FRONTGRADE

APPLICATION NOTE

UT700

Enable the Interrupt Controller Module
UT700 LEON 3FT

9/29/2017
Version #: 1.0.1

Table 1: Cross Reference of Applicable Products

Product Name	Manufacturer Part Number	SMD #	Device Type	Internal PIC Number
UT700 LEON	UT700	5962-13238	Interrupt Controller Module	WQ03

1.0 Overview

The UT700 LEON 3FT SPARC™ processor consists of multiple embedded modules that support a wide range of applications. Each module has embedded control logic allowing the modules to function independently without the microprocessor intervention. This spatial paradigm improves the throughput of the UT700 by enabling all the modules to operate in concurrency.

This operational paradigm for the embedded modules posed a different challenge to data transactions between the microprocessor and the modules; module software drivers need to know when data is available for retrieving. Data retrieval through software synchronous access method is adding another layer of complexity to the application program. This software synchronous access method employs software polling methods that result in a nondeterministic software response time and reduces the throughput of the system. To overcome this data retrieval problem, the UT700 provides an asynchronous data availability notification method that eliminates the needs to poll for data availability while providing a deterministic response time to invoke the module software drivers.

This asynchronous data availability notification method is made possible with the use of the UT700's Interrupt Controller that provides interrupts to invoke the software drivers for data retrieval. The UT700's Interrupt Controller provides 15 interrupt services and an extended interrupt service to facilitate more interrupt services to the UT700. We explore the Interrupt Controller, its features and how to configure it to perform tasks.

Note: The description in this application note describes how to directly use the memory mapped interface of a specific hardware peripheral. If you are using an operating system such as RTEMS, Linux, and VxWorks or an environment such as BCC then it is recommended to use the infrastructure provided by those environments instead of accessing the peripheral directly as described in this application note.

2.0 Application Note Layout

This application note (AN) starts by providing a brief description of the Interrupt Controller memory map, the associated registers, the 15 interrupts and the extended interrupts services. This description of the Interrupt Controller falls under the Interrupt Controller Hardware sections.

After the Interrupt Controller Hardware sections, this AN provides a high-level flow diagram to depict the correct sequential steps to initialize the interrupt services. We will describe each block in the order as shown in the flow diagram. The description of the flow diagram falls under the Interrupt Controller Initialization sections.

Finally, we can apply this knowledge using C programming code to enable the UT700 Interrupt Controller to provide asynchronous data availability notification to the system and invokes the module device drivers to performance data retrieval. The C code programming examples fall under the Interrupt Controller Programming sections.

These subsections are described in detail below:

- Interrupt Controller Hardware
- Interrupt Controller Initialization
- Interrupt Controller Programming

3.0 Interrupt Controller Hardware

The Interrupt Controller provides a way for an asynchronous notification to the system. It has seven registers, and these registers are mapped in the peripheral's memory region from 0x8000_0200 to 0x8000_02C0. These interrupts have two level setting options; a lower priority with a 0 level setting (Also the default setting) and a higher priority with a 1 level setting. Besides the interrupt level setting options, the interrupt number also shows the priority of that interrupt; the higher the interrupt number, the higher is the priority.

Table 2: Interrupt Priority Settings

Interrupt Number	Level Setting	
	Case 1	Case 2
15	0	0
14	0	0
13	0	0
3	0	1
2	0	1
1	0	1

Table 2 shows the 6 interrupts with different level setting options. In both cases, if all the interrupts happened at the same time, this is how the Interrupt Controller decodes them as shown **in Table 3.** Interrupt number 9 is also used as an extended interrupt, a feature to increase the number of interrupts in the system while remaining backward compatible with the older devices.

Table 3: Interrupt Decoding Order

Decodes Order	Interrupt Number	
	Case 1	Case 2
1	15	3
2	14	2
3	13	1
4	3	15
5	2	14
6	1	13

For more details about the Interrupt Controller, see the UT700 Function Manual **Chapter 5.**

4.0 Interrupt Controller Initialization

Interrupt Controller Initialization is a precise task. If the initialization is performed in the incorrect order, the desire interrupt service routine (ISR) will not function or the system will crash; debugging a non-functional ISR is an intensive laboring job, no debugger can assist in this process. Therefore, knowing the correct procedural steps to initialize the interrupt controller helps get the desired ISR working and ready for additional software flow control required by the ISR to perform its task.

Figure 1 shows the procedural steps to initialize the interrupt controller. We will describe each step in the flow diagram in the following sections.

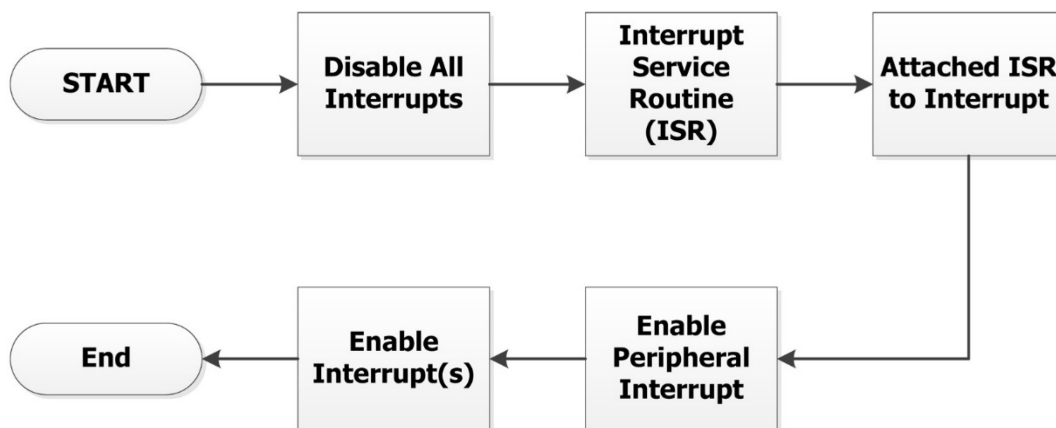

Figure 1: Interrupt Controller Initialization Steps

4.1 Disable All Interrupts

On power on reset, the Interrupt Mask Register is clear; the interrupts are masked (Disabled). Nonetheless, it is a good practice to clear the interrupt level register (ILR), clear the interrupt mask register (IMR) and set the interrupt clear register (ICR).

4.2 Interrupt Service Routine (ISR)

The ISR for Bare C Compiler (BCC) takes the form as shown in **Code 1.** This ISR function format applies to both ISR and extended ISR. The parameter “irq” is the interrupt number 1 to 15.

```
void moduleName_irqHandler(int irq)
{
    /* add your code below */
}
```

Code 1: ISR Function Format

For the extended ISR, we need to add function calls within the extended ISR to invoke the respective device drivers, see **Code 2.**

```
void extended_irqHandler(int irq)
{
    /* add your code below */

    /* read the Extended Interrupt Acknowledge Register */

    /* call the respective device driver */
}
```

Code 2: Extended ISR Function Format

We will describe the ISR and extended ISR in detail in Interrupt Controller Programming sections.

4.3 Attached ISR to Interrupt

This process associates the ISR to the appropriate interrupt. In BCC, we use the following function as shown in **Code 3.** Once the ISR is attached to a particular interrupt, this ISR will only service that interrupt.

```
catch_interrupt(func* moduleName_irqHandler, int irq);
```

Code 3: Catch Interrupt Function

4.4 Enable Peripheral Interrupt

This process enables peripheral device interrupt associated to that Interrupt number. Each peripheral device has its own set of registers to support interrupt; refer to the UT700 Functional Manual, **Chapter 5** and the module chapter, for the more information.

4.5 Enable Interrupt

Finally, we enable the Interrupt mask register (IMR) to enable the interrupt that we have associated to the ISR. This completes the Interrupt Controller Initialization process.

Section 5.0 shows how these steps are done using C programming language. We also show how to use the Interrupt Force Register to invoke an ISR; a feature that is useful for testing ISR.

5.0 Interrupt Controller Programming

We learned from **Section 4.0** how to enable the Interrupt Controller, how the interrupts and extended interrupts work, how to create interrupt routines and how to attach the interrupt routines.

In the following sections, we will provide programming examples to initialize the Interrupt Controller and use the Interrupt Force register to test ISR.

5.1 Disable All Interrupts

These are the codes to disable all interrupts. It is also a good practice to write a zero to any reserved pin to ensure backward compatibility.

```
INTCTRL.ILR.R    = 0;            // Interrupt level register
INTCTRL.ICR.R    = 0x0001_FFFE;  // Interrupt clear register
INTCTRL.IMR.R    = 0;            // Interrupt mask register
```

Code 4: IRQ Function

5.2 Interrupt Service Routine (ISR)

Both the ISR **(Code 5)** and extended ISR **(Code 6)** take on the form as shown. The only difference besides the software flow control for different modules is the extended ISR is required to parse for the extended interrupt number and invoke their interrupt handler respectively. Appended are the examples:

```
void moduleName_irqHandler(int irq)
{
    /* add your code below */

    /* read module data and store it in buffer */
    /* refer to UT700 Functional Manual for the module registers */

    /* return */
}
```

Code 5: IRQ Function

The extended IRQ handler provides entry points to invoke ISRs for additional modules as shown in **Code 6.**

```
void extended_irqHandler(int irq)
{
    /* add your code below */
    uint32_t EID;

    /* read the Extended Interrupt Acknowledge Register */
    EID = INTCTRL.EIAR.B.EID;

    /* call the respective device driver */
    switch(EID)
    {
        case SPI:
                SPI_irqHandler();       // call SPI ISR
            break;

        case BC1553:
                BC1553_irqHandler();    // call 1553 ISR
            break;

        default:
            break;
    }
}
```

Code 6: Extended IRQ Function

5.3 Attached ISR to Interrupt

Attaching ISR to the interrupt is similar for both ISR and extended ISR as shown in **Code 7** and **Code 8** respectively.

```
catch_interrupt(moduleName_irqHandler, irq);
```

Code 7: Attach IRQ to Interrupt

```
catch_interrupt(extended_irqHandler, irq);
```

Code 8: Attach Extended IRQ to Interrupt

5.4 Enable Peripheral Interrupt

Refer to the UT700 Functional Manual, the chapter of that module you are using and how to initialize the module registers.

5.5 Enable Interrupt Mask

It is proper to enable the Interrupt Mask **(Code 9)** last to prevent spurious interrupt. For example, if we swap the initialization procedure in Sections **5.4** and **5.5**, the spurious interrupt might happen while we are enabling the peripheral interrupt.

```
INTCTRL.IMR.R = 1 << irq;       // Interrupt mask register
```

Code 9: Enable Interrupt Mask

5.6 Manually Trigger Interrupt

The interrupt controller provides an Interrupt Force Register to allow users to test if their ISR is correctly attached to the interrupt controller. If we follow the instruction from **5.1** to **5.5** with an IRQ number 2, then by setting the Interrupt Force Register bit 2 **(Code 10)** will invoke the IRQ handler.

```
INTCTRL.IFR.R   = 1 << irq;     // Interrupt force register
```

Code 10: Manually Trigger Interrupt

6.0 Summary and Conclusion

After going through this AN, the reader should know how to enable the Interrupt Controller, how interrupts and extended interrupts work, how to create interrupt routines, how to attach the interrupt routines and how to test the interrupt routines using the Interrupt Force register.

For more information about our UT700 LEON 3FT/SPARC™ V8 Microprocessor and other products please visit our website, www.frontgrade.com or email us at https://frontgrade.com/contact-us.

Appendix A: Header File

This header file is designed for this application note purpose only.

```
/***************************************************************************\
* MODULE: Interrupt Controller (INTCTRL)
\***************************************************************************/
#define INTCTRL_ADDR 0x80000200
struct INTCTRL_TAG {                          // Table 5.2: IRQ Controller Register
        union {
                vuint32_t R;                  // Interrupt level register (ILR)
0x80000200
                struct {
                  vuint32_t RES31_16:16;      // Reserved
                  vuint32_t IL :15;           // Interrupt level for interrupt IL[n]
                  vuint32_t RES:1;            // Reserved
                } B;
        } ILR;
        union {
                  vuint32_t R;                // Interrupt pending register (IPR)
0x80000204
                  struct {
                  vuint32_t RES31_16:16;      // Reserved
                  vuint32_t IP :15;           // Interrupt pending for interrupt IP[n]
                  vuint32_t RES:1;            // Reserved
                } B;
        } IPR;
        union {
                  vuint32_t R;                // Interrupt force register (IFR)
0x80000208
                  struct {
                  vuint32_t RES31_16:16;      // Reserved
                  vuint32_t IF :15;           // Force interrupt IF[n]
                  vuint32_t RES:1;            // Reserved
                } B;
        } IFR;
        union {
                  vuint32_t R;                 // Interrupt clear register (ICR)
0x8000020C
                  struct {
                  vuint32_t RES31_16:16;      // Reserved
                  vuint32_t IC :15;           // Clear interrupt IC[n]
                  vuint32_t RES:1;            // Reserved
                } B;
        } ICR;
        union {
                  vuint32_t R;                // Interrupt status register (ISR)
0x80000210
                  struct {
                  vuint32_t NCPU:4;           // Number of CPUs in the system -1
                  vuint32_t BA:1;             // Broadcast Available - set to 1 if
NCPU > 0
                  vuint32_t RES26_20:7;       // Reserved
```

```
                  vuint32_t EIRQ:4;            // Extended Interrupts 1-15
                  vuint32_t STATUS:16;         // Power-down status of CPU[n]
                } B;
} ISR;
        vuint32_t padding[11];                 // Padding

        union {
                  vuint32_t R;                 // Interrupt mask register (IMR)
0x80000240
                  struct {
                  vuint32_t RES31_16:16;       // Reserved
                  vuint32_t IM :15;            // Interrupt mask for IM[n]
                  vuint32_t RES:1;             // Reserved
                } B;
        } IMR;

        vuint32_t padding[31];                 // Padding

        union {
                  vuint32_t R;                 // Extended Interrupt acknowledge
register (EIAR) 0x800002C0
                  struct {
                  vuint32_t RES31_5:27;        // Reserved
                  vuint32_t EID :5;            // Extended interrupt ID (EID)
                } B;
        } EIAR;
};
```

Revision History

Date	Revision #	Author	Change Description	Page #
03/03/2017	1.0.0	MTS	Initial Release	
09/29/2017	1.0.1	MTS	Add note	1

Frontgrade Technologies Proprietary Information Frontgrade Technologies (Frontgrade or Company) reserves the right to make changes to any products and services described herein at any time without notice. Consult a Frontgrade sales representative to verify that the information contained herein is current before using the product described herein. Frontgrade does not assume any responsibility or liability arising out of the application or use of any product or service described herein, except as expressly agreed to in writing by the Company; nor does the purchase, lease, or use of a product or service convey a license to any patents, rights, copyrights, trademark rights, or any other intellectual property rights of the Company or any third party.