Majenko Technologies

Working with chipKIT^{TM}Interrupts

AN1132 Revision 3

How to write software to interact with the PIC32^{TM}interrupt system using the chipKIT^{TM}core

Contents

1 Introduction **2**
- 1.1 Priorities . . . 2
- 1.2 Vectors and IRQs . . . 2

2 Controlling Interrupts: Enabling and Disabling **3**

3 Configuring An Interrupt **4**

4 Writing an ISR **5**
- 4.1 Global Variables . . . 5

5 Bringing It All Together **6**

A Common Vector Names **7**
- A.1 PIC32MX . . . 7
- A.2 PIC32MZ . . . 7

B Interrupt Request Names **9**
- B.1 PIC32MX . . . 9
- B.2 PIC32MZ . . . 10

C Document Revisions **12**

1 Introduction

The chipKIT^{TM}programming environment API provides a complete abstraction layer around the internal PIC32^{TM}interrupt subsystem. The functions provided allow the complete manipulation of the interrupts at runtime without burdening the user with the requirement of low-level knowledge of the inner workings of the interrupts.

The interrupt vector table is relocated into RAM instead of being in Flash. This allows the entries to be manipulated at runtime instead of being set at compile time.

When working with interrupts it is important to remember some key facts about how they work.

1.1 Priorities

Interrupts have different priorities. They are numbered from 0 through 7 and higher numbered priority interrupts take precedence over lower numbered interrupts. The priority used for any one interrupt is entirely the user's choice and there are no hard and fast rules defining what priorities should be assigned to which interrupts. The only caveat is that interrupt priority 0 means the interrupt is disabled.

If two interrupts share the same priority there is the option of a sub-priority. These sub-priorities are numbered 0 through 3 with again the higher sub-priority taking precedence over a lower sub-priority.

1.2 Vectors and IRQs

On the PIC32MX series of chips there are both interrupt vectors and interrupt request (IRQ) numbers. There isn't a direct 1:1 mapping between them, and it is important to remember when a vector is needed and when an IRQ is needed. The vector is the location in the vector table, and the IRQ is the logical number of the interrupt. Some different IRQ numbers point to the same vector, so it is important to ensure that your interrupt handlers are able to cope with that.

2 Controlling Interrupts: Enabling and Disabling

The initial configuration of the interrupt system is a fairly complex process, but the good news is that all of the configuration is done for you in the chipKIT^{TM}core. In fact, on the chipKIT^{TM}platform, interrupts are enabled by default when a board is powered on. Not only that, but the system is flexible enough that you can also enable and disable the whole interrupt system manually for those times when timing is absolutely critical or interrupting an operation is not feasible. We will explain how you can do this below.

To disable all interrupts, call the `disableInterrupts()` function. To enable all interrupts, call the `enableInterrupts()` function. Note that enabling and disabling happens immediately after the function call.

Note also that these two functions return a single `uint32_t` value that can be used with the `restoreInterrupts()` function. This returned value is called the *status register* and it is useful because it stores what the state of the interrupt system was - whether it was enabled or disabled - just prior to the `disableInterrupts()` or `enableInterrupts()` call. This is valuable information when you want to return to the state you were in prior to the function call. By simply calling the `restoreInterrupts(value)` function with the `uint32_t` value that was returned from your `disableInterrupts()` or `enableInterrupts()` call, you can return to the previous state, regardless of what it was. In this manner, different nested routines can handle enabling and disabling of interrupts without interfering with each other.

With all that said, using `restoreInterrupts()` as opposed to `enableInterrupts()` makes for more robust code.

```
uint32_t status = disableInterrupts();
// Run some sensitive code
restoreInterrupts(status);
```

Example 1: Disabling then re-enabling interrupts

3 Configuring An Interrupt

There are four basic steps to configuring an individual interrupt:

1. Connect the vector to the interrupt routine
2. Set the interrupt priorities
3. Clear any pending interrupt flag
4. Enable the interrupt

The first step, connecting the vector to the interrupt routine, is done using the `setIntVector(vector, function)` function. This function takes two parameters. The first is the vector number, the second is the name of the function to call for any interrupts associated with that vector. All the interrupt vectors have names defined for them, so you don't need to remember all the numbers. Refer to Appendix A on page 7 for a list of common vector names or how to find the right names for your target chip.

```
setIntVector(_TIMER_3_VECTOR, myISR);
```

Example 2: Connecting a vector to an ISR

Setting the priority and sub-priority of an interrupt vector are both done using the `setIntPriority(vector, priority, sub)` function. Note that this is performed on the vector, not the interrupt number.

```
setIntPriority(_TIMER_3_VECTOR, 4, 0);
```

Example 3: Setting the priority of an interrupt vector to 4, sub-priority 0

The clearing of any pending interrupt flag should always be done just before an interrupt is enabled. This is because interrupt flags can still be set even when interrupts are disabled and as soon as the interrupt is enabled any pending interrupt flags will be processed. So you should always use `clearIntFlag(irq)` and `setIntEnable(irq)` together.

```
clearIntFlag(_TIMER_3_IRQ);
setIntEnable(_TIMER_3_IRQ);
```

Example 4: Clearing the interrupt flag and enabling the interrupt

The `clearIntFlag()` function must also be called from within your ISR to inform the CPU that the interrupt has been handled; otherwise, the ISR will immediately be called again until the flag has been cleared.

4 Writing an ISR

The compiler utilizes a system of *attributes* to adjust how a function is called or compiled. Because the MX and MZ family of chips differ in how they handle the interrupts and attributes, a handy macro has been provided to craft the correct attributes for you: `__USER_ISR`

The basic form of an interrupt handler's definition is:

```
void __USER_ISR myISR()
```

It is important to remember to call `clearIntFlag()` in your interrupt routine. For example:

```
void __USER_ISR myISR() {
    // Perform your interrupt operations
    clearIntFlag(_TIMER_3_IRQ);
}
```

Example 5: Simple interrupt routine

4.1 Global Variables

Any variables used within your ISR but defined outside your ISR are considered global variables, and they require special consideration.

Interrupts, by nature, are unpredictable - triggering ISRs to run at any time. Since your global variable is used within the ISR, its value can also change at any time. However, your compiler will not know that your variable can change within the ISR unless you distinguish it in some way. The compiler has optimization routines that determine whether your variable is likely to change. If these routines don't "see" anything nearby that could modify your variable, they won't make the necessary space for your variable. In essence, they "optimize" it out.

To avoid this mishap, use the `volatile` keyword to tag your global variable. This keyword "tells" the compiler not to perform any optimization on the variable, as the variable can change at any time. In addition, the variable will always be referenced directly from RAM and never from a CPU register. As such, access to this variable will be somewhat slower, so use the `volatile` keyword only when absolutely necessary.

```
volatile uint32_t counter = 0;

void __USER_ISR myISR() {
    counter++;
    clearIntFlag(_TIMER_3_IRQ);
}
```

Example 6: Use of a volatile variable

5 Bringing It All Together

```
volatile uint32_t counter = 0;

void __USER_ISR myISR() {
    counter++;
    clearIntFlag(_TIMER_3_IRQ);
}

void setup() {
    // You would need to add code here to configure the timer 3

    setIntVector(_TIMER_3_VECTOR, myISR);
    setIntPriority(_TIMER_3_VECTOR, 4, 0);
    clearIntFlag(_TIMER_3_IRQ);
    setIntEnable(_TIMER_3_IRQ);

    Serial.begin(9600);
}

void loop() {
    delay(1000);
    Serial.print("Count is now: ");
    Serial.println(counter);
}
```

Example 7: Complete interrupt example

A Common Vector Names

A.1 PIC32MX

This list of vectors is taken from the header file for the PIC32MX795F512L chip. The chip you are using may not have the same list. To find the list of vector names for your specific chip you should refer to the header file for your chip. The header file is located within the pic32mx/include/proc folder within the pic32-tools compiler.

_CORE_TIMER_VECTOR
_CORE_SOFTWARE_0_VECTOR
_CORE_SOFTWARE_1_VECTOR
_EXTERNAL_0_VECTOR
_TIMER_1_VECTOR
_INPUT_CAPTURE_1_VECTOR
_OUTPUT_COMPARE_1_VECTOR
_EXTERNAL_1_VECTOR
_TIMER_2_VECTOR
_INPUT_CAPTURE_2_VECTOR
_OUTPUT_COMPARE_2_VECTOR
_EXTERNAL_2_VECTOR
_TIMER_3_VECTOR
_INPUT_CAPTURE_3_VECTOR
_OUTPUT_COMPARE_3_VECTOR
_EXTERNAL_3_VECTOR
_TIMER_4_VECTOR
_INPUT_CAPTURE_4_VECTOR
_OUTPUT_COMPARE_4_VECTOR
_EXTERNAL_4_VECTOR
_TIMER_5_VECTOR
_INPUT_CAPTURE_5_VECTOR
_OUTPUT_COMPARE_5_VECTOR
_SPI_1_VECTOR
_I2C_3_VECTOR
_I2C_1A_VECTOR
_SPI_3_VECTOR
_SPI_1A_VECTOR
_UART_1_VECTOR
_UART_1A_VECTOR
_I2C_1_VECTOR
_CHANGE_NOTICE_VECTOR
_ADC_VECTOR
_PMP_VECTOR
_COMPARATOR_1_VECTOR
_COMPARATOR_2_VECTOR
_I2C_4_VECTOR
_I2C_2A_VECTOR
_SPI_2_VECTOR
_SPI_2A_VECTOR
_UART_3_VECTOR
_UART_2A_VECTOR
_I2C_5_VECTOR
_I2C_3A_VECTOR
_SPI_4_VECTOR
_SPI_3A_VECTOR
_UART_2_VECTOR
_UART_3A_VECTOR
_I2C_2_VECTOR
_FAIL_SAFE_MONITOR_VECTOR
_RTCC_VECTOR
_DMA_0_VECTOR
_DMA_1_VECTOR
_DMA_2_VECTOR
_DMA_3_VECTOR
_DMA_4_VECTOR
_DMA_5_VECTOR
_DMA_6_VECTOR
_DMA_7_VECTOR
_USB_1_VECTOR
_CAN_1_VECTOR
_CAN_2_VECTOR
_ETH_VECTOR
_UART_4_VECTOR
_UART_1B_VECTOR
_UART_6_VECTOR
_UART_2B_VECTOR
_UART_5_VECTOR
_UART_3B_VECTOR
_FCE_VECTOR

A.2 PIC32MZ

This list of vectors is taken from the header file for the PIC32MZ2048ECG100 chip. The chip you are using may not have the same list. To find the list of vector names for your specific chip you should refer to the header file for your chip. The header file is located within the pic32mx/include/proc folder within the pic32-tools compiler.

_CORE_TIMER_VECTOR
_CORE_SOFTWARE_0_VECTOR
_CORE_SOFTWARE_1_VECTOR
_EXTERNAL_0_VECTOR
_TIMER_1_VECTOR
_INPUT_CAPTURE_1_ERROR_VECTOR
_INPUT_CAPTURE_1_VECTOR
_OUTPUT_COMPARE_1_VECTOR
_EXTERNAL_1_VECTOR
_TIMER_2_VECTOR
_INPUT_CAPTURE_2_ERROR_VECTOR
_INPUT_CAPTURE_2_VECTOR
_OUTPUT_COMPARE_2_VECTOR
_EXTERNAL_2_VECTOR
_TIMER_3_VECTOR
_INPUT_CAPTURE_3_ERROR_VECTOR
_INPUT_CAPTURE_3_VECTOR
_OUTPUT_COMPARE_3_VECTOR
_EXTERNAL_3_VECTOR
_TIMER_4_VECTOR
_INPUT_CAPTURE_4_ERROR_VECTOR
_INPUT_CAPTURE_4_VECTOR
_OUTPUT_COMPARE_4_VECTOR
_EXTERNAL_4_VECTOR

```
_TIMER_5_VECTOR
_INPUT_CAPTURE_5_ERROR_VECTOR
_INPUT_CAPTURE_5_VECTOR
_OUTPUT_COMPARE_5_VECTOR
_TIMER_6_VECTOR
_INPUT_CAPTURE_6_ERROR_VECTOR
_INPUT_CAPTURE_6_VECTOR
_OUTPUT_COMPARE_6_VECTOR
_TIMER_7_VECTOR
_INPUT_CAPTURE_7_ERROR_VECTOR
_INPUT_CAPTURE_7_VECTOR
_OUTPUT_COMPARE_7_VECTOR
_TIMER_8_VECTOR
_INPUT_CAPTURE_8_ERROR_VECTOR
_INPUT_CAPTURE_8_VECTOR
_OUTPUT_COMPARE_8_VECTOR
_TIMER_9_VECTOR
_INPUT_CAPTURE_9_ERROR_VECTOR
_INPUT_CAPTURE_9_VECTOR
_OUTPUT_COMPARE_9_VECTOR
_ADC1_VECTOR
_ADC1_DC1_VECTOR
_ADC1_DC2_VECTOR
_ADC1_DC3_VECTOR
_ADC1_DC4_VECTOR
_ADC1_DC5_VECTOR
_ADC1_DC6_VECTOR
_ADC1_DF1_VECTOR
_ADC1_DF2_VECTOR
_ADC1_DF3_VECTOR
_ADC1_DF4_VECTOR
_ADC1_DF5_VECTOR
_ADC1_DF6_VECTOR
_ADC1_DATA0_VECTOR
_ADC1_DATA1_VECTOR
_ADC1_DATA2_VECTOR
_ADC1_DATA3_VECTOR
_ADC1_DATA4_VECTOR
_ADC1_DATA5_VECTOR
_ADC1_DATA6_VECTOR
_ADC1_DATA7_VECTOR
_ADC1_DATA8_VECTOR
_ADC1_DATA9_VECTOR
_ADC1_DATA10_VECTOR
_ADC1_DATA11_VECTOR
_ADC1_DATA12_VECTOR
_ADC1_DATA13_VECTOR
_ADC1_DATA14_VECTOR
_ADC1_DATA15_VECTOR
_ADC1_DATA16_VECTOR
_ADC1_DATA17_VECTOR
_ADC1_DATA18_VECTOR
_ADC1_DATA19_VECTOR
_ADC1_DATA20_VECTOR
_ADC1_DATA21_VECTOR
_ADC1_DATA22_VECTOR
_ADC1_DATA23_VECTOR
_ADC1_DATA24_VECTOR
_ADC1_DATA25_VECTOR
_ADC1_DATA26_VECTOR
_ADC1_DATA27_VECTOR
_ADC1_DATA28_VECTOR
_ADC1_DATA29_VECTOR
_ADC1_DATA30_VECTOR
_ADC1_DATA31_VECTOR
_ADC1_DATA32_VECTOR
_ADC1_DATA33_VECTOR
_ADC1_DATA34_VECTOR
_ADC1_DATA43_VECTOR
_ADC1_DATA44_VECTOR
_CORE_PERF_COUNT_VECTOR
_CORE_FAST_DEBUG_CHAN_VECTOR
_SYSTEM_BUS_PROTECTION_VECTOR
_SPI1_FAULT_VECTOR
_SPI1_RX_VECTOR
_SPI1_TX_VECTOR
_UART1_FAULT_VECTOR
_UART1_RX_VECTOR
_UART1_TX_VECTOR
_I2C1_BUS_VECTOR
_I2C1_SLAVE_VECTOR
_I2C1_MASTER_VECTOR
_CHANGE_NOTICE_A_VECTOR
_CHANGE_NOTICE_B_VECTOR
_CHANGE_NOTICE_C_VECTOR
_CHANGE_NOTICE_D_VECTOR
_CHANGE_NOTICE_E_VECTOR
_CHANGE_NOTICE_F_VECTOR
_CHANGE_NOTICE_G_VECTOR
_PMP_VECTOR
_PMP_ERROR_VECTOR
_COMPARATOR_1_VECTOR
_COMPARATOR_2_VECTOR
_USB_VECTOR
_USB_DMA_VECTOR
_DMA0_VECTOR
_DMA1_VECTOR
_DMA2_VECTOR
_DMA3_VECTOR
_DMA4_VECTOR
_DMA5_VECTOR
_DMA6_VECTOR
_DMA7_VECTOR
_SPI2_FAULT_VECTOR
_SPI2_RX_VECTOR
_SPI2_TX_VECTOR
_UART2_FAULT_VECTOR
_UART2_RX_VECTOR
_UART2_TX_VECTOR
_I2C2_BUS_VECTOR
_I2C2_SLAVE_VECTOR
_I2C2_MASTER_VECTOR
_ETHERNET_VECTOR
_SPI3_FAULT_VECTOR
_SPI3_RX_VECTOR
_SPI3_TX_VECTOR
_UART3_FAULT_VECTOR
_UART3_RX_VECTOR
_UART3_TX_VECTOR
_I2C3_BUS_VECTOR
_I2C3_SLAVE_VECTOR
_I2C3_MASTER_VECTOR
```

_SPI4_FAULT_VECTOR
_SPI4_RX_VECTOR
_SPI4_TX_VECTOR
_RTCC_VECTOR
_FLASH_CONTROL_VECTOR
_PREFETCH_VECTOR
_SQI1_VECTOR
_UART4_FAULT_VECTOR
_UART4_RX_VECTOR
_UART4_TX_VECTOR
_I2C4_BUS_VECTOR
_I2C4_SLAVE_VECTOR
_I2C4_MASTER_VECTOR
_SPI5_FAULT_VECTOR
_SPI5_RX_VECTOR
_SPI5_TX_VECTOR
_UART5_FAULT_VECTOR
_UART5_RX_VECTOR
_UART5_TX_VECTOR
_I2C5_BUS_VECTOR
_I2C5_SLAVE_VECTOR
_I2C5_MASTER_VECTOR
_SPI6_FAULT_VECTOR
_SPI6_RX_VECTOR
_SPI6_TX_VECTOR
_UART6_FAULT_VECTOR
_UART6_RX_VECTOR
_UART6_TX_VECTOR

B Interrupt Request Names

B.1 PIC32MX

This list of interrupts is taken from the header file for the PIC32MX795F512L chip. The chip you are using may not have the same list. To find the list of interrupt names for your specific chip you should refer to the header file for your chip. The header file is located within the pic32mx/include/proc folder within the pic32-tools compiler.

_CORE_TIMER_IRQ
_CORE_SOFTWARE_0_IRQ
_CORE_SOFTWARE_1_IRQ
_EXTERNAL_0_IRQ
_TIMER_1_IRQ
_INPUT_CAPTURE_1_IRQ
_OUTPUT_COMPARE_1_IRQ
_EXTERNAL_1_IRQ
_TIMER_2_IRQ
_INPUT_CAPTURE_2_IRQ
_OUTPUT_COMPARE_2_IRQ
_EXTERNAL_2_IRQ
_TIMER_3_IRQ
_INPUT_CAPTURE_3_IRQ
_OUTPUT_COMPARE_3_IRQ
_EXTERNAL_3_IRQ
_TIMER_4_IRQ
_INPUT_CAPTURE_4_IRQ
_OUTPUT_COMPARE_4_IRQ
_EXTERNAL_4_IRQ
_TIMER_5_IRQ
_INPUT_CAPTURE_5_IRQ
_OUTPUT_COMPARE_5_IRQ
_SPI1_ERR_IRQ
_SPI1_RX_IRQ
_SPI1_TX_IRQ
_I2C1A_ERR_IRQ
_I2C3_BUS_IRQ
_SPI1A_ERR_IRQ
_SPI3_ERR_IRQ
_UART1A_ERR_IRQ
_UART1_ERR_IRQ
_I2C1A_RX_IRQ
_I2C3_SLAVE_IRQ
_SPI1A_RX_IRQ
_SPI3_RX_IRQ
_UART1A_RX_IRQ
_UART1_RX_IRQ
_I2C1A_TX_IRQ
_I2C3_MASTER_IRQ
_SPI1A_TX_IRQ
_SPI3_TX_IRQ
_UART1A_TX_IRQ
_UART1_TX_IRQ
_I2C1_BUS_IRQ
_I2C1_SLAVE_IRQ
_I2C1_MASTER_IRQ

_CHANGE_NOTICE_IRQ
_ADC_IRQ
_PMP_IRQ
_COMPARATOR_1_IRQ
_COMPARATOR_2_IRQ
_I2C2A_ERR_IRQ
_I2C4_BUS_IRQ
_SPI2_ERR_IRQ
_SPI2A_ERR_IRQ
_UART2A_ERR_IRQ
_UART3_ERR_IRQ
_I2C2A_RX_IRQ
_I2C4_SLAVE_IRQ
_SPI2_RX_IRQ
_SPI2A_RX_IRQ
_UART2A_RX_IRQ
_UART3_RX_IRQ
_I2C2A_TX_IRQ
_I2C4_MASTER_IRQ
_SPI2A_TX_IRQ
_SPI2_TX_IRQ
_UART2A_TX_IRQ
_UART3_TX_IRQ
_I2C3A_ERR_IRQ
_I2C5_BUS_IRQ
_SPI3A_ERR_IRQ
_SPI4_ERR_IRQ
_UART2_ERR_IRQ
_UART3A_ERR_IRQ
_I2C3A_RX_IRQ
_I2C5_SLAVE_IRQ
_SPI3A_RX_IRQ
_SPI4_RX_IRQ
_UART2_RX_IRQ
_UART3A_RX_IRQ
_I2C3A_TX_IRQ
_I2C5_MASTER_IRQ
_SPI3A_TX_IRQ
_SPI4_TX_IRQ
_UART2_TX_IRQ
_UART3A_TX_IRQ
_I2C2_BUS_IRQ
_I2C2_SLAVE_IRQ
_I2C2_MASTER_IRQ
_FAIL_SAFE_MONITOR_IRQ
_RTCC_IRQ
_DMA0_IRQ
_DMA1_IRQ
_DMA2_IRQ
_DMA3_IRQ
_DMA4_IRQ
_DMA5_IRQ
_DMA6_IRQ
_DMA7_IRQ
_FLASH_CONTROL_IRQ
_USB_IRQ
_CAN1_IRQ
_CAN2_IRQ
_ETHERNET_IRQ
_INPUT_CAPTURE_ERROR_1_IRQ
_INPUT_CAPTURE_ERROR_2_IRQ
_INPUT_CAPTURE_ERROR_3_IRQ
_INPUT_CAPTURE_ERROR_4_IRQ
_INPUT_CAPTURE_ERROR_5_IRQ
_PMP_ERROR_IRQ
_UART1B_ERR_IRQ
_UART4_ERR_IRQ
_UART1B_RX_IRQ
_UART4_RX_IRQ
_UART1B_TX_IRQ
_UART4_TX_IRQ
_UART2B_ERR_IRQ
_UART6_ERR_IRQ
_UART2B_RX_IRQ
_UART6_RX_IRQ
_UART2B_TX_IRQ
_UART6_TX_IRQ
_UART3B_ERR_IRQ
_UART5_ERR_IRQ
_UART3B_RX_IRQ
_UART5_RX_IRQ
_UART3B_TX_IRQ
_UART5_TX_IRQ

B.2 PIC32MZ

The PIC32MZ family has a direct 1:1 mapping between interrupt vector numbers and interrupt request numbers. You should use the vector name in place of any interrupt request names for the PIC32MZ family.

Glossary

interrupt flag Whenever an interrupt occurs. 4, 11

vector table The vector table is a list of addresses in memory. When an interrupt occurs. 2, 11

C Document Revisions

1. MJ . . . 24/02/2015 . . . . Initial draft
2. MJ . . . 26/02/2015 . . . . Added User ISR macro information
3. SB . . . 03/03/2015 . . . . Improved wording for interrupt workings