chipKIT™ PIC32 Interrupt Handling

As you might already know, the chipKIT™ platform is based on the Microchip PIC32MX and PIC32MZ microcontrollers (MCUs). In this document, we will provide a deep dive into PIC32-based interrupt handling. For the Microchip Reference Manual for PIC32 Interrupts, click here.

General Concepts

An interrupt happens when a hardware event on the MCU suspends the normal flow of execution. At this time, the execution switches to an ISR (Interrupt Service Routine) in order to perform the required task, after which, execution returns to its previous state prior to the interrupt. Since hardware events can occur at any time, interrupts are unpredictable, so we do face some challenges with program execution flow. Because of interrupts, data used by a program can seemingly change unexpectedly, at any time, within a line of code, or within a multi-cycle instruction of the program, but this is all part of the 'interrupt' process. In order for this process to work properly, ISRs require special prologue and epilogue code that will do the following:

- Preserve the execution state of the main-line program
- Instantiate a state suitable for the ISR to execute and then restore the main-line execution state when ISR execution is complete

The context switch to an ISR is actually quite complex, as many things need to happen:

- Setting the instruction pointer to the vectored ISR
- Preserving the program register sets
- Setting and restoring execution priorities
- Enabling/disabling interrupts
- Dealing with many other hardware conditions potentially beyond the programmer's required understanding

As you can see, an ISR is not just a regular function, but a special routine that contains important code sequences. Fortunately, the compiler can generate the ISR prologue and epilogue code for you as long as you specify the resources the ISR will use, so thankfully, you can successfully write and use interrupts with only a conceptual understanding of the details.

Interrupts (IRQs), Interrupt Vectors, and Events

People often use the term 'interrupt' in a collective sense, to describe the more fundamental preemptive event implemented in an MCU. In reality, however, there are three components of the 'interrupt' system: events, interrupts (IRQs), and vectored interrupt service routines (ISRs). As you might see, they can be easily confused. In fact, even example code in official documentation often provides unnecessary code sequences. To help dispel some confusion, we will review the three components in more detail.

Events

The *event* is the fundamental signal of a microcontroller. An event is not controllable; it happens in response to some condition and results in a flags being set in a peripheral registers. Here are the steps that take place:

1. A condition produces an *event*
2. System sets the associated peripheral register flags

Unfortunately, because people often confuse the various components of the 'interrupt' system, they say inter-peripheral "interrupts" instead of inter-peripheral *events*. This in turn leads them to write bad examples that enable interrupts and clear interrupt flags, when those actions are unnecessary. For example, to trigger an SPI transfer on a DMA (Direct Memory Access) event does not require you to enable interrupts, check interrupt flags, or provide an ISR routine because such events merely happen, and they are automatically cleared. In other words, the SPI will continue to get an *event* each time the DMA signals it without the need for enabled or cleared interrupt flags. Regrettably, this is probably the most commonly misdocumented error in most examples.

Let's move on to the next component of the '*interrupt*' system.

Interrupts (IRQs)

The word '*interrupt*' is actually a short name for '*interrupt request*' (IRQ). For each event in the system there is an IRQ number assigned to the event. All events are wired to the interrupt controller where the event will set a program accessible "IF" flag. It is the interrupt controller that bridges events to interrupt requests (IRQ). In a sense, the interrupt controller is, in itself, another hardware peripheral that responds to hardware events by setting the program accessible flags in response to the event. For the most part, the interrupt controller exists to trigger the transfer of execution to an ISR; it does not exist and does not participate in triggering other hardware peripherals.

Depending on whether the IRQ is enabled in the interrupt controller (the "IE" flag), the interrupt controller may initiate a transfer of execution to an interrupt service routine (ISR).

Perhaps some confusion arises between *events* and *interrupts* (IRQs) because as programmers, we think in terms of responding to IRQs as presented by the interrupt controller. Since the interrupt controller responds directly to all hardware events, it is easy to confuse whether we are writing code to the event or the interrupt controller's IRQ. Hardware peripherals are wired directly to events and do not require the interrupt controller to be triggered, whereas ISRs are triggered by the interrupt controller, a layer above events. When an event is triggered in the hardware here is what happens:

1. A condition produces an *event*
2. Any peripheral hard wired to that event will have its peripheral register flag set and will respond to the event as the peripheral is configured.
3. Since all events are wired to the interrupt controller, the "IF" flag associated to that event is set in the interrupt controller.

4. If the interrupt controller is configured to trigger an ISR by having the “IE” flag set, execution is transferred to the associated vectored interrupt service routine (ISR).

Note that because there is a one-to-one mapping of events to interrupts for PIC32 devices, the PIC32 documentation often refers to *events* as *interrupts*. That is ok as long as you understand what we just discussed about the difference between *events* and *interrupts* (IRQs).

Interrupt Vectors and Interrupt Service Routines

The interrupt controller is responsible for initiating a transfer of execution to an ISR when an enabled interrupt is triggered. In order to do this, the interrupt controller needs to know what ISR to trigger and it does so by looking up the interrupt vector pointer in an interrupt vector table based off of the IRQ number. However, just because an IRQ has been presented to the interrupt controller, the following conditions must be met before the interrupt controller will transfer execution to the ISR:

- The interrupt flag (IF) is set
- The interrupt is enabled (i.e., the IE flag is set, and master interrupts are enabled)
- The priority of the ISR is higher than the currently running code

Here are the steps that take place, now taking into account *interrupt vectors* and ISRs.

1. A condition produces an *event*
2. The “IF” flag associated to that event is set in the interrupt controller
3. If the IE flag for that interrupt is enabled, and master interrupts are enabled, and the interrupting priority is higher than the current running priority, the interrupt is taken
4. The interrupt controller maps the IRQ number to an interrupt vector index
5. The interrupt vector table is indexed, retrieving the ISR address
6. Execution is transferred to the ISR

For an interrupt to be enabled, both the global interrupt enable (CP0 Status.IE) bit and the interrupt enable flag for the specific interrupt (IECXx) must be set.

In addition, the priority (IPCx) level of the interrupt must be set to one of the seven priority levels available to PIC32 devices. Priority 0 is the lowest, but also means the interrupt is disabled. Priority 7 is the highest interrupt priority. If two interrupts need to be serviced at the same time, the one with the highest priority will be serviced first. In addition, a lower priority interrupt running its ISR will be preempted by a higher priority interrupt.

Subpriority levels are available in the same IPCx registers, but they affect the CPU behavior when two interrupts of the same priority occur. In this case, the interrupt with the higher subpriority will be serviced first. If one interrupt is being serviced and another interrupt occurs with the same priority but a higher subpriority, the new interrupt will have to wait for the other ISR to finish first.

Notice that the execution of the ISR is not dependent on the underlying event, but on whether the interrupt flag (IF) is set. The interrupt flag is usually set by the hardware in response to an event. However, it can also be set manually in software simply by setting the IF flag for the desired IRQ.

On the MX MCUs, there is a many-to-one mapping of interrupts (IRQs) to interrupt vectors (ISRs). That is, several peripherals will share the same interrupt vector, or ISR. For example, SPI, UART, and I2C share interrupt vectors, or ISRs. There is an implication here, if you ever enable an interrupt, you better have an ISR for it because the interrupt vector will get executed. If no ISR is installed, either a default ISR will be executed, or an ISR fault will occur. In the chipKIT core libraries, we have preinstalled every interrupt vector to point either to the general exception fault ISR or to a core peripheral ISR handler—if it is one of the chipKIT core's natively supported peripherals (like the serial-monitor UART).

The many-to-one mapping of IRQs to ISRs does cause a complication, though; if the chipKIT core is to support multiple peripherals that share an ISR, there has to be a way to dynamically set the ISR at runtime so that the appropriate ISR for the peripheral is installed for the intended peripheral in use. For example, let's say you want to use UART2 in one sketch and SPI4 in another. Since both UART2 and SPI4 use the same *interrupt vector number*/ISR, there has to be a way at runtime to install either the UART2 ISR or the SPI4 ISR, depending on the application. To do this, call `setIntVector()` to install the appropriate ISR for the peripheral in use. However, for chipKIT core peripheral libraries, such as Serial, DSPI, or DTWI, setting the ISR is handled automatically for you and you don't need to worry about using `setIntVector()` (more on these specifics later).

On the MZ MCU there is a one-to-one mapping of IRQs to interrupt vectors, so no peripheral will share ISRs with other peripherals, and technically the use of `setIntVector()` is not needed. However, `setIntVector()` is supported on the MZ so that user-defined ISRs may be installed at runtime without knowing the specifics of the *interrupt vector table*. This also allows for common code to be written for both the MX and MZ MCUs.

What happens after the trigger of an interrupt

The things that happen after an interrupt occurrence vary considerably depending on the hardware architecture, case in point PIC32MX vs. PIC32MZ. Generally speaking, however, when an interrupt occurs, the following happens:

- If (all/any) interrupts are enabled
  - If that specific interrupt is enabled AND the priority for the interrupt is higher than the current execution priority
    - Halt the code execution
    - Load the ISR address
    - Invoke the ISR execution

Considerations when writing ISRs

An ISR can be called at any time and will run at a higher execution priority than the main loop code. However, while it is running at a higher priority, it may not be at the highest priority and another ISR could preempt the ISR at any time. On the PIC32 family, an ISR will not preempt itself since another interrupt on that same IRQ will be at the same priority and will not preempt the current execution, and thus will not cause recursive/reentrant execution. While this is true for the PIC32 MCU, it is not necessarily true in general across other processor architectures.

When execution starts in the ISR, the IF flag will be set, and interrupts will still be enabled; even the IRQ for the executing ISR interrupt will be enabled. This state is determined as a combination of the hardware when the interrupt occurred and the prologue code generated by the compiler. For example, it is very possible that interrupts are disabled by the hardware but re-enabled by the ISR prologue code; suffice it to say, interrupts will be enabled when entering the ISR.

The ISR should be written to execute quickly—on the order of microseconds—because the ISR has preempted the main loop which may be doing some timing critical task. In general, little consideration is given to interrupts occurring when coding the main loop tasks. ISRs should also be written to assume multiple interrupts have occurred—not just one—because other higher priority interrupts may have run while multiple interrupts occurred on the lower priority IRQ. Also, it is up to the ISR to clear the IF flag for the IRQ. Many coders will do this as the last step in the ISR, but that has its problems should another interrupt occur while processing the ISR. As long as the IF flag is set, it is impossible to know if more interrupts came in. A more practical solution is to clear the IF flag as the very first statement in the ISR, then write the ISR assuming multiple interrupts had occurred. Then at the end of the ISR, check to see if the IF flag is set (indicating an interrupt came in while processing the ISR) and loop back again and process the ISR again to handle the new interrupt. Or, just return from the ISR and if the IF flag is set, the interrupt controller will immediately re-invoke the ISR without ever returning to the main loop.

Remember, the interrupt controller runs on top of the event system and controls the invoking of ISRs. If the interrupt is enabled, all it takes to invoke the ISR is to set the IF flag. This can be done by the interrupt controller when the event triggers, or the IF flag can be manually set by software. Often times when processing a stack of items in an ISR, it is easy to write it processing one item and then manually setting the IF flag at the end of the ISR and let the interrupt system re-invoke the ISR to process the next item on the stack. This technique can also be used to prime a sequence of events (such as an I2C communication) by filling an interrupt-driven buffer and then manually triggering the interrupt by setting the IF flag and then letting the peripheral and interrupt process flush the buffer.

Understand, since the interrupt controller is built on top of the event systems, setting the IF flag will not trigger an event. For example, if the DMA is configured to trigger off of an event to transfer a cell of data, setting the IF flag for that IRQ will not trigger the DMA cell transfer. This happens because the DMA peripheral is triggered via events, not IRQs. For example, to trigger a DMA cell transfer the DCHxECONbits.CHSIRQ is set to the IRQ (number) to specify the “event” that the DMA will trigger. While the documentation uses the IRQ acronym, it is really the event that is being specified. If the cell transfer is to be triggered by software, the self-resetting DCHxECONbits.CFORCE bit must be set in the DMA controller to force the trigger. The only reason the CFORCE bit exists in the DMA controller is that it just won’t work to set the IRQ IF flag, as the interrupt controller cannot initiate events.

PIC32 MX / MZ interrupt events

When an MX / MZ interrupt event occurs, the hardware will find the appropriate ISR based on a hardware-defined, *interrupt vector number* (see PIC32 family datasheets). The hardware will use this *vector number* to index into an *interrupt vector table* containing ISR start addresses. Then the hardware will load the ISR start address into the program counter (a.k.a. instruction pointer) for immediate execution.

With a hardware/software mix, the hardware and the ISR prologue code will work in cooperation to switch the execution state from the mainline program to the ISR. On the MX and MZ MCUs, the hardware switches the priority; however, hardware or software can save or switch the register set depending on how the ISR was defined. The PIC32 MCU supports, to varying degrees, hardware shadow register sets that the hardware can switch between. If a shadow register set is not used, the ISR prologue code will save the current register set on the stack before executing the user-defined ISR code.

In order for the compiler to generate the correct prologue/epilogue code, the user must write the ISR in a way that is consistent with how the hardware will respond. When writing the ISR, the user must ensure the following two requirements are completed:

1) Setting the hardware control registers programmatically
2) Specifying the attributes on the ISR
    a. Specify options to aid the compiler in generating fast code
    b. Let the compiler generate slower code that will figure it all out at runtime

Note above that for specifying attributes on the ISR, a user has the option to either specify some parameters to allow the compiler to generate faster code, or have the compiler generate slower code that figures it all out. The user must be aware that specifying parameters comes with the bonus of faster execution time, but with the risks of elusive bugs during runtime if the code is not written correctly and consistently. These bugs will appear, for example, if the hardware runs at a different priority than that specified on the ISR. Likewise, if the prologue /epilogue code assumes the hardware is using a shadow register set, when the hardware actually doesn't, the user will most definitely see register corruption.

To aid users in getting things right, the compiler has an option for dynamically figuring everything out in the prologue code at runtime. Instead of the fast method, the user chooses to <u>not</u> specify any attributes on the ISR function, which "tells" the compiler to determine the hardware condition and usage at runtime. Note, however, that choosing this option increases the size of the prologue /epilogue code and increases the execution time of the ISR.

Interrupt handing in the chipKIT™ core environment

Predefined ISRs

The chipKIT core libraries predefine both the hardware and software for all the required core ISRs—including the core-timer ISR and core-peripheral ISRs. To see the peripheral definitions, open the board's variant "Board_Defs.h" file, where you will see things like `IPL2SOFT` and `IPL3SRS`—for priority 2 software (copied) context saving and priority 3 and shadow register-set context saving, respectively. Keep in mind that you don't want to change these values, as this mechanism is paired with how the chipKIT core sets up the hardware control registers. Again, we always want the compiler prologue/epilogue code to match with the hardware configuration to avoid elusive bugs.

User-defined ISRs

User-defined ISRs are most commonly used with external interrupts. Fortunately, the chipKIT core has a set of built-in interrupt handlers to provide the standard Arduino functions `attachInterrupt()` and `detachInterrupt()` for external interrupt pins. The chipKIT core provides ISRs for all five external interrupts available on the MX and MZ MCUs. The user merely needs to create a standard C callback function and then attach that callback function to the system using `attachInterrupt()`. You can remove the callback function with `detachInterrupt()`. There is no need to apply any ISR function attributes to your callback function as the core has handlers already installed to deal with the idiosyncrasies of defining the ISRs. However, remember that your callback function will be called from within the chipKIT-core external interrupt handler, so keep the code short, as it can be invoked at any time. Also keep in mind that the chipKIT core defines the external interrupts to run at priority level 4 (IPL4), just above the CPU which typically runs at priority 3 (IPL3).

To define your own interrupt handlers outside of the chipKIT core, you will need to define your interrupt handler with the appropriate interrupt attributes. Because the MX and MZ MCUs are different, you can use the chipKIT-core defined "`__USER_ISR`" macro, which will adjust for either MCU. The ISR should be declared as:

```
void __USER_ISR myISR(void);
```

You are responsible for setting the hardware *vector* and the *priority* with `setIntVector()` and `setIntPriority()`. Don't worry about matching the ISR priority or register set to the hardware because the `__USER_ISR` macro will instruct the compiler to generate prologue/epilogue code to determine the hardware settings at runtime.

If you wish to use the most optimal prologue/epilogue code and define your ISR with all of the appropriate attributes, you must do so using the native compiler interrupt attributes and set your hardware registers consistently. This will vary from the MX and MZ and should only be considered if such performance is needed and you are comfortable with the interrupt mechanism. Getting this wrong will probably yield sporadic and unpredictable results.

The hardware specifics of the PIC32 MCU interrupt handling

The hardware specifics vary significantly between the MX and MZ MCUs. As was stated before, the MX MCU has a many-to-one mapping of IRQs to *interrupt vectors*, whereas the MZ has a one-to-one mapping between IRQs and *interrupt vectors*. It is, however, more important to note that there is a difference in the way the *vector* is stored, found, and invoked. Also, the MZ family has seven shadow register sets whereas the MX family can have either one or none.

How events and IRQs work is considerably similar between the MX and MZ, but interrupt-vector lookup is substantially different and will be described below.

The MX MCU

Shadow Register Set

Most PIC32MX MCU variants support one shadow register set. Other PIC32MX MCU variants have no shadow register set. For devices with a single shadow register set, the shadow register set is preassigned to a particular priority level. In the case of chipKIT™, the shadow register set is assigned to priority 7. (`FSRSSEL => DEVCFG3: <18:16> == 7`). So whenever an ISR will run at priority 7 it should use the IPL7SRS attribute, or no attribute should be specified so the compiler generated prologue/epilogue code will dynamically figure it out.

Interrupt Vector

At compile and link time, an *interrupt vector table* is created and placed in the flash address space somewhere. It is not particularly important where, but typically it is somewhere near the beginning of program flash. The exact location is specified by the linker script. In the case of an MPLAB X default placement, the *interrupt vector table* is in bootflash at location 0x9FC01000. However, the chipKIT™ core is unable to write bootflash as the bootloader resides there, so the chipKIT core's default linker script loads the interrupt vector table at 0x9D000000—the very first byte in program flash. The *interrupt vector table* for the MX MCU has a maximum of 64 entries.

Conceptually the *interrupt vector table* would just be a list of 64 ISR pointers. However, this is not how it works. The table is structured as 64 blocks of executable space. The size of each executable block is defined by the vector spacing, usually specified in the linker script to be 1, which indicates a spacing of 32 bytes per block. So with 64 entries at 32 bytes each the table is 64 * 32 = 2048 bytes long. There is an additional 512 substantially unused bytes at the beginning of the table, so the table is actually 2560 bytes long. In the chipKIT core, we reserve 1 page—or 4096 bytes—for the *interrupt vector table*. The additional bytes are reserved for future use, which will probably never be realized. The MX MCU allows for vector spacing of 0, 8, 16, 32, 64, 128, 256, and 512 bytes; however, the linker spaces these at 32 byte intervals, so the linker only supports 0, 32, 64, 128, 256, and 512 bytes.

When an *interrupt* is taken, the MCU maps the IRQ to the *interrupt vector number* and then execution is transferred to the address based off of the following calculation:

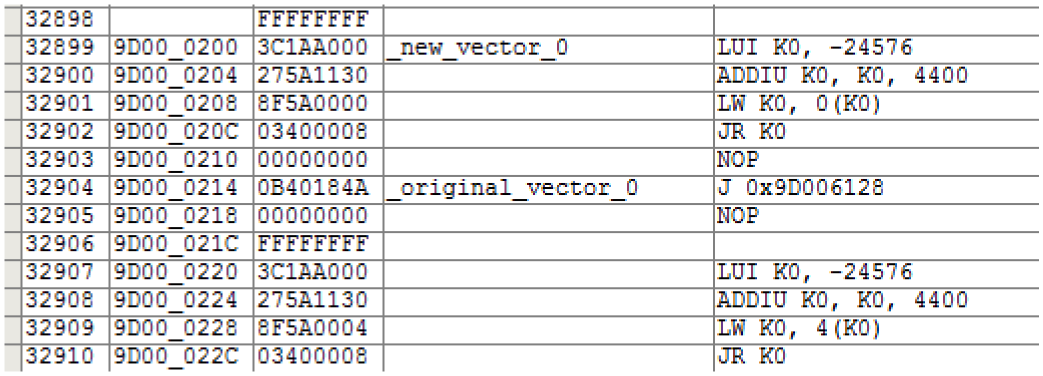
*Execution Address = EBase + (vector spacing) * (interrupt vector number) + 0x200*

Where:

EBase == 0x9D000000; defined in the linker script as the start of program flash
vector spacing == 32 bytes; defined in the linker script as _vector_spacing = 0x00000001

Execution is transferred to this address. Typically the compiler will load a jump instruction there that will jump to the ISR defined for that *interrupt vector*. However, if the code can fit within the 32 bytes, the code could be instantiated in the 32 byte block without a jump instruction. This is often done for the general exception handler which is just a while(1), or a jump to itself.

Here is an example of the *interrupt vector table* generated by the chipKIT core:



32898		FFFFFFFF		
32899	9D00_0200	3C1AA000	_new_vector_0	LUI K0, -24576
32900	9D00_0204	275A1130		ADDIU K0, K0, 4400
32901	9D00_0208	8F5A0000		LW K0, 0(K0)
32902	9D00_020C	03400008		JR K0
32903	9D00_0210	00000000		NOP
32904	9D00_0214	0B40184A	_original_vector_0	J 0x9D006128
32905	9D00_0218	00000000		NOP
32906	9D00_021C	FFFFFFFF		
32907	9D00_0220	3C1AA000		LUI K0, -24576
32908	9D00_0224	275A1130		ADDIU K0, K0, 4400
32909	9D00_0228	8F5A0004		LW K0, 4(K0)
32910	9D00_022C	03400008		JR K0

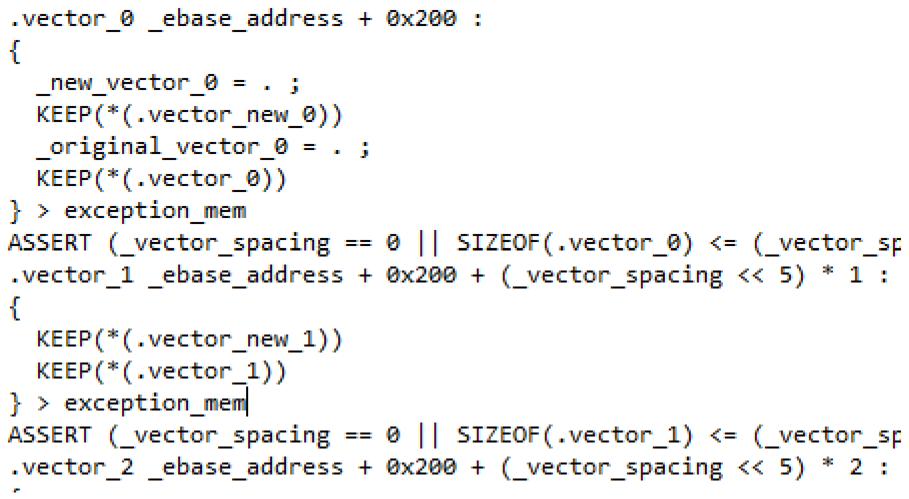
Notice that vector 0 starts at 0x9D000200; it does so because of that 512 unused bytes at the start of the table. However, notice that within the 1st 32-byte block, there seem to be two code fragments, _new_vector_0, and _original_vector_0. These fragments are a result of the chipKIT core's method to resolve the many-to-one IRQ to *interrupt vector* problem, and the implementation of `setIntVector()`.

Many IRQ to one interrupt vector solution

If the chipKIT core had not fooled around with the *interrupt vector table*, the code found at `_original_vector_0` would have been placed at `_new_vector_0`; notice that it contains a simple jump instruction. Typically the compiler would take the ISR address, generate a jump instruction, and place the jump at the beginning of the 32-byte vector code space.

However, because several peripherals share the same interrupt vector, there is a need to dynamically change the ISR address at runtime. If the *interrupt vector table* were placed in RAM, we could just go in and modify the jump address at runtime. However, the *interrupt vector table* is loaded in flash, and the jump address is fixed and cannot be changed at runtime.

To understand how to fix this problem, we need to understand how the compiler and linker go about creating the *interrupt vector table*. Look at the linker script code to generate the *interrupt vector table*:



```
.vector_0 _ebase_address + 0x200 :
{
  _new_vector_0 = . ;
  KEEP(*(.vector_new_0))
  _original_vector_0 = . ;
  KEEP(*(.vector_0))
} > exception_mem
ASSERT (_vector_spacing == 0 || SIZEOF(.vector_0) <= (_vector_sp
.vector_1 _ebase_address + 0x200 + (_vector_spacing << 5) * 1 :
{
  KEEP(*(.vector_new_1))
  KEEP(*(.vector_1))
} > exception_mem
ASSERT (_vector_spacing == 0 || SIZEOF(.vector_1) <= (_vector_sp
.vector_2 _ebase_address + 0x200 + (_vector_spacing << 5) * 2 :
```

First notice there is a section for each *interrupt vector*, `.vector_0`, `.vector_1`, `.vector_2` and so on. Notice how each of these section are specifically place some multiple of `_vector_spacing * 32` apart starting at `_ebase_address + 0200`. Then notice that in each section we are `KEEP()`ing code placed by the compiler in sections `.vector_new_x` and `.vector_x`.

Before we fooled with the vector table, all of that `.vector_new_x` stuff wasn't there. The compiler would place the jump instruction to the ISR in the appropriate interrupt vector code section based off of the vector-number attribute specified on the ISR function. When the MCU took the interrupt, it started execution at the calculated address, and the compiler-generated jump instruction was executed to jump to the ISR.

However, we need a way to modify the ISR jump address at runtime. The solution was to indirectly jump from a value stored in RAM. Notice how at `_new_vector_0` the code there loads a table in K0, indexes into K0, and then jumps indirectly through K0. That table is a table the chipKIT core defines in RAM. There is a very special assembly file `vector_table.S` that defines a little snippet of code that gets placed at `.vector_new_0` that generates an indirect jump through a table called `_isr_primary_install`. `_isr_primary_install`—a 64-entry array of ISR function pointers pre-initialized to the general exception handler.

When `setIntVector()` is called, the supplied ISR pointer is written to the specified vector entry in the `_isr_primary_install` table, and when the interrupt is taken, the indirect jump using the new ISR address is used.

An interesting question is, "why do we keep the original ISR jump instruction placed at `_oringinal_vector_x`?" This is for backwards compatibility. With the existence of `setIntVector()`, there is no need to tell the compiler which vector an ISR should be installed at; that is because it will be done at runtime with `setIntVector()`. However, old code is still using the ISR vector attribute defined on the original ISR code. We go ahead and placed this compiler-generated jump instruction after our new indirect jump code just so we know what it is, and so we can recover it. When the chipKIT core initializes the `_isr_primary_install` table, it first checks to see if that jump instruction is something other than the general exception handler, and if it is, then the chipKIT core will preload the `_isr_primary_install` entry with the original ISR address. This way, old code that specified the vector number attribute on the ISR and did not call `setIntVector()` would still work; well up until someone calls `setIntVector()` that overwrote that entry.

If you look closely at how `setIntVector()` is defined, it returns the previous value set for the vector. Ideally, if you set and remove an ISR, you would return the entry to its original ISR value. Of course, envisioning a rational system where an interrupt vector would be used for more than one peripheral for the life of a sketch is difficult.

The MZ MCU

Shadow Register Sets

The MZ MCU supports seven shadow register sets. There is no requirement that a shadow register set be used, but when the chipKIT core initializes *multivector interrupts*, it sets a shadow register set to each execution priority. For ease of use, each priority level contains the numerically same shadow register set; that is, priority 0 has shadow register set 0, 1 is 1, 2 is 2 and so on to priority 7 is shadow register set 7. The actual code that does this is:

```
PRISS = 0x76543210;
```

It is highly recommended that this not be tampered with as the chipKIT core makes assumptions about this assignment of priorities and register sets.

If you write your own ISR and specify the priority and register set attribute, you should use IPLxSRS where x is the priority you are running the ISR at. As on the MX, make sure if you specify a priority on the ISR attribute you also set the hardware priority (IPCXx) consistently or sporadic and unpredictable results will occur.

Interrupt Vector

The MZ MCU has a one-to-one mapping of IRQs to interrupt vectors and the problem of peripherals sharing is not an issue with the MZ MCU. However on the MZ the interrupt vector table is part of the memory mapped control registers starting at location `0xBF810540` (OFF000) and contains 191 vector addresses.

Unlike the MX, the MZ does indeed have an *interrupt vector table* that consists solely of ISR addresses, and when an interrupt occurs, the MZ MCU indexes into the interrupt vector table and gets the ISR address. However, there are a few twists to that ISR address; the ISR address is relative to the EBase address and is only 18 bits long. This means the ISR must reside within 256K of EBase. The formula for getting the ISR address is:

```
ISRAddr = ((uint32_t *) &OFF00)[vector number] + EBase;
```

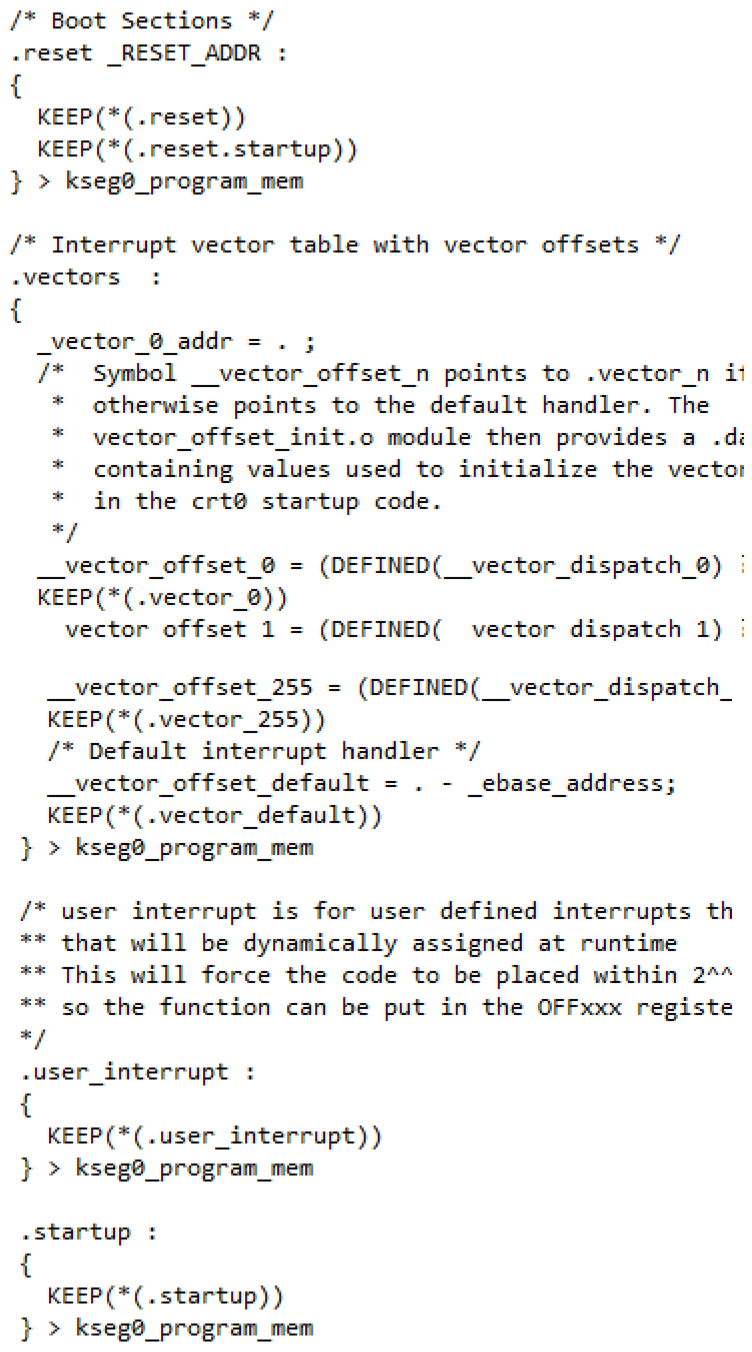
Where:

&OFF00 == 0xBF810540	memory mapped location of the interrupt vector table
EBase == 0x9D000000	assigned by the default chipKIT core linker script

But there are still some issues; the ISR must be within 256K of EBase, and the chipKIT core sets EBase at the beginning of flash, so that means all ISRs must be within the first 245K of flash. To ensure that the ISR is placed at the beginning of flash, as part of the `__USER_ISR`, we place the ISR in a special `.user_interrupt` code section, which the linker script places at the beginning of flash. Here is how the `__USER_ISR` macro is defined:

```
        #define __USER_ISR __attribute__((nomips16, interrupt(),
section(".user_interrupt")))
```

In the linker script we have the following:



```
/* Boot Sections */
.reset _RESET_ADDR :
{
  KEEP(*(.reset))
  KEEP(*(.reset.startup))
} > kseg0_program_mem

/* Interrupt vector table with vector offsets */
.vectors  :
{
  _vector_0_addr = . ;
  /*  Symbol __vector_offset_n points to .vector_n i
   *  otherwise points to the default handler. The
   *  vector_offset_init.o module then provides a .d
   *  containing values used to initialize the vecto
   *  in the crt0 startup code.
   */
  __vector_offset_0 = (DEFINED(__vector_dispatch_0)
  KEEP(*(.vector_0))
    vector offset 1 = (DEFINED(  vector dispatch 1)

   __vector_offset_255 = (DEFINED(__vector_dispatch_
   KEEP(*(.vector_255))
   /* Default interrupt handler */
   __vector_offset_default = . - _ebase_address;
   KEEP(*(.vector_default))
 } > kseg0_program_mem

 /* user interrupt is for user defined interrupts th
 ** that will be dynamically assigned at runtime
 ** This will force the code to be placed within 2^^
 ** so the function can be put in the OFFxxx registe
 */
 .user_interrupt :
 {
   KEEP(*(.user_interrupt))
 } > kseg0_program_mem

 .startup :
 {
   KEEP(*(.startup))
 } > kseg0_program_mem
```

Where:

_ebase_address == 0x9D000000

_RESET_ADDR == 0x9D001000

The ISRs will be place after the .reset code but before the .startup code, very low in the flash program space, but not exactly at EBase. The implication here is that the sum in size of all ISRs must be less than 256K bytes in length, in order to be placed within the 256K range of EBase. If the actual math is done, it is something like:

262144 – 4096 – 16 == 258032 bytes

There are no checks if you have more than 256K of ISR code; however, if the sketch consumes that much interrupt routine space, it is arguable that the sketch was very poorly written and probably would not run anyway, as to just execute that much ISR code would cause all kinds of CPU starvation.

Now you might have noticed that the MZ linker script also has a vector table defined in flash. You can see entries in the linker script for `__vector_offset_0` -> `__vector_offset_255`; yet we said there was no vector table in flash? By default the compiler will not put the ISR address in the *interrupt vector table*; instead the compiler will put an address to the `__vector_offset_x` location and then place a jump instruction there to the actual ISR. The compiler does this to circumvent the 256K EBase restriction, as all of the jump instructions will be placed before the .startup code, well within the 256K EBase limit, and then the jump instruction can access the entire address spaced of the processor. However, this introduces a double jump, one from the address found in the interrupt vector table, and then another to jump to the ISR. By using the `at_vector` attribute (instead of the vector attribute) you can tell the compiler to not add this double jump and instead, put the actual ISR address in the *interrupt vector table*. Here is an example of how the serial monitor UART ISR is defined with the `at_vector` attribute:

```
      void
__attribute__((nomips16,at_vector(_SER0_VECTOR),interrupt(_SER0_IPL_ISR)))
IntSer0Handler(void)
```

We need to be a little careful when looking at the chipKIT core interrupt definitions because these were all written to generate the most efficient prologue/epilogue code. Doing so requires careful consistency between the hardware register sets and priorities definitions and the ISR definition. In the chipKIT core `__USER_ISR` macro we ask the compiler to generate the long form of the prologue/epilogue and determine our register set and priority at runtime, so we do not specify any register set or priority attributes. Likewise, by not even specifying a vector attribute, this tells the compiler to not place the ISR in any interrupt vector table or generate a jump instruction. In a sketch, if an interrupt is to be used it is expected that `setIntVector()` will be used, and `setIntVector()` will just write the supplied ISR address into the interrupt vector table directly, much like `setIntVector()` wrote the ISR address into the `_isr_primary_install` table on the MX. Of course `setIntVector()` takes into account the EBase offset when storing the ISR address. As a reminder, whenever setting an interrupt with `setIntVector()`, also set the priority with `setIntPriority()`.