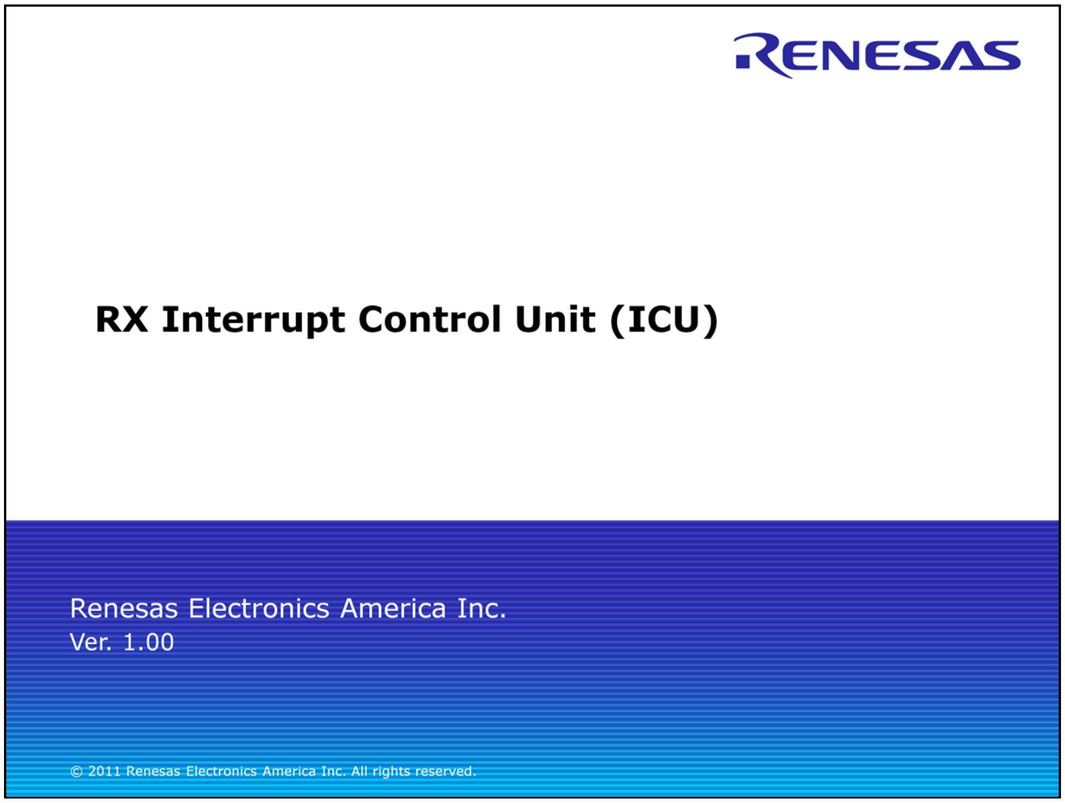
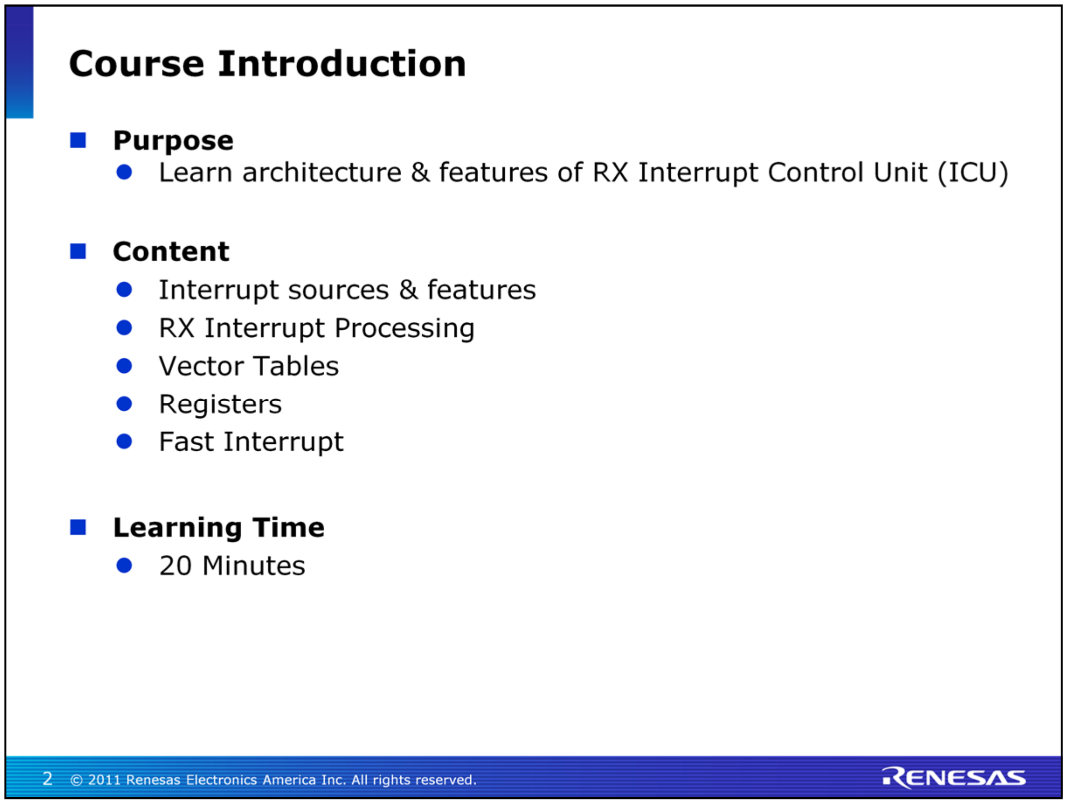


RX Interrupt Control Unit (ICU)

Renesas Electronics America Inc.
Ver. 1.00

Hello and welcome to this Renesas Interactive module that provides an architectural overview of the RX Interrupt Control Unit (ICU)



Course Introduction

- **Purpose**
  - Learn architecture & features of RX Interrupt Control Unit (ICU)
- **Content**
  - Interrupt sources & features
  - RX Interrupt Processing
  - Vector Tables
  - Registers
  - Fast Interrupt
- **Learning Time**
  - 20 Minutes

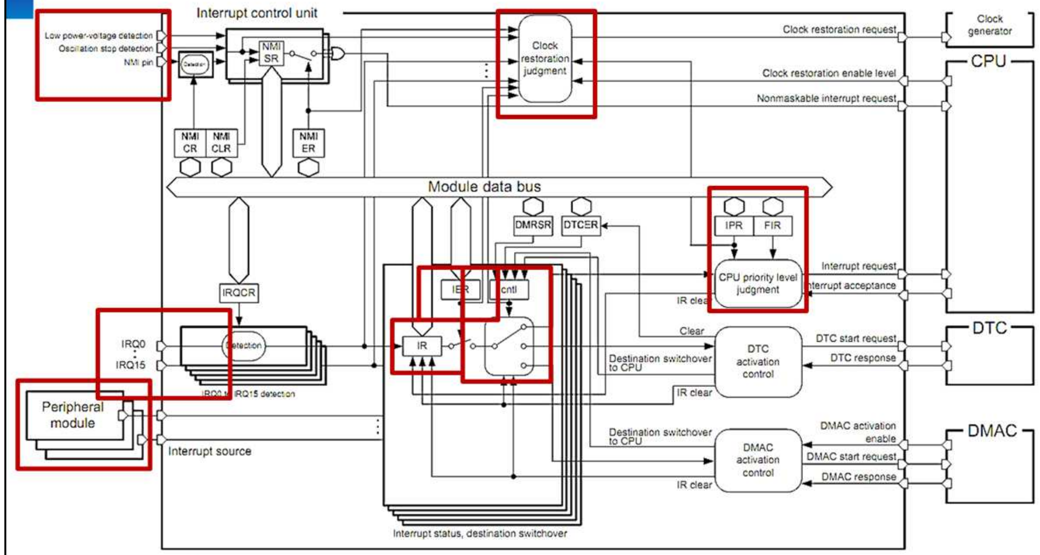
The purpose of this Renesas Interactive module is to give you a basic understanding of the RX Interrupt Control Unit.

In this course, we'll cover:

- Sources that generate interrupts on the RX, along with features common to most interrupts
- The steps that the CPU goes through to acknowledge and service interrupts
- The fixed & re-locatable vector tables
- Control registers in the ICU
- The RX Fast Interrupt mechanism

This session should be done in about 20 minutes.

RX Interrupt Control Unit



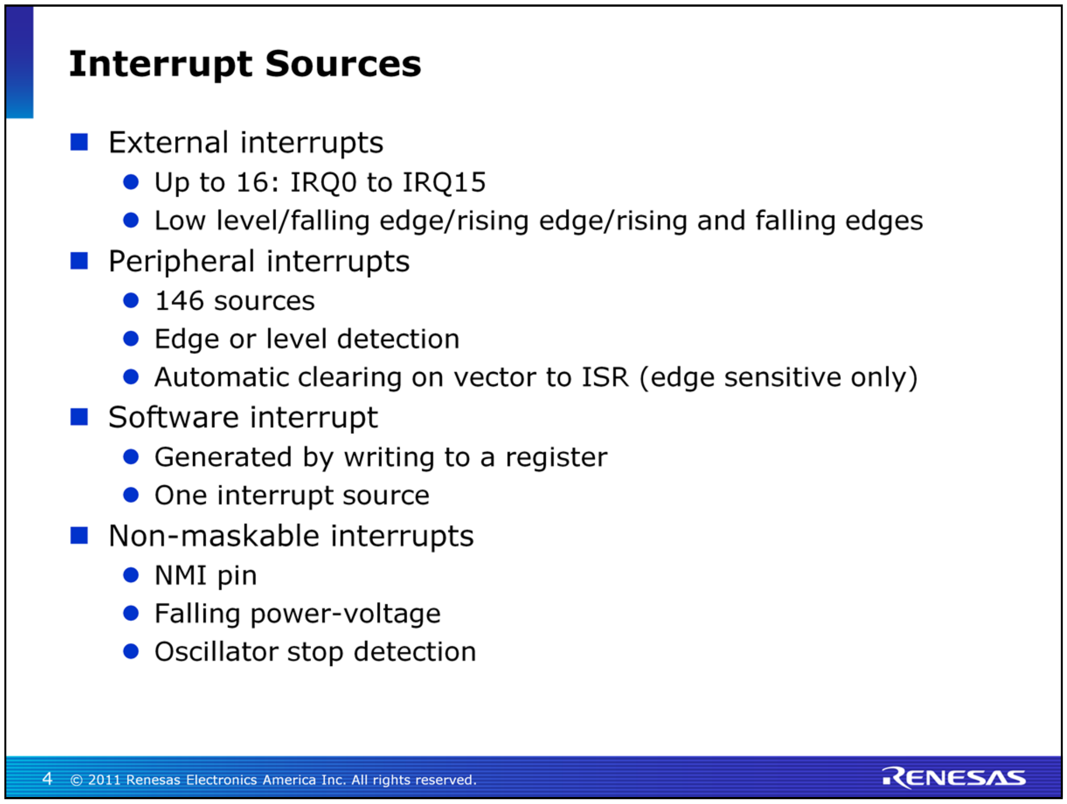
[Legend]

NMIER:	Non-maskable interrupt enable register
NMICR:	NMI pin interrupt control register
NMICLR:	Non-maskable interrupt clear register
NMISR:	Non-maskable interrupt status register
IRQCR:	IRQ control register
IR:	Interrupt request register
IER:	Interrupt request enable register
IPR:	Interrupt source priority register
FIR:	Fast interrupt set register
DTCER:	DTC activation enable register
DMRSR:	DMAC activation source select register

Here’s the block diagram of the RX Interrupt Control Unit. We’ll review each of the major functional blocks.

Interrupts can be generated by a number sources including:

- On-chip peripherals such as timers, AD converters, and communications channels
- External interrupt pins
- Special sources like the oscillation stop detection and low power detection circuits.
- Each source has an Interrupt Request status bit that is set when a particular source is pending
- Each source also has an Interrupt Enable Bit that gates the signal to CPU
- Control registers can be set to route the signal to the DTC or DMAC in place of the CPU in order to setup automatic data transfers to and from peripherals without interrupting the CPU.
- A priority scheme ensures that the most important interrupts are serviced first, and that higher priority interrupts can interrupt lower priority interrupts. It also allows the CPU to lock out low priority interrupts during critical processing sections.
- Finally, the ICU provides means to exit low-power modes of the CPU.



Interrupt Sources

- External interrupts
  - Up to 16: IRQ0 to IRQ15
  - Low level/falling edge/rising edge/rising and falling edges
- Peripheral interrupts
  - 146 sources
  - Edge or level detection
  - Automatic clearing on vector to ISR (edge sensitive only)
- Software interrupt
  - Generated by writing to a register
  - One interrupt source
- Non-maskable interrupts
  - NMI pin
  - Falling power-voltage
  - Oscillator stop detection

4

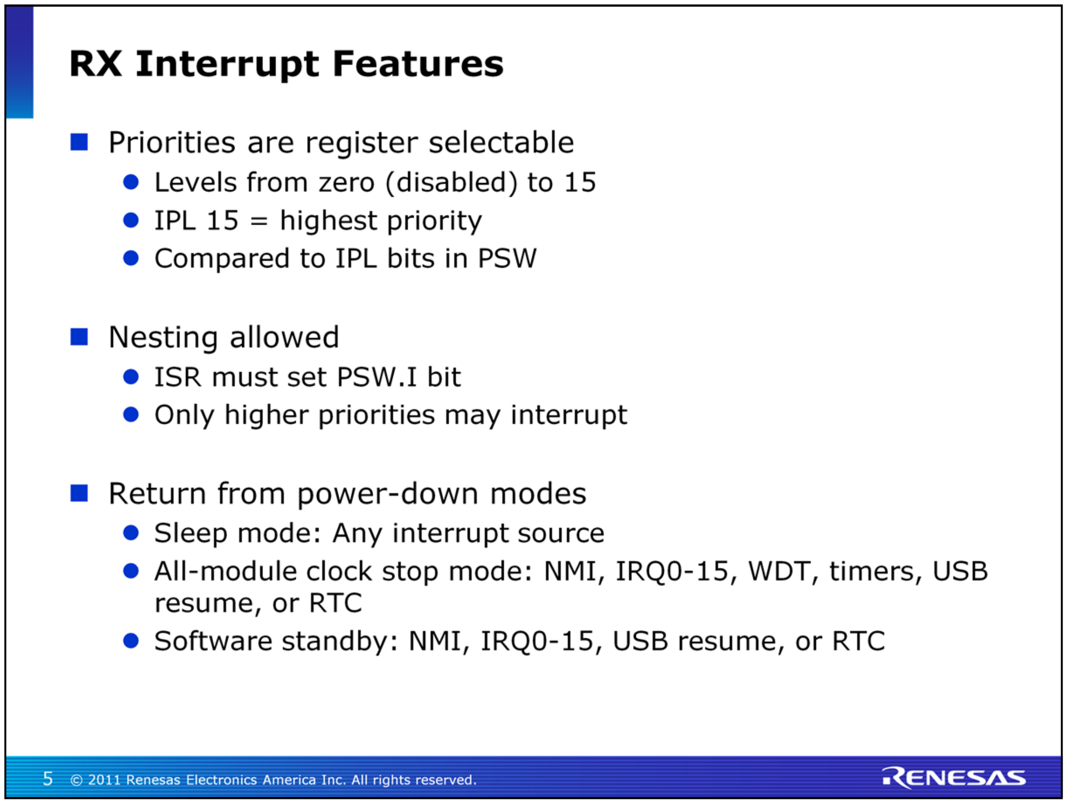
Interrupts on the RX can come from a number of sources.

There are external interrupts. These are generated by circuitry connected to the sixteen external IRQ pins. Not all 16 IRQ signals are available on all RX packages. Interrupts on the IRQ lines can be configured to be level or edge sensitive

The RX has an extremely rich set of peripherals that can generate interrupts. There are over 100 different peripheral interrupt sources. Depending on the source, these can be edge or level sensitive. For edge sensitive interrupts, the interrupt controller automatically clears these interrupts when it vectors to the ISR, simplifying your ISR.

A software interrupt is available triggered by write to a dedicated register.

Finally, there some special case interrupt sources used for systems monitoring including a dedicated Non-maskable Interrupt pin and interrupts for monitoring system voltage and the main system clock.



RX Interrupt Features

- Priorities are register selectable
  - Levels from zero (disabled) to 15
  - IPL 15 = highest priority
  - Compared to IPL bits in PSW
- Nesting allowed
  - ISR must set PSW.I bit
  - Only higher priorities may interrupt
- Return from power-down modes
  - Sleep mode: Any interrupt source
  - All-module clock stop mode: NMI, IRQ0-15, WDT, timers, USB resume, or RTC
  - Software standby: NMI, IRQ0-15, USB resume, or RTC

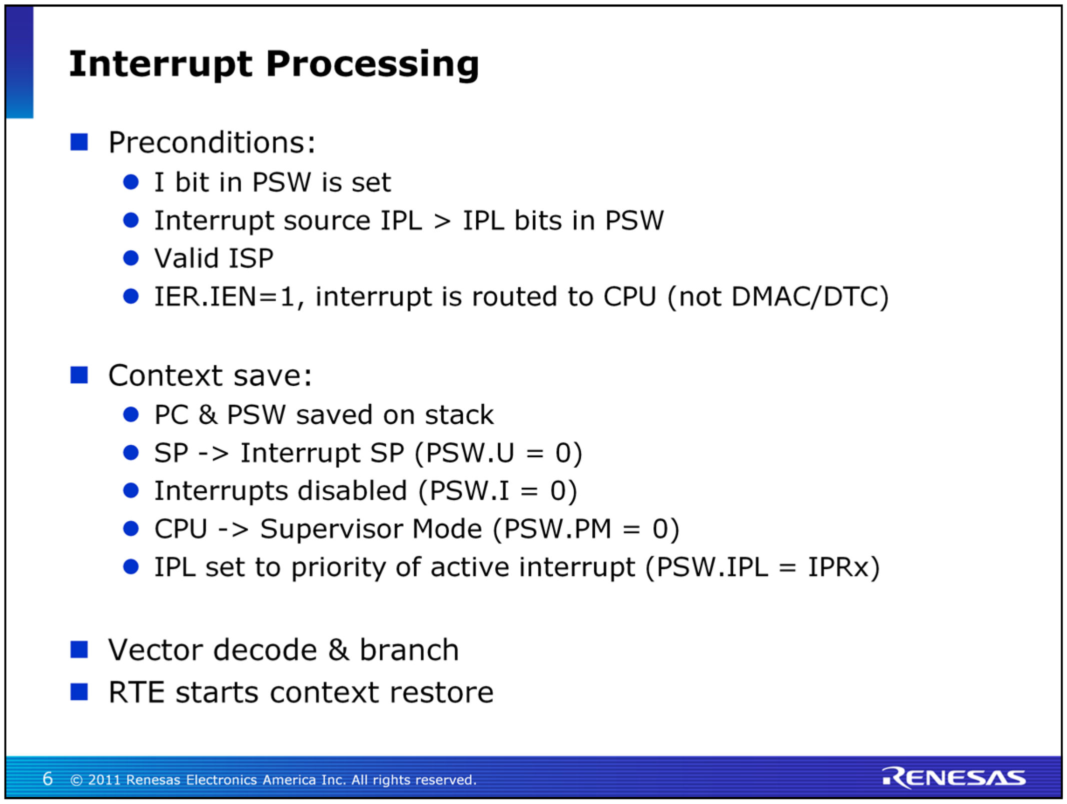
5

Now let's see some of the features that apply generally to RX interrupts.

First, the RX has a multiple priority level interrupt structure. Each interrupt source must be assigned a priority level. Priority levels range from the lowest level of 0 to the highest level of 15. A level of zero disables the interrupt. An interrupt priority level of 15 is the highest priority in the system. There is a corresponding Interrupt Priority Level in the Program Status Word. When an interrupt goes active, its interrupt priority level is compared to the current IPL in the PSW. If the interrupt's priority level is higher, the interrupt is taken, otherwise it remains pending. If two interrupts become active at the same time, the higher priority interrupt is serviced first.

Priority levels also make nesting possible.   In order to enable nesting, your ISR must set the global interrupt flag: the I bit in the Program Status Word. The Renesas compiler provides an optional directive that does this for you when you prototype your ISR.

Nesting allows interrupts with a higher priority to be serviced during lower level interrupts. Interrupts also provide the mechanism for the RX to return from low-power modes. Depending on the mode, different interrupts can bring the part back to run mode



Interrupt Processing

- Preconditions:
  - I bit in PSW is set
  - Interrupt source IPL > IPL bits in PSW
  - Valid ISP
  - IER.IEN=1, interrupt is routed to CPU (not DMAC/DTC)
- Context save:
  - PC & PSW saved on stack
  - SP -> Interrupt SP (PSW.U = 0)
  - Interrupts disabled (PSW.I = 0)
  - CPU -> Supervisor Mode (PSW.PM = 0)
  - IPL set to priority of active interrupt (PSW.IPL = IPRx)
- Vector decode & branch
- RTE starts context restore

6 RENESAS

Let's look at what happens during processing of an interrupt. There are some register references here that will be discussed on later slides.

First, it's important to understand that the following preconditions need to be in place. The global interrupt enable bit, the I bit, in the Program Status Word must be set to 1. The priority level for the interrupt source must be greater than the current interrupt priority level in the PSW. And, since interrupts use a separate stack, the Interrupt Stack Pointer must be initialized. These first three items are mentioned here for completeness. Startup code generated by the tool chain usually take care of these details.

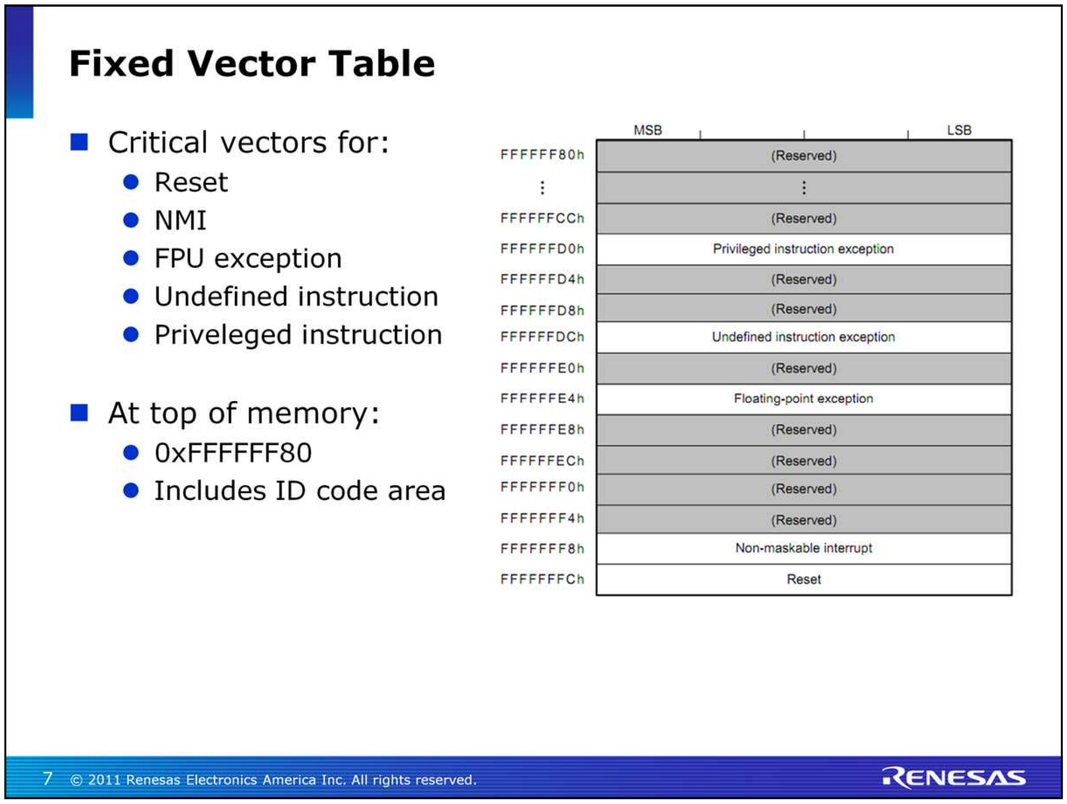
Finally, the interrupt source must be enabled in the ICU and routed to the CPU rather than the DMAC or DTC. Your code is responsible for this.

With all of this in place, when the interrupt fires the CPU takes the interrupt and starts by saving the current processing context. These steps are automatically done by the CPU.

- First, the Program Counter and Program Status Word are pushed onto the stack.
- The stack pointer is switched from the User Stack Pointer to the Interrupt Stack Pointer by clearing the U bit in the PSW.
- The global interrupt enable bit in the PSW is cleared, disabling further interrupts.
- The CPU switches to supervisor mode by clearing the PM bit in the PSW.
- The IPL bit field in the PSW is set to the priority of the active interrupt. This allows nesting of higher priority interrupts if the ISR chooses to set the PSW I bit.

Now the CPU decodes the proper vector and branches to your code in the Interrupt Service Routine.

When your service routine is complete, it executes a Return From Exception instruction, and the context saved earlier is restored and normal processing is resumed.



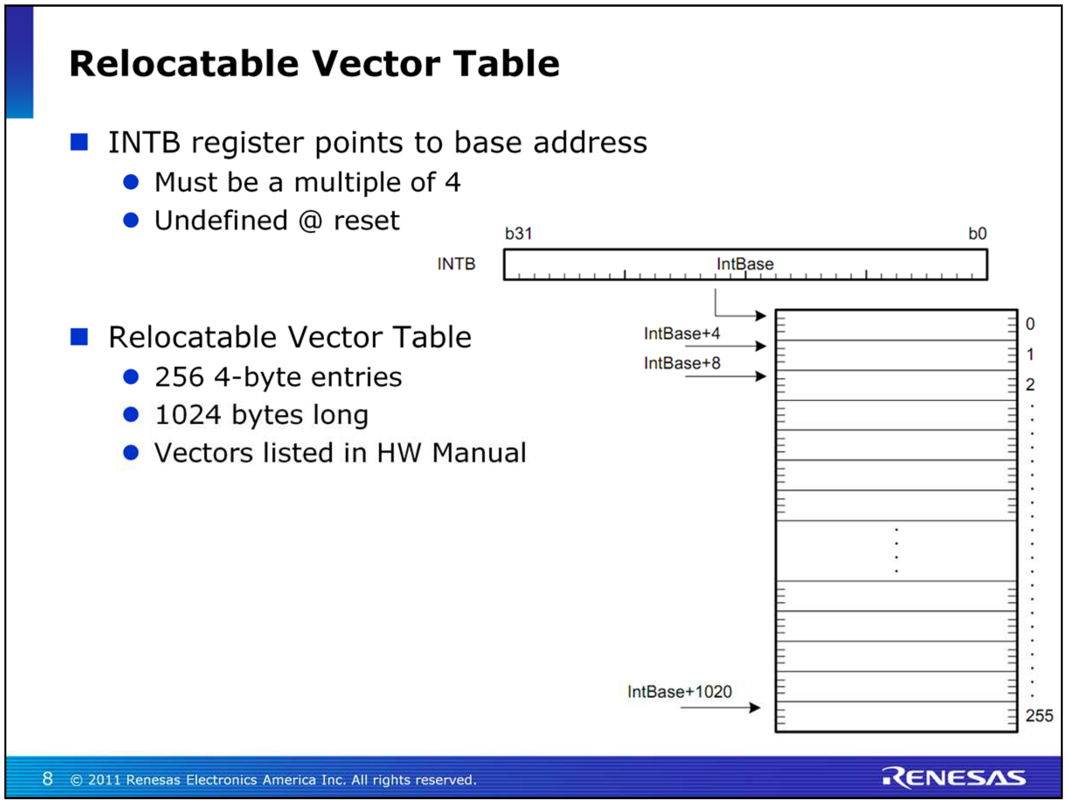
Fixed Vector Table

- Critical vectors for:
  - Reset
  - NMI
  - FPU exception
  - Undefined instruction
  - Priveleged instruction
- At top of memory:
  - 0xFFFFFF80
  - Includes ID code area

	MSB … LSB
FFFFFF80h	(Reserved)
⋮	⋮
FFFFFFCCh	(Reserved)
FFFFFFD0h	Privileged instruction exception
FFFFFFD4h	(Reserved)
FFFFFFD8h	(Reserved)
FFFFFFDCh	Undefined instruction exception
FFFFFFE0h	(Reserved)
FFFFFFE4h	Floating-point exception
FFFFFFE8h	(Reserved)
FFFFFFECh	(Reserved)
FFFFFFF0h	(Reserved)
FFFFFFF4h	(Reserved)
FFFFFFF8h	Non-maskable interrupt
FFFFFFFCh	Reset

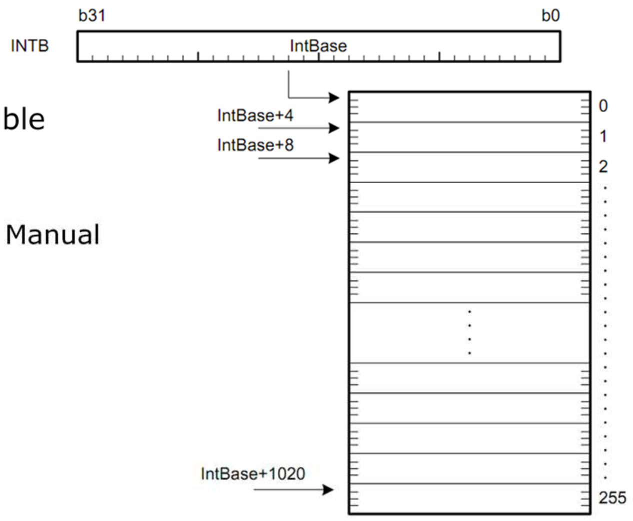
There are two vectors tables in the RX: a fixed vector table and a re-locatable vector table.

The fixed vector table includes critical vectors for a number of special exceptions. It resides at the very top of the 32-bit memory map. Entries listed as “Reserved” should be set to hex FF FF FF FF.



Relocatable Vector Table

- INTB register points to base address
  - Must be a multiple of 4
  - Undefined @ reset
- Relocatable Vector Table
  - 256 4-byte entries
  - 1024 bytes long
  - Vectors listed in HW Manual

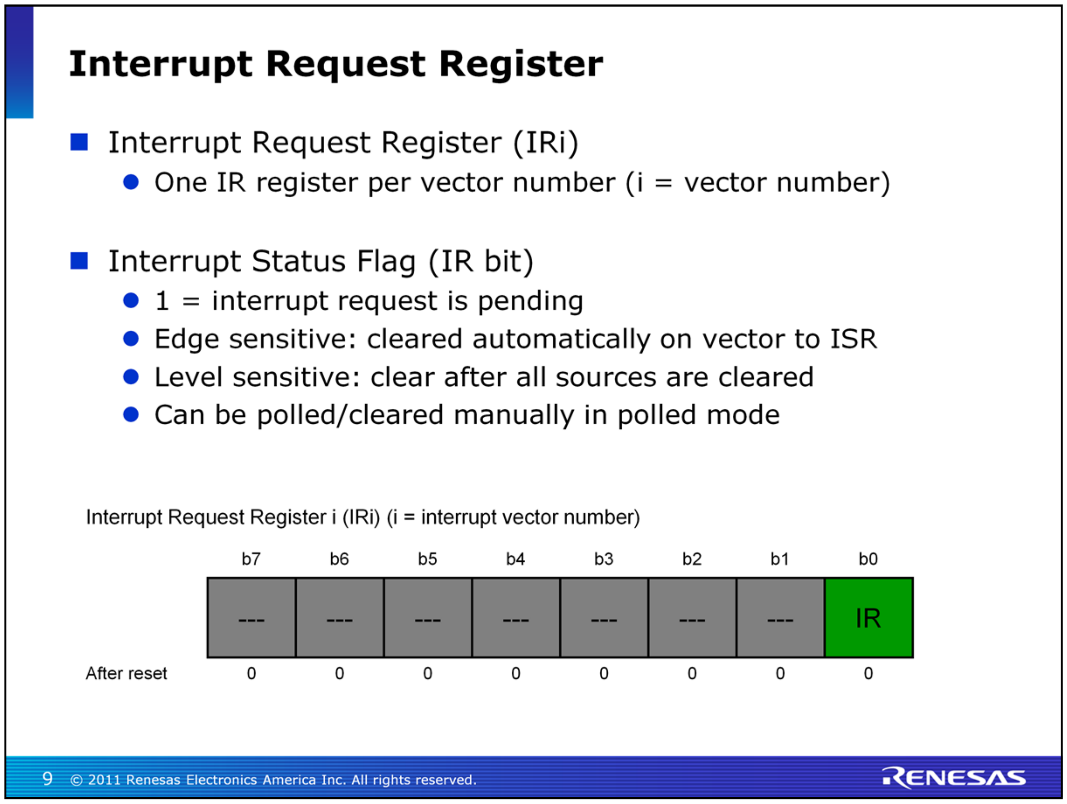


The RX also has a re-locatable vector table.

A dedicated CPU register, the Interrupt Base Register, points to the starting address of the re-locatable vector table.

The address in the INTB register must be an even multiple of 4 and it's important to remember that the INTB register is undefined at reset. Usually your C startup code will initialize this properly, but if your code is running off into the weeds when an interrupt fires it's a good idea to check this register.

The Re-locatable Vector Table has 256 entries, each contains the 4-byte address of an interrupt service routine. Some of the entries are reserved, so check the hardware manual for details.



Interrupt Request Register

- Interrupt Request Register (IRi)
  - One IR register per vector number (i = vector number)
- Interrupt Status Flag (IR bit)
  - 1 = interrupt request is pending
  - Edge sensitive: cleared automatically on vector to ISR
  - Level sensitive: clear after all sources are cleared
  - Can be polled/cleared manually in polled mode

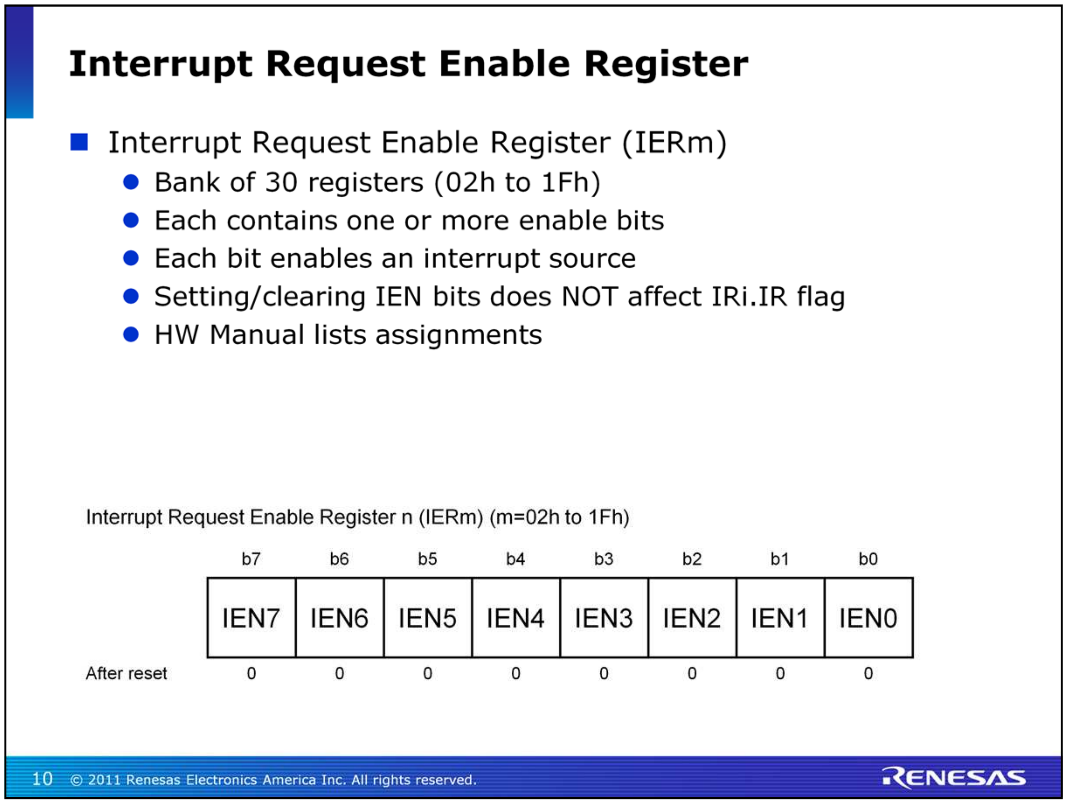
Interrupt Request Register i (IRi) (i = interrupt vector number)

	b7	b6	b5	b4	b3	b2	b1	b0
	---	---	---	---	---	---	---	IR
After reset	0	0	0	0	0	0	0	0

Now let's review some of the registers in the Interrupt Control Unit that affect interrupt operation.

First up is the Interrupt Request Register. There is one IR register for each interrupt source in the RX, indexed by their matching vector number.

Each IR register has a single field: the Interrupt Status Flag or IR bit. When IR is set to 1, an interrupt request for this source is pending. Clearing of the IR bit depends on the type of interrupt detection for the source that triggered the interrupt. For edge-sensitive interrupts, the IR bit is cleared automatically when the interrupt is taken. For level sensitive interrupts, the IR bit remains set until all pending interrupts for that source have been serviced. The IR bit can be polled in non-interrupt mode to see when a source has gone active. If you are using a peripheral in polled mode, you'll need to clear this bit manually.



Interrupt Request Enable Register

- Interrupt Request Enable Register (IERm)
  - Bank of 30 registers (02h to 1Fh)
  - Each contains one or more enable bits
  - Each bit enables an interrupt source
  - Setting/clearing IEN bits does NOT affect IRi.IR flag
  - HW Manual lists assignments

Interrupt Request Enable Register n (IERm) (m=02h to 1Fh)

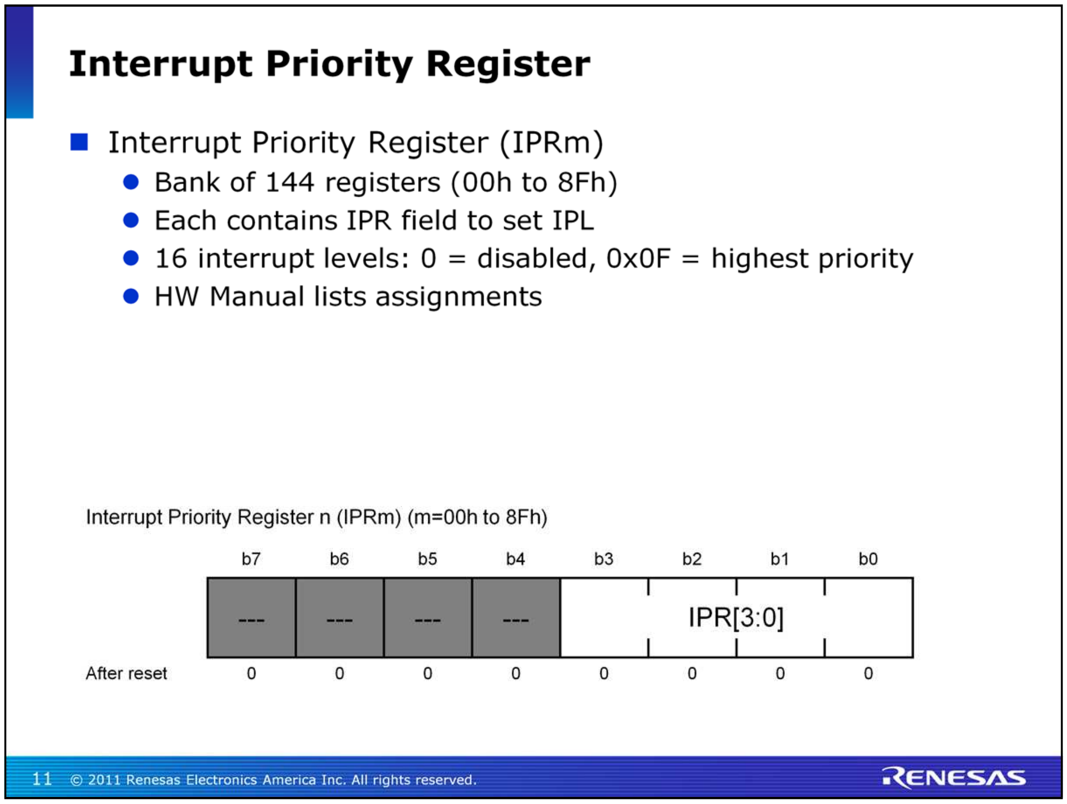
	b7	b6	b5	b4	b3	b2	b1	b0
	IEN7	IEN6	IEN5	IEN4	IEN3	IEN2	IEN1	IEN0
After reset	0	0	0	0	0	0	0	0

10 © 2011 Renesas Electronics America Inc. All rights reserved. RENESAS

Next up is the Interrupt Request Enable Registers, or IER registers. The IER is used to turn interrupts for individual sources on and off.

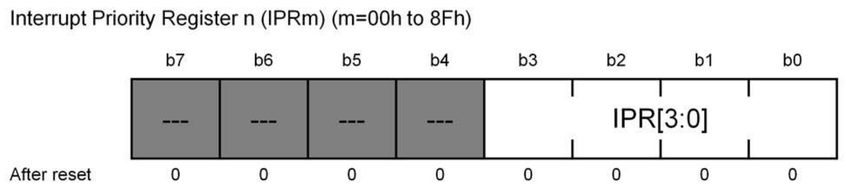
There are 30 IER registers and each contains enable bits for multiple sources. Each IEN enable bit enables a single interrupt source and setting or clearing the IEN bit does NOT affect the IR bit in the Interrupt Request Register from the last slide. So make sure you clear the IR bit before enabling an interrupt or the interrupt will fire right away.

The hardware manual details the register and bit assignments for each interrupt source.



Interrupt Priority Register

- Interrupt Priority Register (IPRm)
  - Bank of 144 registers (00h to 8Fh)
  - Each contains IPR field to set IPL
  - 16 interrupt levels: 0 = disabled, 0x0F = highest priority
  - HW Manual lists assignments

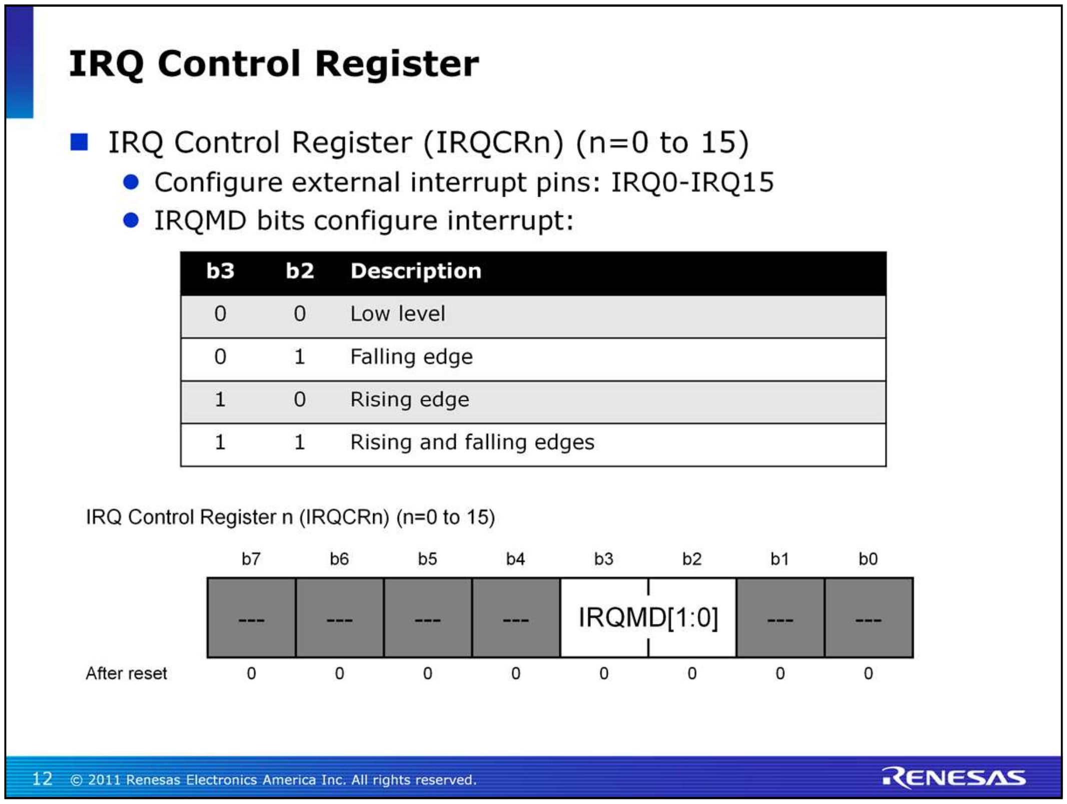


11 © 2011 Renesas Electronics America Inc. All rights reserved.

RENESAS

As we learned earlier, the RX supports multiple interrupt priority levels. This brings us to the Interrupt Priority Registers. There is one IPR register for each interrupt source group. Multiple discreet interrupt sources may be contained in one group and controlled by one IPR register. The IPR contains one field to set the priority level of a given interrupt group. There are sixteen interrupt priority levels ranging from 0 to 15 or hex 0x0F. Higher values are higher priority, and a priority of zero disables an interrupt.

The ICU chapter in the HW manual lists the corresponding IPR for each interrupt source



IRQ Control Register

- IRQ Control Register (IRQCRn) (n=0 to 15)
  - Configure external interrupt pins: IRQ0-IRQ15
  - IRQMD bits configure interrupt:

b3	b2	Description
0	0	Low level
0	1	Falling edge
1	0	Rising edge
1	1	Rising and falling edges

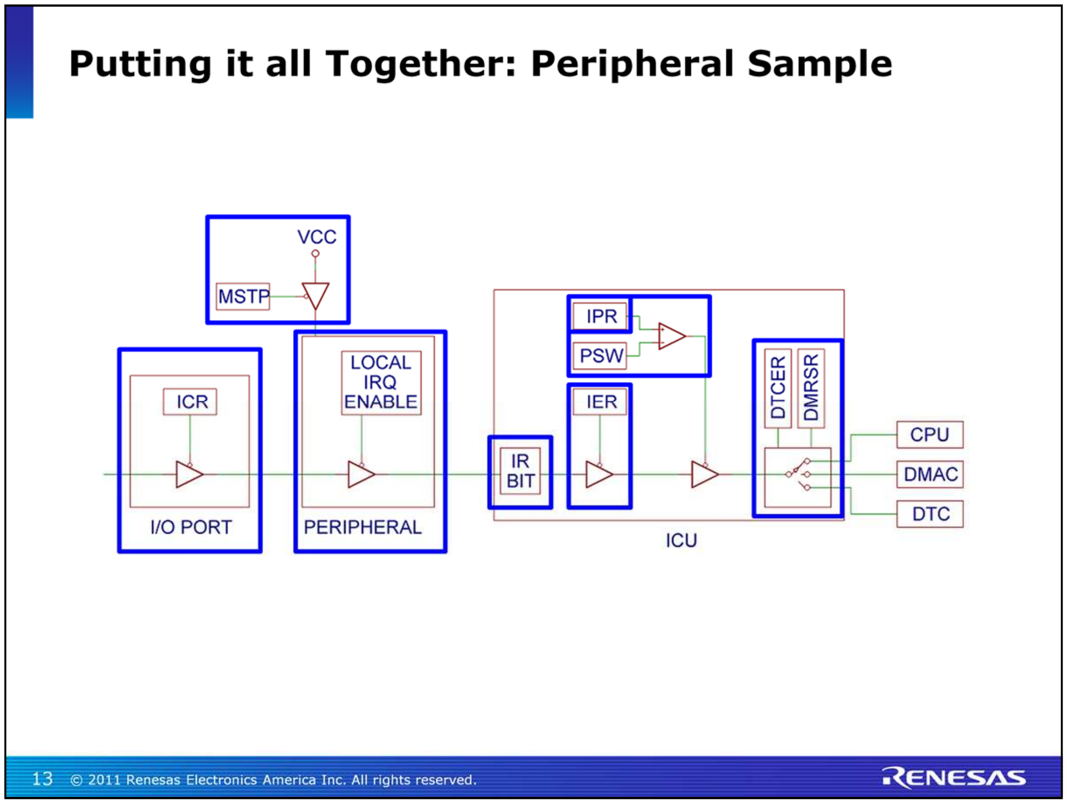
IRQ Control Register n (IRQCRn) (n=0 to 15)

	b7	b6	b5	b4	b3	b2	b1	b0
	---	---	---	---	IRQMD[1:0]		---	---
After reset	0	0	0	0	0	0	0	0

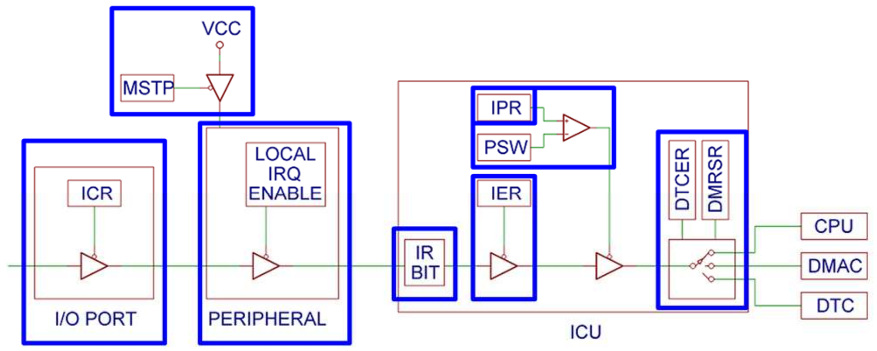
The external IRQ pins have an additional control register: the IRQ Control Register.

There are 16 IRQCR registers one for each IRQ pin to configure how interrupts are detected for each.

The IRQMD bits in the register determine whether the interrupt looks for a low level, rising edges, falling edges, or both edges.



Putting it all Together: Peripheral Sample



Let's see how to put this all together to generate interrupts from an on-chip peripheral. In this case we'll talk about a serial communications interface or SCI, but similar steps are necessary for other peripherals.

Here's a simplified schematic view of the on-chip systems we must setup. There are three main sections that we need to configure: The I/O port that connects to the outside world, the SCI peripheral, and the interrupt control unit or ICU.

Starting with the I/O port and the external pins, we have to configure the I/O ports properly. In the case of a serial channel, we need to set the receive pin as an input, and to enable the input buffer by setting the proper ICR bit. This connects the signal from the pin to the on-chip peripheral. With the ICR set the signal now reaches the SCI. Before we can begin writing to the SCI's registers, we have to take the SCI out of stop mode by clearing the MSTP bit for it. Remember that to save power most of the RX's peripherals are OFF by default and we have to turn them on before we can use them.

With the SCI powered up, we can configure the baud rate and other communications settings. RX peripherals have local interrupt enable bits that gate the interrupt signal from the peripheral into the ICU. In the case of the SCI, there are four different interrupts that can be generated by the SCI to the ICU.

With the local interrupt enables set in the peripheral, the IR bit for each corresponding interrupt source becomes live in the ICU. At this point, applications that poll for status rather than using interrupts can look at the IR bit to see the peripheral status. In polled applications the IR bit must be cleared manually by your code. We need to set priority level for this interrupt by writing to the proper Interrupt Priority Register.

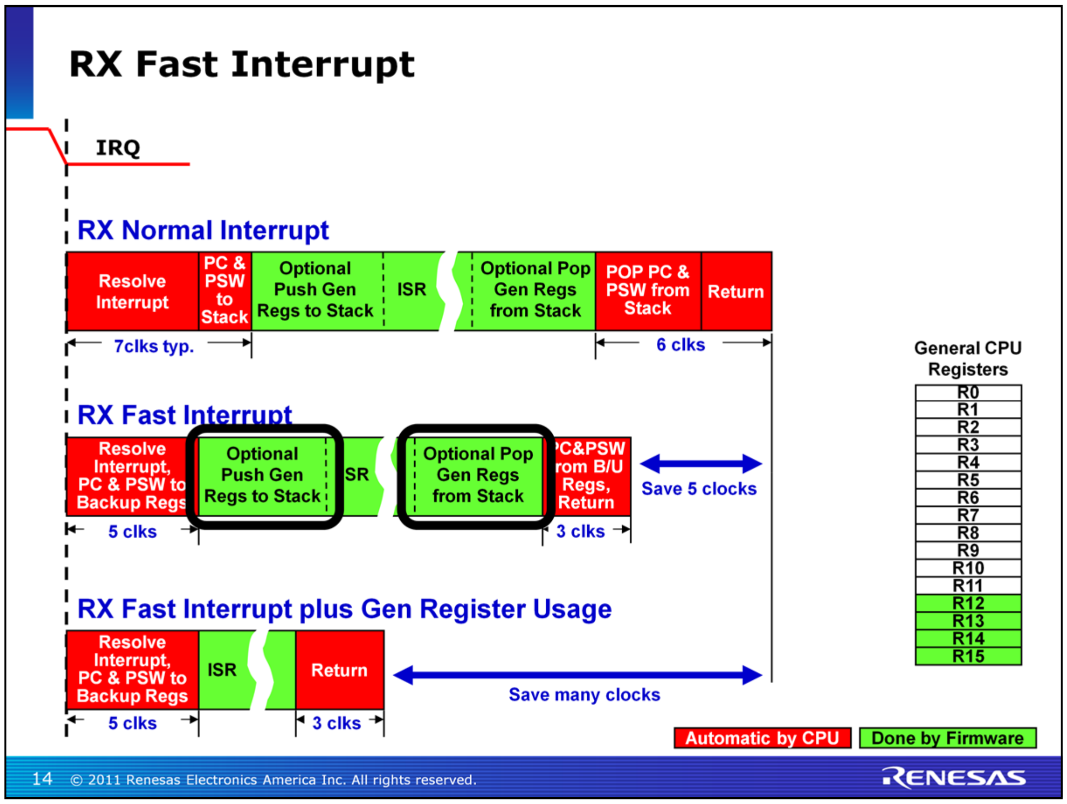
Now that the peripheral signal is making it into the ICU, we need to enable the interrupt for each source using the proper IER bit in Interrupt Request Enable Register. Remember that each register contains bits for more than one interrupt source, so be careful with how you modify the registers. When the interrupt goes active, it will only be serviced if it's priority level is greater than that of the current interrupt priority level in the CPU's Program Status Word.

While this description sounds like a lot, it only takes a handful of lines of code in practice.

It is possible to route interrupt signals to the DMA controller or the DTC to automate moving of data without the need for CPU intervention.

Refer to the Renesas Interactive modules on DMA and DTC for details.

RX Fast Interrupt


IRQ
RX Normal Interrupt
Resolve Interrupt
PC & PSW to Stack
Optional Push Gen Regs to Stack
ISR
Optional Pop Gen Regs from Stack
POP PC & PSW from Stack
Return
7clks typ.
6 clks
RX Fast Interrupt
Resolve Interrupt, PC & PSW to Backup Regs
Optional Push Gen Regs to Stack
ISR
Optional Pop Gen Regs from Stack
PC&PSW from B/U Regs, Return
Save 5 clocks
5 clks
3 clks
RX Fast Interrupt plus Gen Register Usage
Resolve Interrupt, PC & PSW to Backup Regs
ISR
Return
Save many clocks
5 clks
3 clks
General CPU Registers
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
Automatic by CPU
Done by Firmware
14 © 2011 Renesas Electronics America Inc. All rights reserved.

We've seen how to get interrupts enabled and routed to the CPU core. Let's take a look at what happens once interrupt processing starts, and how we can optimize it.

Here's how the RX handles a normal interrupt:

Once the current instruction completes and the interrupt fires, the CPU automatically resolves the interrupt source and selects the correct vector, pushes the Program Status Word and Program Counter on the stack, modifies the PSW to reflect the current interrupt state, and then starts execution of the user's Interrupt Service Route, or ISR. This hardware automated portion of this process takes typically 7 clock cycles, at which point we're in our ISR.

The first thing we need to do is to save whatever registers we'll be using onto the stack so that we can restore them on the way out of the ISR. Then we have our actual ISR processing, followed by the restoration of those registers we saved on the way in. This completes software interrupt processing, and as the ISR is exited with the RTE instruction hardware processing continues with the hardware restoration of the PC and the PSW from the stack. This portion takes 6 clocks.

This is pretty efficient, but what if we could cut out the pushing and popping of the PSW and PC? Well we can with the RX Fast Interrupt. You can specify one interrupt source as the Fast Interrupt. The Fast Interrupt differs from other interrupts in that the PC and PSW, instead of being stored on the stack, are instead stored in dedicated backup registers which are much faster to access. So the hardware portion of the context save & restore are speed up.

Here's what it looks like:

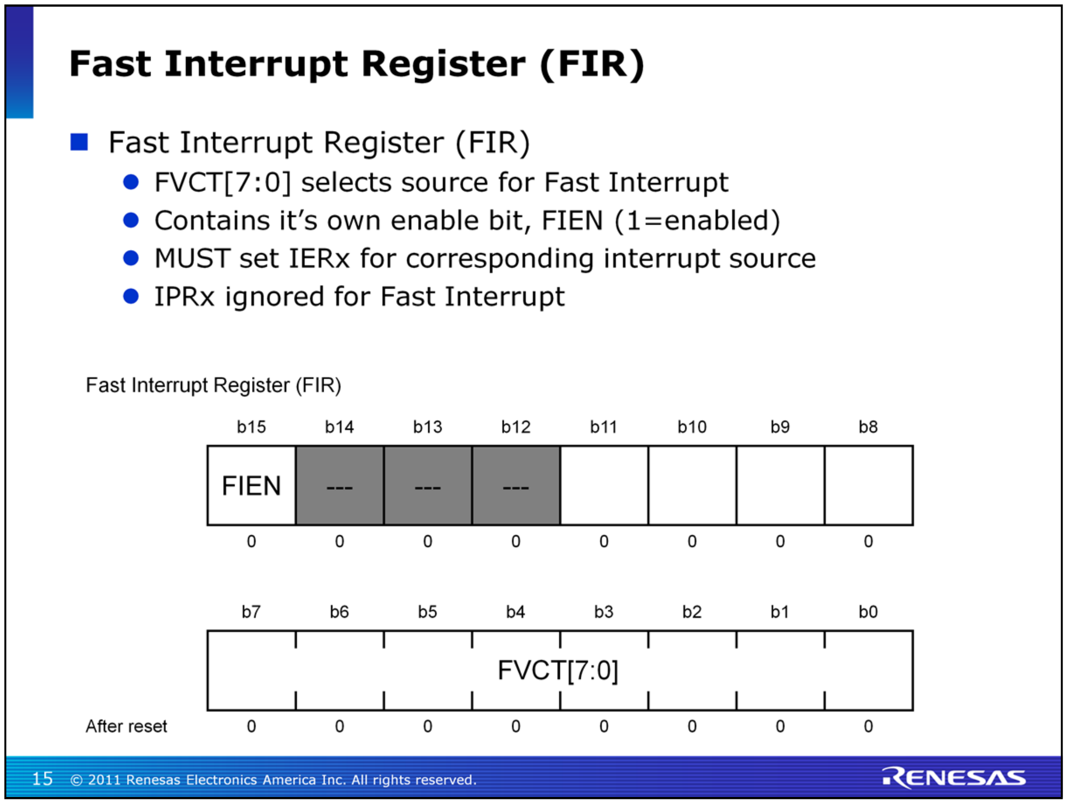
When the interrupt fires now the PC and PSW are stored in the backup registers. This saves 2 clocks on entry. As before, software saves additional registers on entry to the ISR, executes the ISR code, and then restores registers on exit. Once the Return From Exception instruction is executed, the hardware portion of the context restore is shortened by 3 clocks over a normal interrupt as the PC and PSW are restored from the backup registers rather than the stack.

Using the fast interrupt, we've saved 5 clocks over a standard interrupt. This is a nice improvement, but you can see there is still some housekeeping going on in software as we enter the ISR. Software has to save to the stack the registers it will use, and then pop them off on the way out. Wouldn't it be great if we could dedicate a few of the general purpose registers for use by the ISR to eliminate this overhead?

The RX compiler allows you to set aside up to four registers for use only by the fast interrupt routine.

Let's see how the ISR looks now:

We still have the same hardware entry, typically 5 clocks. But now there is no need to push or pop registers, the ISR starts running immediately with no need for saving context on entry, and no restoring of context on exit. And the hardware side of the return remains at 3 clocks. By setting aside a small block of registers, you can save many clocks over a standard interrupt. And while your main line code may experience a small decrease in performance as a result of running off of a smaller register set in this mode, with the RX you have the choice of how you'd like to optimize performance for your application.



Fast Interrupt Register (FIR)

- Fast Interrupt Register (FIR)
  - FVCT[7:0] selects source for Fast Interrupt
  - Contains it's own enable bit, FIEN (1=enabled)
  - MUST set IERx for corresponding interrupt source
  - IPRx ignored for Fast Interrupt

Fast Interrupt Register (FIR)

	b15	b14	b13	b12	b11	b10	b9	b8
	FIEN	---	---	---				
	0	0	0	0	0	0	0	0

	b7	b6	b5	b4	b3	b2	b1	b0
	FVCT[7:0]							
After reset	0	0	0	0	0	0	0	0

15 © 2011 Renesas Electronics America Inc. All rights reserved. RENESAS

A dedicated register in the ICU is used to configure the Fast Interrupt Vector.

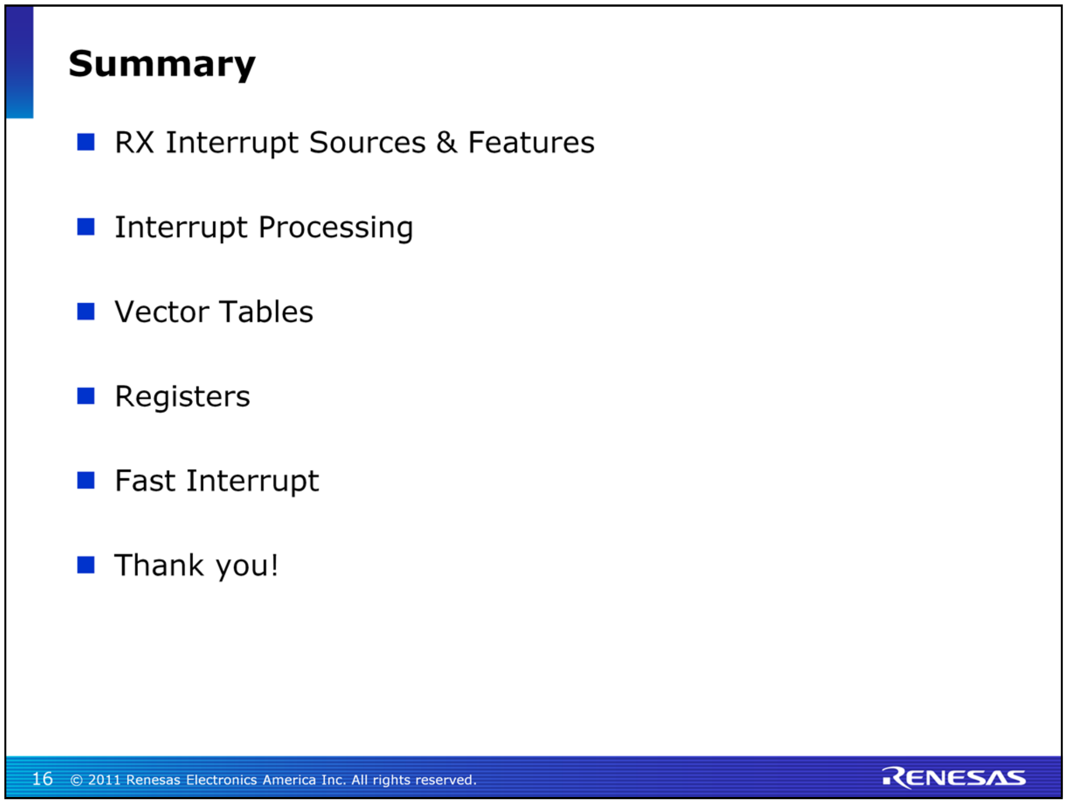
The Fast Interrupt Register, or FIR, specifies the vector designated as the Fast Interrupt. There are two fields in the FIR register. Place the vector number for the interrupt you want to be the fast interrupt into the FVCT field.

The FIEN bit enables or disables the Fast Interrupt. Set it to 1 to enable it.

Don't forget that the Interrupt Request Enable must be set for this source, too

The Interrupt Priority Register for the source selected as the Fast Interrupt is ignored. Fast Interrupts effectively have the highest priority in the system.

As a final note, you need to specify the ISR for the Fast Interrupt in your C source code using one of your compiler's directives so that the compiler generates the proper Return From Fast Interrupt instruction at the end of the ISR rather than the typical RTE instruction. Check your compiler documentation for details.



Summary

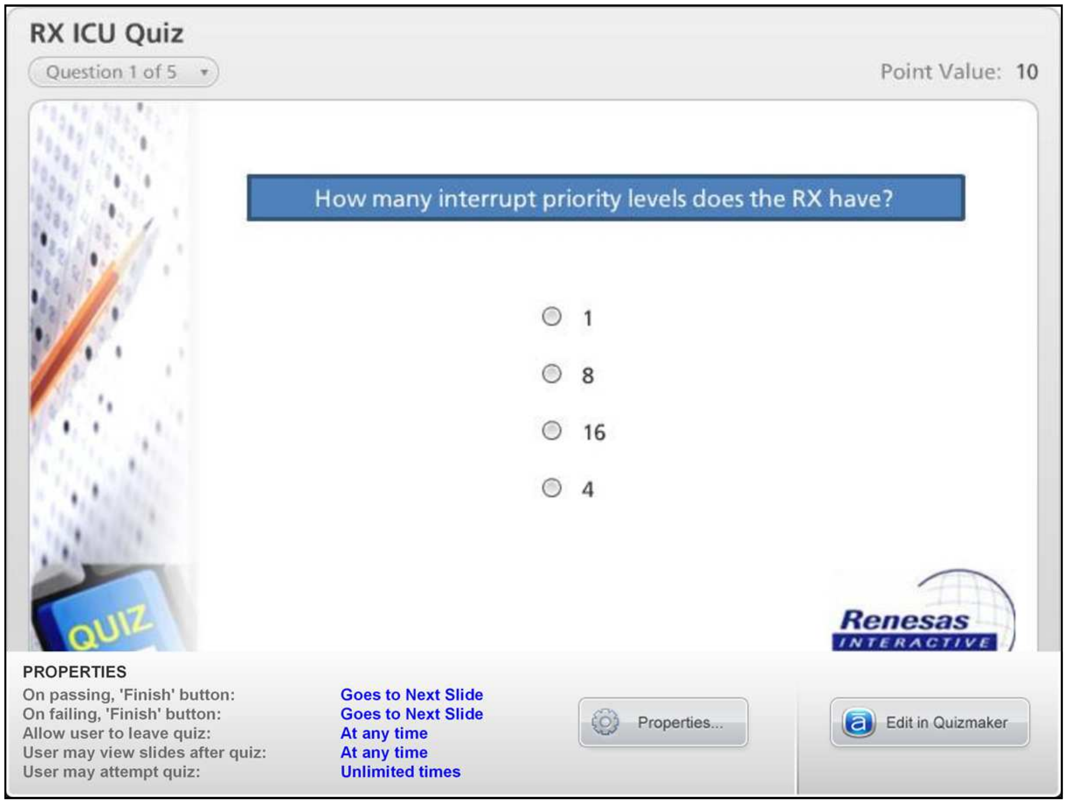
- RX Interrupt Sources & Features
- Interrupt Processing
- Vector Tables
- Registers
- Fast Interrupt
- Thank you!

16 © 2011 Renesas Electronics America Inc. All rights reserved.

In this module you've learned about the RX Interrupt Control Unit including:

- Sources & features of RX interrupts
- Standard RX interrupt processing
- The fixed & re-locatable vector tables
- Registers in the ICU to configure interrupt behavior
- Fast Interrupt mechanism of the RX

Thanks for watching!

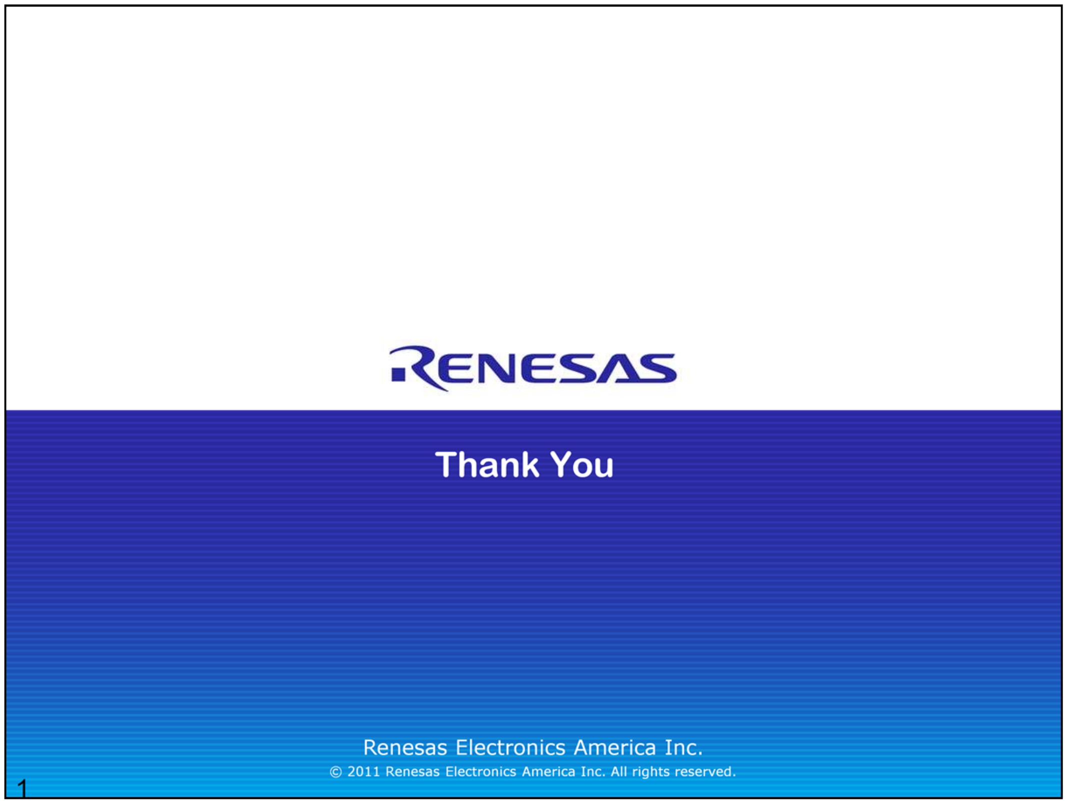


How many interrupt priority levels does the RX have?

- ○ 1
- ○ 8
- ○ 16
- ○ 4

PROPERTIES

On passing, 'Finish' button:	Goes to Next Slide
On failing, 'Finish' button:	Goes to Next Slide
Allow user to leave quiz:	At any time
User may view slides after quiz:	At any time
User may attempt quiz:	Unlimited times



RENESAS

Thank You

Renesas Electronics America Inc.

1

Thank You