Using Linux 2.4.18 to Initialize the PCI Subsystem in IDT Integrated Communications Processors

Application Note AN-447

Notes

By Harpinder Singh

Introduction

Although this application note describes the software required to properly configure the PCI interface integrated within the RC32438 integrated communications processor running the Linux operating system, the concepts described apply equally to other software environments and to other members of the IDT™ Interprise™ processor family.

Overview

The RC32438 is a member of the IDT™ Interprise™ family of PCI integrated communications processors. It incorporates a high performance CPU core and a number of on-chip peripherals. The Interprise processor family frequently utilizes PCI as an interconnection to inexpensive peripheral devices, such as additional Ethernet interfaces, disk controllers, multimedia codecs, and wireless access devices. The integrated PCI bridge is 32 bit, PCI revision 2.2 compliant, supports clock frequencies from 16 MHz to 66 MHz, and has a built-in PCI arbiter supporting up to 6 external masters in the host mode.

An important feature of the PCI subsystem is the address translation and byte swapping attributes for transactions across the PCI bus. This feature enables the system to run in whatever byte-ordering configuration makes most sense, regardless of the use of PCI peripheral devices.

The PCI subsystem code in Linux is organized in three tiers: board, architecture, and Linux kernel. At the lowest tier is the board level PCI initialization that resides in the directory "/arch/mips/rc32438/79EB438/". The middle tier, which is architecture-specific, consists of a few files located in the directory "/arch/mips/kernel/" which implement MIPS architecture-specific functions for the PCI subsystem. The highest tier, located in the directory "/drivers/pci/", is architecture-independent and is part of the Linux driver source code tree.

Board Level PCI Functionality

The PCI subsystem in the RC32438 provides four address-space windows, each of which can be programmed to be either PCI memory space or PCI I/O space of different sizes. The hardware also provides the flexibility of address translation and byte swapping for all four PCI windows. For each PCI window, the RC32438 PCI subsystem makes available a set of three registers: control, base, and mapping.

Main Data Structures (/arch/mips/rc32438/79EB438/setup.c)

Summary

Four structures of the type 'resource' are used to manage PCI memory and I/O resources. The initialization code uses a memory resource starting at address 0x50000000 with a size of 16 Mbytes. The I/O resource starts at address 0x18800000 with a size of 1 Mbyte. It is recommended to increase the memory resource to 64 Mbytes as many PCI devices, such as video encoders, request memory in excess of 16 Mbytes.

Detailed Explanation

Linux definition of the resource data structure is listed below:

```
struct resource {
    const char *name;/* Name of the resource */
    unsigned long start, end;/* Start and ending address */
```

Notes

```
        unsigned long flags;/* Resource flags */
        struct resource *parent, *sibling, *child; /* tree-like structure */
};
```

The first PCI window is configured to be a resource with the 'memory flag' set. The beginning address of the resource is 0x50000000 and it ends at address 0x5FFFFFFF. This allows allocation of up to 256 Mbytes of memory. Currently, the IDT Linux BSP allocates only 32 Mbytes of this space. This is the parent resource, and the resource "rc32438_res_pci_mem2" is its child. The PCI window resources do not use nesting; therefore, defining parent, sibling, and child has no chip-specific consequence. The PCI memory windows are hard-coded to specific addresses.

Although this approach fulfills most of the requirements, a better method would be to use more easily accessible macros or defines to demarcate memory start and end.

```
struct resource rc32438_res_pci_mem1 = {
    "PCI Mem1",/* resource name */
    0x50000000,/* resource beginning address */
    0x5FFFFFFF,/* resource ending address */
    IORESOURCE_MEM,/* resource flag: memory window */
    &rc32438_res_pci_mem1, /* Parent: root */
    NULL,/* No siblings */
    &rc32438_res_pci_mem2/* child resource */
};
```

The second resource is configured to be a memory window with 256 Mbytes of address range starting from address 0x60000000 and ending at address 0x6FFFFFFF.

```
struct resource rc32438_res_pci_mem2 = {
    "PCI Mem2",/* resource name */
    0x60000000,/* resource beginning address */
    0x6FFFFFFF,/* resource ending address */
    IORESOURCE_MEM,/* resource flag: memory window */
    &rc32438_res_pci_mem1, /* Parent: root */
    NULL,/* No siblings */
    NULL/* No children */
};
```

The third resource is configured to be a memory window with 4 Mbytes of address range starting from address 0x18C00000 and ending at address 0x18FFFFFF.

```
struct resource rc32438_res_pci_mem3 = {
    "PCI Mem3",/* resource name */
    0x18C00000,/* resource beginning address */
    0x18FFFFFF,/* resource ending address */
    IORESOURCE_MEM,/* resource flag: memory window */
    &rc32438_res_pci_mem1, /* Parent: root */
```

Notes

```
        NULL,/* No siblings */
        NULL/* No children */
    };
```

The fourth resource is configured to be an I/O window with 1 Mbyte of address range starting at address 0x18800000 and ending at address 0x188FFFFF. */

```
    struct resource rc32438_res_pci_io1 = {
        PCI I/O1",/* resource name */
        0x18800000,/* resource beginning address */
        0x188FFFFF,/* resource ending address */
        IORESOURCE_IO/* resource flag: IO window */
    };
```

Note that the current Linux implementation for the 79EB438 evaluation board utilizes only "rc32438_res_pci_mem1" and "rc32438_res_pci_io1" resources.

Code Execution Sequence:

```
    /* Initialize IO and memory resources to be used by PCI subsystem for allocating resources to the PCI devices found on the PCI bus */
    ioport_resource.start = rc32438_res_pci_io1.start;
    ioport_resource.end = rc32438_res_pci_io1.end;
    iomem_resource.start = rc32438_res_pci_mem1.start;
    iomem_resource.end = rc32438_res_pci_mem2.end;

     /* Disable IP bus error for PCI scanning. For vacant PCI slots, the PCI subsystem does not return any value. If the IP Bus error is not disabled, the scanning will not go through correctly */
    pciCntlVal=rc32438_pci->pcic;/* Read the PCI control register */
    pciCntlVal &= 0xFFFFFF7;/* Disable IP bus error: bit 7(0x80) */
    rc32438_pci->pcic = pciCntlVal;/* Write the PCI control register */

    /* Initialize the PCI bridge */
    rc32438_pcibridge_init();
```

PCI Bridge Initialization (/arch/mips/rc32438/79EB438/pci_ops.c)

Summary

The PCI bridge initialization implements operations to initialize local PCI registers and the PCI configuration registers of the host bridge. At the end of initialization, the program resets the target not ready bit in the PCI control register. This permits external PCI devices to access the local resources of the host processor across PCI. Note that the BAR registers for the host are populated by this function.

Notes

The Linux Kernel PCI “scan and allocate resources” function allocates resources to all other PCI devices on the system. Also, the BAR registers for the host are allocated memory addresses as follows:

- *0x00000000 (128 MB memory: maps DDR memory resident on the host for usage by PCI devices on the system)*
- *0x18800000 (1MB, I/O)*
- *0x18000000 (2MB I/O to allow external PCI devices to access the host registers)*
- *0x48000000 (dummy value, BAR disabled).*

For PCI accesses from the local side of the host, four PCI windows are allocated base addresses 0x50000000 (16MB, memory), 0x60000000 (256MB, memory), 0x18C00000 (4MB, I/O), and 0x18900000 (1MB, I/O).

Detailed Explanation

At the beginning of the PCI bridge initialization function, the PCI mode is checked. Initialization continues only if the PCI is configured to be in the Host mode.

```
 /* Read PCI control register */
pcicValue = rc32438_pci->pcic;


/* Isolate the PCI Mode by shifting control 6 bits to right */
pcicValue = (pcicValue >> PCIM_SHFT) & PCIM_BIT_LEN;


/* If the PCI is not configured in host mode then return. */
if (!((pcicValue == PCIM_H_EA) ||
 (pcicValue == PCIM_H_IA_FIX) ||
(pcicValue == PCIM_H_IA_RR))) {
/* Not in Host Mode, return ERROR */
return;
}
```

Once the PCI mode of the host device is confirmed, PCI bridge initialization proceeds. The PCI control register is modified to enable the PCI interface, and the arbiter is configured for dynamic idle grant mode and arbiter parking enabled.

```
 /* Enables the Idle Grant mode, Arbiter Parking */
 pcicData |=(PCIC_igm_m|PCIC_eap_m|PCIC_en_m);
 rc32438_pci->pcic = pcicData; /* Enables PCI bus Interface */
```

Before proceeding further, the code resets all the local PCI registers to quiescent state as shown below.

```
/* Clear the status register and mask all corresponding interrupts */
rc32438_pci->pcis = 0;
rc32438_pci->pcism = 0xFFFFFFFF;


/* Zero out the PCI decoupled registers and mask interrupts */
rc32438_pci->pcidac=0;
rc32438_pci->pcidas=0; /* clear the status */
```

Notes

```
rc32438_pci->pcidasm=0x0000007F; /* Mask all the interrupts */

/* Clear the control registers and mask PCI Messaging Interrupts */
rc32438_pci_msg->pciiic = 0;
rc32438_pci_msg->pciiim = 0x7;
rc32438_pci_msg->pciioic = 0;
rc32438_pci_msg->pciioim = 0x7;
```

The next step allocates the local PCI translation registers for accessing PCI address space as illustrated below. The swap bit in the PCI local base control register is set for big endian systems.

```
/* Setup PCILB0 as Memory Window starting at 0x50000000 */
rc32438_pci->pcilba[0].a = (unsigned int) (PCI_ADDR_START);

/* Address placed on the PCI bus is the same as on the Host local bus*/
rc32438_pci->pcilba[0].m = (unsigned int) (PCI_ADDR_START);

/* Setup PCILBA1 as Memory window of size 16 Mbytes */
rc32438_pci->pcilba[0].c = ( ((SIZE_16MB & 0x1f) <<
PCILBAC_size_b));

For big endian systems, the control register enables the swap bit as shown below:
rc32438_pci->pcilba[0].c = ( ((SIZE_16MB & 0x1f) << PCILBAC_size_b)
  | PCILBAC_sb_m);
```

In similar fashion, the other PCI local base registers for the little endian mode on the 79EB438 board are programmed as shown below. For a big endian system, the swap bit needs to be initialized in the control register.

```
/* Local Base Register Space 1: memory with size of 256 Mbytes */
rc32438_pci->pcilba[1].a = 0x60000000;
rc32438_pci->pcilba[1].m = 0x60000000;
rc32438_pci->pcilba[1].c = ((SIZE_256MB & 0x1f) <<
PCILBAC_size_b);

/* Local Base Register Space 2: IO with 4 Mbytes range */
rc32438_pci->pcilba[2].a = 0x18C00000;
rc32438_pci->pcilba[2].m = 0x18FFFFFF;
rc32438_pci->pcilba[2].c = ((SIZE_4MB & 0x1f) << PCILBAC_size_b);

/* Local Base Register Space 3: IO with 1 Mbytes range */
rc32438_pci->pcilba[3].a = 0x18900000;
```

Notes

```
rc32438_pci->pcilba[3].c = (((SIZE_1MB & 0x1ff) << PCILBAC_size_b)
| PCILBAC_msi_m);
```

After completing the PCI host local register initialization, this function programs the PCI configuration registers for the host, including the BAR registers and the corresponding control/mapping register extensions to the PCI configuration address space.

The BAR0 is assigned the address 0x00000000, which is a memory-based I/O. The control register (0x44) for BAR0 is programmed for 128 Mbytes in size. BAR0 is made "prefetchable" with appropriate bit settings to convert memory read transactions to memory read multiple transactions. The mapping register (0x48) for BAR0 is assigned a value of 0x0, similar to the BAR0 register.

BAR1 is assigned the address 0x18800000, which is an I/O window. The size of BAR1 is programmed to be 1 Mbyte in the corresponding control register (0x4C). The BAR1 mapping register (0x50) is set to value 0x0. This results in translating all PCI I/O transactions with an address such as 0x188abcde for the host to an address 0x000abcde for the local bus.

The BAR2 register is programmed as an I/O with the address 0x18000000. The control register (0x54) sets the size to 2 Mbytes and the mapping register (0x58) carries value 0x18000000, similar to the BAR2 register. In other words, no address translation is performed on BAR2 PCI addresses. This window allows external agents on the PCI bus to access the CPU local registers.

The BAR4 register is assigned a dummy value of 0x48000000 and the control register (0x5C) disables the BAR3 register.

In the final step, the program resets the 'target not ready' bit in the PCI control register.

```
/* Read PCI control register value */
pciCntlVal=rc32438_pci->pcic;

/* Reset PCI target not ready bit */
pciCntlVal &=~(PCIC_tnr_m);

/* Write the updated value to the PCI control register */
rc32438_pci->pcic = pciCntlVal;
```

PCI Configuration Function

The PCI configuration functions are used by the Linux kernel PCI subsystem. On power-up, the kernel PCI subsystem scans all the PCI devices available on the PCI bus. It performs configuration read cycles of the Vendor-Device ID PCI register for all the possible PCI slots. For the slots with valid vendor-device ID, a PCI device data structure is created by the kernel. Subsequently, the PCI devices are allocated resources and enabled by initializing their PCI configuration address space registers.

The BSP level PCI functionality provides Double-Word (32 bit)/Word (16 bit)/Byte read and write functions for PCI configuration space accesses. These functions are defined in file "/arch/mips/rc32438/79EB438/pci_ops.c" and are illustrated below.

Notes

The "config_access" is the primary underlying function that executes all the PCI configuration reads and writes. It is invoked by "config_write" and "config_read" functions. It accepts five parameters:

- *The first parameter, 'type', specifies whether it is PCI read or write operation.*
- *The second parameter, 'bus', specifies the PCI bus number. For a primary PCI bus, the value is 0; for a secondary bus, the value must fall in the region "0<bus<256".*
- *The Third parameter, 'devfn', provides the device and function address/number whose PCI configuration registers need to be accessed.*
- *The fourth parameter, 'where', specifies the register in the PCI configuration space that needs to be accessed.*
- *The fifth and final parameter, 'data', is the placeholder for the data read/written to the PCI configuration registers.*

Macros used by the function are:

PCI_SLOT:

```
/* Provides the PCI slot number */
#define PCI_SLOT(devfn)(((devfn) >> 3) & 0x1f)
```

PCI_FUNC:

```
/* Provides the PCI function number */
#define PCI_FUNC(devfn)((devfn) & 0x07)
```

PCI_CFG_SET:

/* Generate PCI configuration register address for primary PCI bus. The parameter "off" is the register offset in the PCI configuration space. The generated address is written to PCI configuration address register (0xB800_000C). Subsequent reads and writes to PCI configuration data register (0xB800_0010) complete the read or write cycle. */

```
#define PCI_CFG_SET(slot,func,off) \
(rc32438_pci->pcicfga = (0x80000000 | ((slot)<<11) | \
((func)<<8) | (off)))
```

The code listing for the function "config_access" is as shown below:

```
static int
config_access(u8 type, u8 bus, u8 devfn, u8 where, u32 *data)
{
/* Find the PCI device slot */
u8 slot = PCI_SLOT(devfn);

/* Find the PCI function number */
u8 func = PCI_FUNC(devfn);

/* 79EB438 board has only 4 slots numbered 2-5 */
if (bus != 0 || slot > 5) {
*data = 0xFFFFFFFF;
return PCIBIOS_DEVICE_NOT_FOUND;
```

Notes

```
}

/* Find the address of the configuration address space */
PCI_CFG_SET(slot, func, where);

/* Find the PCI access cycle and read/write data accordingly. */
if (type == PCI_ACCESS_WRITE)
      rc32438_pci->pcicfgd = *data;

else
*data = rc32438_pci->pcicfgd;

/* Checks for target 'master abort' not implemented since it may take a comparatively long time
(CPU clock speed is much more than the PCI bus speed) for the status to update.*/

return 0;
}
```

The "config_access" function is used to implement "config_write" and "config_read" functions.

```
static inline int config_write(u8 bus, u8 devfn, u8 where, u32 data)
{
return config_access(PCI_ACCESS_WRITE, bus, devfn, where, &data);
}

static inline int config_read(u8 bus, u8 devfn, u8 where, u32 *data)
{
return config_access(PCI_ACCESS_READ, bus, devfn, where, data);
}
```

PCI BIOS Initialization Sequence

The kernel "pci_init" function ("/drivers/pci/pci.c") call invokes the services of architecture and board-specific PCI functionality by making a call to "pcibios_init" function in file "/arch/mips/kernel/pci.c". Two important tasks of PCI configuration are completed in "pcibios_init". The first task is to scan for all PCI devices on the bus, configure them, and allocate memory resources. Board-specific fix-up, such as IRQ assignment, is performed as part of the second task. Detailed illustrations of these steps along with the code implementation are discussed below.

Overview:

The Linux kernel uses a "pci_channel" data structure to define a PCI bus. The IDT 79EB438 board uses only one host and therefore has only one such define.

```
struct pci_channel {
```

Notes

```
struct pci_ops *pci_ops;
struct resource *io_resource;
    struct resource *mem_resource;
    int first_devfn;
    int last_devfn;
    };
```

The corresponding definition of the data structure defined in the BSP is

```
    struct pci_channel mips_pci_channels[] = {
    {
    &rc32438_pci_ops,  /* table of config functions */
    &rc32438_res_pci_io1, /* IO resource */
    &rc32438_res_pci_mem1, /* Memory resource */
    PCI_DEVFN(1,0),  /* PCI Slot 2 */
    PCI_DEVFN(6,0)   /* PCI Slot 5 */
    },
    { NULL, NULL, NULL, 0, 0} /* Null member to mark the end */
    };
```

The data structure "pci_ops" is used to store pointers to configuration read/write functions.

```
    struct pci_ops {
       int (*read_byte)(struct pci_dev *, int where, u8 *val);
       int (*read_word)(struct pci_dev *, int where, u16 *val);
       int (*read_dword)(struct pci_dev *, int where, u32 *val);
       int (*write_byte)(struct pci_dev *, int where, u8 val);
       int (*write_word)(struct pci_dev *, int where, u16 val);
       int (*write_dword)(struct pci_dev *, int where, u32 val);
    };
```

The corresponding data structure definition in the BSP is

```
    static struct pci_ops rc32438_pci_ops = {
       read_config_byte,
       read_config_word,
       read_config_dword,
       write_config_byte,
       write_config_word,
       write_config_dword
    };
```

The resources to the bus are allocated in a loop:

Notes

```
/* assign resources */
busno=0;
for (p= mips_pci_channels; p->pci_ops != NULL; p++) {
busno = pciauto_assign_resources(busno, p) + 1;
}
```

The leaf function "pciauto_bus_scan" allocates resources to the PCI devices as illustrated in the code commentary below.

```
/* Check all six BAR registers of the PCI device */
for (bar = PCI_BASE_ADDRESS_0; bar <= PCI_BASE_ADDRESS_5; bar+=4) {
/* Write all 1's to the BAR register */
early_write_config_dword(hose, top_bus,
current_bus,
pci_devfn,
bar,
0xffffffff);

/* Read back to determine the memory size implemented */
early_read_config_dword(hose, top_bus,
current_bus,
pci_devfn,
bar,
&bar_response);

/* If BAR is not implemented go to the next BAR */
if (!bar_response)
continue;

 /* Check the BAR type and set the address mask */
if (bar_response & PCI_BASE_ADDRESS_SPACE) {
/* IO space requested by the PCI device */
addr_mask = PCI_BASE_ADDRESS_IO_MASK;
upper_limit = &pciauto_upper_iospc;
lower_limit = &pciauto_lower_iospc;

} else {
/* Memory space requested by the PCI device */
addr_mask = PCI_BASE_ADDRESS_MEM_MASK;
upper_limit = &pciauto_upper_memspc;
```

Notes

```
lower_limit = &pciauto_lower_memspc;
}

/* Calculate requested size of the BAR */
bar_size = ~(bar_response & addr_mask) + 1;

/* Allocate base address after proper alignment */
bar_value = ((*lower_limit - 1) & ~(bar_size - 1)) + bar_size;

/* Check if requested memory is available, if not retry */
if ((bar_value + bar_size) > *upper_limit) {
if (bar_response & PCI_BASE_ADDRESS_SPACE) {
if (io_resource_inuse->child) {
io_resource_inuse =
io_resource_inuse->child;
pciauto_lower_iospc =
io_resource_inuse->start;
pciauto_upper_iospc =
io_resource_inuse->end + 1;
}

} else {
if (mem_resource_inuse->child) {
mem_resource_inuse =
mem_resource_inuse->child;
pciauto_lower_memspc =
mem_resource_inuse->start;
pciauto_upper_memspc =
mem_resource_inuse->end + 1;
}
}
DBG(" unavailable -- skipping, value %x size %x\n",
bar_value, bar_size);
continue;
}

/* Write the value to the BAR register and update the lower limit */
early_write_config_dword(hose, top_bus, current_bus, pci_devfn,
 bar, bar_value);
```

Notes

```
        *lower_limit = bar_value + bar_size;
        }
```

During execution of the "pcibios_init", the software makes a call to "pcibios_fixup_irqs" to do board specific interrupt allocation.

```
    void __init pcibios_fixup_irqs(void)
    {
        struct pci_dev *dev;

        pci_for_each_dev(dev) {
        unsigned int slot;

        /* Bridges should not be assigned IRQ numbers */
        if (dev->bus->number != 0) {
        return;
        }

        /* Find the PCI slot number and read interrupt pin */
        slot = PCI_SLOT(dev->devfn);
        if (slot > 0 && slot <= 5) {
        unsigned char pin;
        pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
        /* In 79EB438, all PCI interrupts are tied to the GPIO pin */
        switch (pin) {
        case 1: /* INTA*/
        dev->irq = GROUP4_IRQ_BASE + 27;
        break;
        case 2: /* INTB */
        dev->irq = GROUP4_IRQ_BASE + 27;
        break;
        case 3: /* INTC */
        dev->irq = GROUP4_IRQ_BASE + 27;
        break;
        case 4: /* INTD */
        dev->irq = GROUP4_IRQ_BASE + 27;
        break;
        default:
```

Notes

```
dev->irq = 0xff;
break;
}
/* Update the PCI Interrupt line field */
pci_write_config_byte(dev, PCI_INTERRUPT_LINE,
    dev->irq);
}
}
}
```

Conclusion

This application note focused primarily on the board specific initialization sequence. The application note described initialization of the PCI bridge, scanning and allocation of resources to the PCI devices, and board-specific initialization such as interrupts. Being board-specific and allowing rich alternate possibilities, this area of the PCI bios code should greatly benefit the system programmer. The information contained in this application note should also help in porting the Linux operating system to new board designs based on RC32438 integrated communications processor.

References

79RC32438 User Reference Manual (IDT, Inc.)

Online Linux Source code: http://lxr.linux.no/source/

IDT-Linux Source Code for 79EB438 Evaluation Board (May be obtained by sending an email to rischelp@idt.com.)

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.