at reset, without initialization these "routing bits" may prevent an interrupt from being taken by the core.

```
asm  (msr DAIFclr, #0xf);    /* This handy register definition allows clearing w/o
RMW */
```

CONFIGURING SPECIFICALLY THE PPI AND SGI INTERRUPTS

PPI and SGI configuration

- PPIs and SGIs are configured through the individual Redistributors, using the GICR_* registers.
- The base address of this frame is referred to as RD_base. In addition, each Redistributor defines the following additional frames: SGI_base
- In GICv3, a 64KB frame for the control and generation of SGIs. The base address of this page is referred to as SGI_base. The frame for each Redistributor must be contiguous and must be ordered as follows:
  - RD_base
  - SGI_base = RD_base + 0x1000 offset

- Important: All GICR_I*R registers are relative to SGI_base!

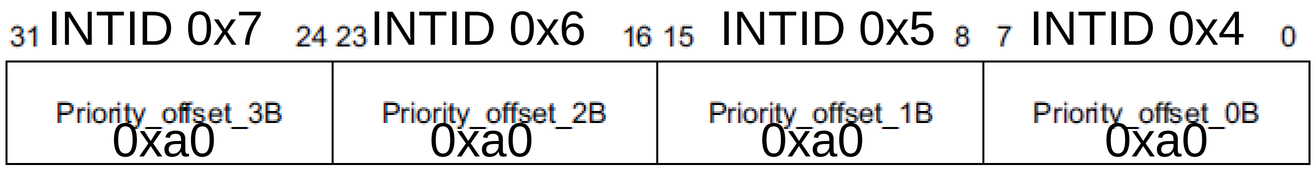
PPI and SGI Configuration – Priority

Each INTID has an associated priority, represented as an 8-bit unsigned value. 0x00 is the highest possible priority, and 0xF8 is the lowest possible priority.

GICD_IPRIORITYR<n>, Interrupt Priority Registers, n = 0 – 254

- GICR_IPRIORITYR<n> is used instead of GICD_IPRIORITYR<n> where n = 0 to 7 (that is, for SGIs and PPIs).

For example, in GICR_IPRIORITYR1:



Address 0x611_0404
on LS2085

```
for (i=0; i<8; i++) {
        addr = GICR_IPRIORITYRn + i * 4;
        value = 0xa0a0a0a0;
        *((unsigned int *) addr) = value;
}
```

Interrupts in Security Model

Each INTID must be assigned a security setting and a group.

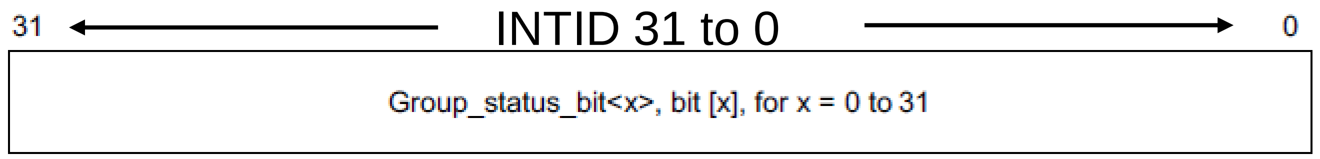
Interrupt Type	Example use
Secure Group 0	Interrupts for EL3 (Secure Firmware)
Secure Group 1	Interrupts for Secure EL1 (Trusted OS)
Non-secure Group 1	Interrupts for the Non-secure state (OS and/or Hypervisor)

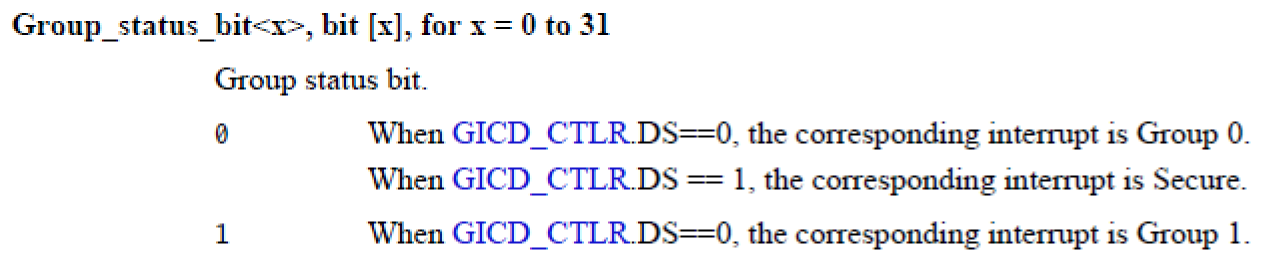
PPI and SGI Configuration – Group

An interrupt can be configured to belong to one of the three distinct interrupt groups. These interrupt groups are Group 0, Secure Group 1 and Non-secure Group 1.

GICD_IGROUPR<n>, Interrupt Group Registers, n = 0 – 31

When affinity routing is enabled for Secure state, GICD_IGROUPR0 is RES0 and equivalent functionality is proved by GICR_IGROUPR0. In other words, GICR_IGROUP0 is used instead of GICD_IGROUPR<n> for SGIs and PPIs (INTID 0 – 31).





Group_status_bit<x>, bit [x], for x = 0 to 31

Group status bit.

0	When GICD_CTLR.DS==0, the corresponding interrupt is Group 0. When GICD_CTLR.DS == 1, the corresponding interrupt is Secure.
1	When GICD_CTLR.DS==0, the corresponding interrupt is Group 1.

Address 0x611_0080
on LS2085

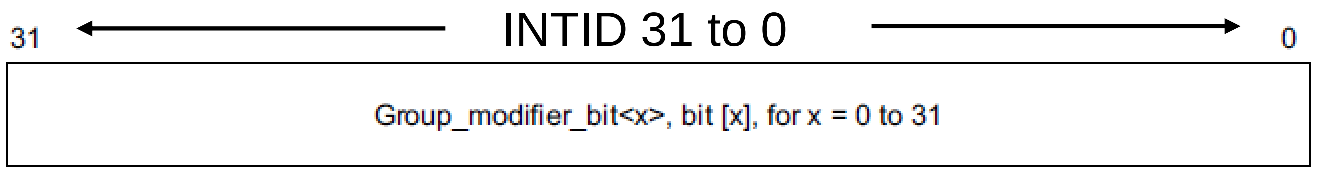
***((unsigned int *) GICR_IGROUPR0) = 0;**

PPI and SGI Configuration – Group (cont'd)

An interrupt can be configured to belong to one of the three distinct interrupt groups. These interrupt groups are Group 0, Secure Group 1 and Non-secure Group 1.

GICD_IGRPMODR<n>, Interrupt Group Modifier Registers, n = 0 – 31

- When affinity routing is enabled for Secure state, GICD_IGRPMODR0 is RES0 and equivalent functionality is provided by GICR_IGRPMODR0.. In other words, GICR_IGROUPMOD0 is used instead of GICD_IGROUPMODR<n> for SGIs and PPIs (INTID 0 – 31).



Group modifier bit	Group status bit	Definition	Short name
0	0	Secure Group 0	G0S
0	1	Non-secure Group 1	G1NS
1	0	Secure Group 1	G1S
1	1	Reserved, treated as Non-secure Group 1	-

Address 0x611_0d00
on LS2085

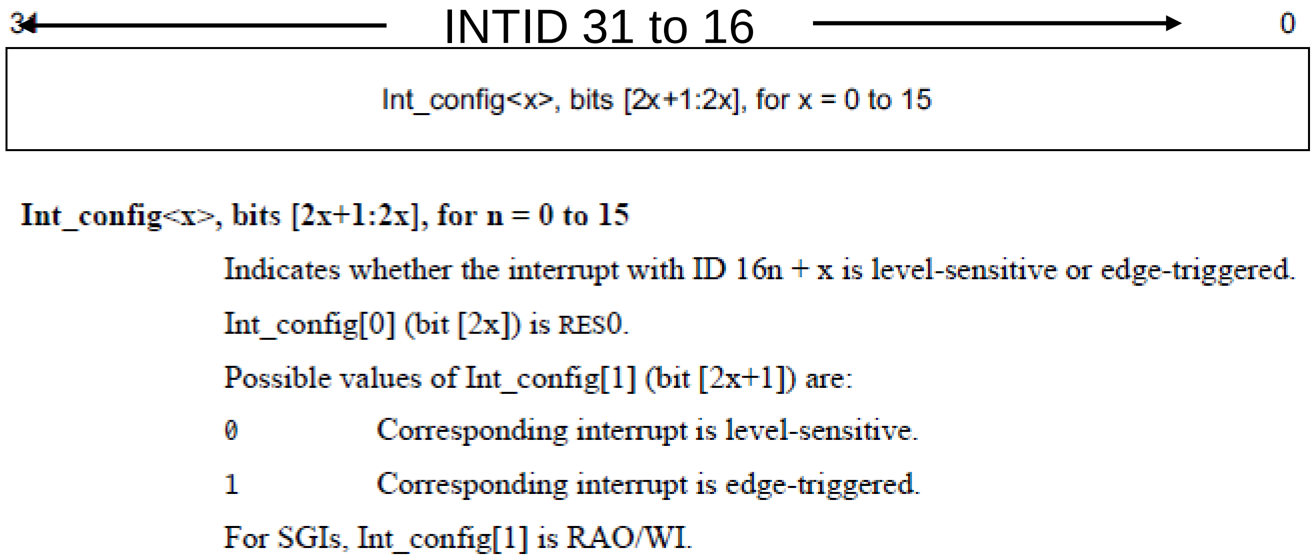
***((unsigned int *) GICR_IGROUPMODR0) = 0;**

PPI and SGI Configuration – Edge-triggered/level-sensitive

If the interrupt is sent as a physical signal, it must be configured to be either edge-triggered or level-sensitive.

SGIs are always treated as edge-triggered, and therefore GICR_ICFGR0 behaves as RAO/WI for these interrupts.

GICD_ICFGR<n>, Interrupt Configuration Registers, n = 0 - 63



Int_config<x>, bits [2x+1:2x], for n = 0 to 15

Indicates whether the interrupt with ID 16n + x is level-sensitive or edge-triggered.

Int_config[0] (bit [2x]) is RES0.

Possible values of Int_config[1] (bit [2x+1]) are:

0	Corresponding interrupt is level-sensitive.
1	Corresponding interrupt is edge-triggered.

For SGIs, Int_config[1] is RAO/WI.

Address 0x611_0c04
on LS2085

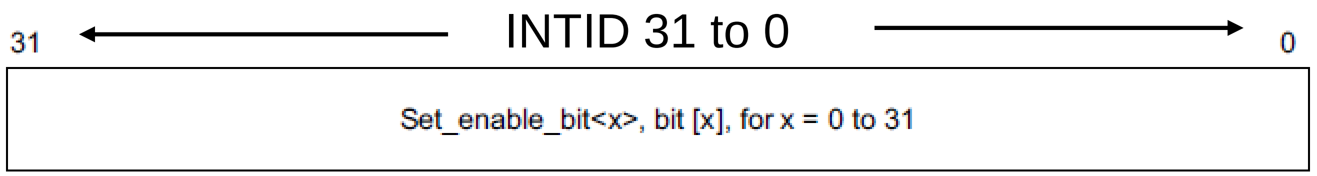
***((unsigned int *) GICR_ICFGR1) = 0;**

PPI and SGI configuration – Enable

Each INTID has an enable bit. Set-enable registers and Clear-enable registers remove the requirement to perform read-modify-write routines.

ARM recommends that the settings outlined in this section are configured before enabling the INTID.

GICD_ISENABLER<n>, Interrupt Set-Enable Registers, n = 0 - 31



Address 0x611_0100
on LS2085

***((unsigned int *) GICR_ISENABLE0) = 0xffffffff;**

THE GENERIC TIMER

Generic Timer

The Generic Timer:

- Provides a system counter, that measures the passing of time in real-time.
- Supports virtual counters that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Timers, that can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

Enabling Timer Interrupt

The CNTFRQ_EL0 register is intended to hold a copy of the current clock frequency to allow fast reference to this frequency by software running on the PE. Need to program in CodeWarrior.

As described in **QorIQ LS2085A Family Reference Manual**:

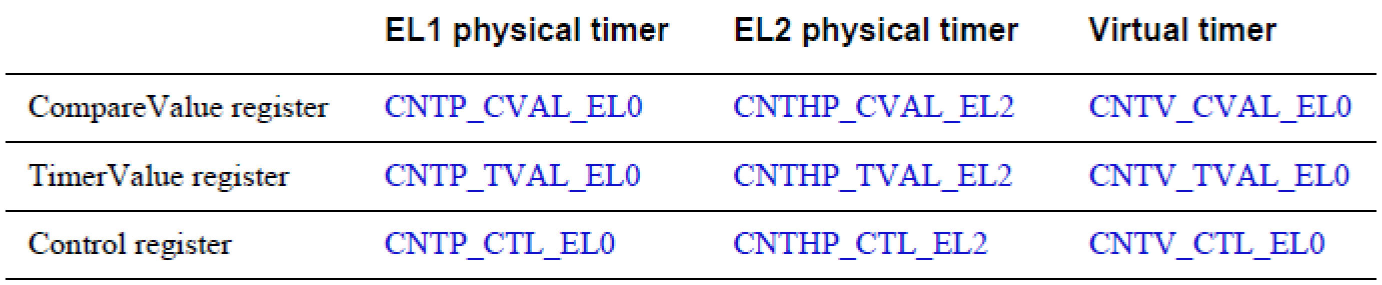
- RCW bits 301-292 SYSCLK_FREQ field is used for proper hardware configuration of the ARM Generic Timer.
- One must enable the timebase for all core clusters in the Power Management Unit - *((unsigned long *) 0x01e318a0) = 1;
- One must enable the clock for the Global Generic Reference Timer - *((unsigned long *) 0x023d0000) = 1;
- Currently I am setting the virtual comparator value at cntv_cval_el0 and enabling cntv_ctl_el0 to generate periodic interrupts.

Timers Provided

Timer	INTID* on LS2085A
Non-secure EL1 physical timer	30
Secure EL1 physical timer	29
Non-secure EL2 physical timer	26
Virtual timer	27

** Not documented in current RM but as recommended in GIC spec*

- Each timer is implemented as three registers:
  - A 64-bit CompareValue register, that provides a 64-bit unsigned upcounter.
  - A 32-bit TimerValue register, that provides a 32-bit signed downcounter.
  - A 32-bit Control register.
- In a multiprocessor implementation, each PE will have the same interrupt number for each timer.



	EL1 physical timer	EL2 physical timer	Virtual timer
CompareValue register	CNTP_CVAL_EL0	CNTHP_CVAL_EL2	CNTV_CVAL_EL0
TimerValue register	CNTP_TVAL_EL0	CNTHP_TVAL_EL2	CNTV_TVAL_EL0
Control register	CNTP_CTL_EL0	CNTHP_CTL_EL2	CNTV_CTL_EL0

EXAMPLE CODE

Sample Code – Output

```
Test of Interrupt from Generic Timer.
GIC-500 initialized.
Core0 Interrupts initialized.
IN InterruptHandler Interrupt: 27; Ticks Now:  125000017; Time Now: 5.000 secs; delta last: 5.000 secs
IN InterruptHandler Interrupt: 27; Ticks Now:  250000021; Time Now: 10.000 secs; delta last: 5.000 secs
IN InterruptHandler Interrupt: 27; Ticks Now:  375000026; Time Now: 15.000 secs; delta last: 5.000 secs
IN InterruptHandler Interrupt: 27; Ticks Now:  500000031; Time Now: 20.000 secs; delta last: 5.000 secs
IN InterruptHandler Interrupt: 27; Ticks Now:  625000036; Time Now: 25.000 secs; delta last: 5.000 secs
IN InterruptHandler Interrupt: 27; Ticks Now:  750000041; Time Now: 30.000 secs; delta last: 5.000 secs
IN InterruptHandler Interrupt: 27; Ticks Now:  875000046; Time Now: 35.000 secs; delta last: 5.000 secs
```

Sample Code – main.c

```
#include <stdio.h>
#include "interrupt_core0.h"
long core_cps = 25000000;

int main(void)
{
                unsigned long value, current = 0;
                int i = 0;

                printf("Test of Interrupt from Generic Timer.\n");
/* enable timebase for all clusters */
                *((unsigned long *) 0x01e318a0) = 0xf;              /* Power Management Unit */
/* enable clock for timer */;
                *((unsigned long *) 0x023d0000) = 0x1;              /* Global Generic Reference Timer */
/* enter assumed timebase */
                asm volatile("msr cntfrq_el0, %0" : : "r" (core_cps)); /* System Counter Frequency */
                isb();
/* enable non-secure (because SCR_EL3.NS = 1) Counter-timer EL1 Physical Control Register */
                /* Generated a INTID = 30 (non-secure EL1 physical timer) but how to control interval? */
//              asm("msr cntp_ctl_el0, %0" : : "r" (1<<0));
/* enable virtual comparison timer */
                asm volatile("mrs %0, cntvct_el0" : "=r" (current)); /* Counter-timer Virtual Count Register */
                asm volatile("mrs %0, cntfrq_el0" : "=r" (value));  /* System Counter Frequency */
                current += 5 * value;  /* try setting five second ahead */
                asm volatile("msr cntv_cval_el0, %0" : : "r" (current)); /* Virtual Compare Count Register */
                /* Generates a INTID = 27 (virtual timer interrupt) */
                current = (1<<0);
                asm("msr cntv_ctl_el0, %0" : : "r" (current));
/* enable an event stream from the virtual counter CNTKCTL_EL1 */
                /* THIS DIDN'T WORK.  MAY DEPEND ON WFE INSTRUCTION */
//              asm volatile("msr S3_0_C14_C1_0, %0" : : "r" (0x94)); /* Counter-timer Kernel Control register */
/* initialize and enable interrupts */
                configure_gic();
                configure_core();
/* Expect to see some kind of timer interrupt */
                while (1){ i++;};
                return 0;
}
```

Sample Code – configure_gic()

```
int configure_gic(void){
    unsigned long val, addr, value;
    value = GICD_CTLR_ARE_NS | GICD_CTLR_ARE_S | GICD_CTLR_ENABLE_G1S | GICD_CTLR_ENABLE_G1NS | GICD_CTLR_ENABLE_G0;
    *((unsigned int *) GICD_CTLR) = value;
    val = *((unsigned int *) (addr = GICR_WAKER));
    val &= ~GICR_WAKER_ProcessorSleep;
    *((unsigned int *) addr) = val;
    do {        val = *((unsigned int *) (GICR_WAKER));
                val &= GICR_WAKER_ChildrenAsleep;
    } while (val != 0);
    /* [4.2.2] Enable Non-secure system register access ICC_SRE_EL1 */
    asm volatile("mrs %0, S3_0_C12_C12_5" : "=r" (val));
    val |= ICC_SRE_EL1_SRE; /* Started with just enabling then added bypass for trial */
    asm volatile("msr S3_0_C12_C12_5, %0" : : "r" (val));
    /* [4.2.2] Enable system register access ICC_SRE_EL2 */
    asm volatile("mrs %0, S3_4_C12_C9_5" : "=r" (val));
    val |= ICC_SRE_EL1_SRE | 0x7;  /* Started with just enabling then added bypass for trial */
    asm volatile("msr S3_4_C12_C9_5, %0" : : "r" (val));
    /* [4.2.2] Enable system register access ICC_SRE_EL3 */
    asm volatile("mrs %0, S3_6_C12_C12_5" : "=r" (val));
    val |= ICC_SRE_EL1_SRE | 0x7;  /* Started with just enabling then added bypass for trial */
    asm volatile("msr S3_6_C12_C12_5, %0" : : "r" (val));
```

Sample Code – configure_gic() (cont’d)

```
        /* [4.2.2] Set priority mask ICC_PMR_EL1 and binary point registers ICC_BPRn_EL1 */
        asm volatile("msr S3_0_C4_C6_0, %0" : : "r" (0xf0)); /* ICC_PMR_EL1 interrupts w/priority >0xf0 will be signaled */
        asm volatile("msr S3_0_C12_C8_3, %0" : : "r" (0x3)); /* ICC_BPR0_EL1 */
        asm volatile("msr S3_0_C12_C12_3, %0" : : "r" (0x3)); /* ICC_BPR1_EL1 */
        /* [4.2.2] Set EOI mode ICC_CTLR_EL1 and ICC_CTLR_EL3 */
        asm volatile("msr S3_0_C12_C12_4, %0" : : "r" (0x0)); /* ICC_CTLR_EL1 */
        asm volatile("msr S3_6_C12_C12_4, %0" : : "r" (0x0)); /* ICC_CTLR_EL3 */
        /* [4.2.2] Enable signaling of each interrupt group ICC_IGRPEN1_EL1 ICC_IGRPEN0_EL1 or ICC_IGRPEN1_EL3 */
        /* set ICC_IGRPEN0_EL1 = S3_0_C12_C12_6 = 1 */
        asm volatile("msr S3_0_C12_C12_6, %0" : : "r" (1));
        isb();
        /* set ICC_IGRPEN1_EL1 = S3_0_C12_C12_7 = 1 */
        asm volatile("msr S3_0_C12_C12_7, %0" : : "r" (1));
        isb();
        /* set ICC_IGRPEN1_EL3 = S3_6_C12_C12_7 = 3*/
        asm volatile("msr S3_6_C12_C12_7, %0" : : "r" (3)); /* try setting both Secure Grp 1 and non-secure Grp 1 */
        isb();
        printf("GIC-500 initialized.\n");
        return 0;
    }
```

Sample Code – configure_core()

```
int configure_core(void){
    unsigned int addr, value;
    int i;
/* enable_all_interrupts */
    for (i=0; i<8; i++){   addr = GICR_IPRIORITYRn + i * 4;
                value = 0xa0a0a0a0;
                *((unsigned int *) addr) = value;      }
    *((unsigned int *) GICR_IGROUP0) = 0x00000000;
    *((unsigned int *) GICR_IGRPMOD0) = 0x00000000;
    addr = GICR_ICFGR + 0x4;  /* Interrupt Configuration Registers 0x0C04 */
    /* Interrupt Configuration Registers 0x0C00 is RA0/WI */
    value = 0;
    *((unsigned int *) addr) = value;
    /* Enable SGI and PPI (Private Peripheral Interrupts) */
    addr = GICR_ISENABLER0;
    value = 0x08000000;
    *((unsigned int *) addr) = value;
    printf("Core0 Interrupts initialized.\n");
    /* [4.2.3] Core configuration - Interrupt masks PSTATE */
    asm volatile("msr DAIFclr, #0xf"); // Might also work
    return 0;
}
```

Sample Code – interruptHandler()

```
void InterruptHandler()
{
        unsigned int iar;
        unsigned long currentp, currentv, value;
        double  Time_in_secs, Delta_in_secs;
        asm volatile("mrs %0, S3_0_C12_C8_0" : "=r" (iar)); /* ICC_IAR0_EL1 */

        asm volatile("mrs %0, cntpct_el0" : "=r" (currentp)); /* Counter-timer Physical Count Register */
        asm volatile("mrs %0, cntvct_el0" : "=r" (currentv)); /* Counter-timer Virtual Count Register */
        asm volatile("mrs %0, cntfrq_el0" : "=r" (value));
        Time_in_secs = (double) currentp / (double) value;
        Delta_in_secs = (double) (currentp - lastp) / (double) value;
        printf("IN InterruptHandler Interrupt: %d; Ticks Now:  %ld; Time Now: %5.3f secs; delta last: %5.3f secs\n",
                                            iar, currentp, Time_in_secs, Delta_in_secs);
        lastp = currentp;
        lastv = currentv;
        currentv = 5 * core_cps + lastv;  /* try setting five seconds ahead */
        asm volatile("msr cntv_cval_el0, %0" : : "r" (currentv)); /* Virtual Compare Count Register */
        /* inform the CPU interface completed processing of the specified Group 0 interrupt */
        asm volatile("msr S3_0_C12_C8_1, %0" : : "r" (iar));  /* ICC_EOIR0_EL1 */
}
```

Summary

- We have looked at the GIC500 (Generic Interrupt Controller) implementation on the NXP QorIQ LS2085A multicore ARMv8 SOC
- We discovered that the GIC500 directs interrupts to cores based on security state and current exception level. (e.g. A non-secure state should never process a secure interrupt and user level code would never service an OS interrupt.)
- We configured the GIC, the CPU and the Generic Timer to cause a periodic timer interrupt to be received and handled by core0.
- Sample code for a CodeWarrior bare-board project was provided that should now be more understandable and extensible to accommodate other interrupts.

SECURE CONNECTIONS
FOR A SMARTER WORLD

ATTRIBUTION STATEMENT