Interrupt Controller

Purpose:
• This module covers the features and interoperability of the Interrupt Controller contained in the SIU. It receives requests from multiple controllers and determines where the interrupt actually came from.

Objectives:
• To understand how the controller disseminates interrupt requests and how the controller itself is structured.

Contents:
• This section covers the structure of the Interrupt Controller along with the priority registers that control how the controller decides the origin of the request.

Learning Time:
• There are 9 pages in this module with 7 reference pages which take approximately 20 minutes to finish.

This module covers the features and interoperability of the Interrupt Controller contained in the SIU. It receives requests from multiple controllers and determines where the interrupt actually came from.

The objective of this module is to provide you with an understanding of how the controller disseminates interrupt requests and how the controller itself is structured.

This section covers the structure of the Interrupt Controller along with the priority registers that control how the controller decides the origin of the request.

There are 9 pages in this module with 7 reference pages which take approximately 20 minutes to finish.
The key features of the interrupt controller are:

21 CPM interrupt sources

3 SIU interrupt sources

24 external sources, 16 from port C and eight individual interrupt request pins.

It controls:
Programmable priority between periodic timer, PCI and the timer-counter

Programmable priority between SCCs, FCCs and MCCs

Two priority schemes for SCCs, grouped and spread priority

Highest priority request

There is a unique vector number for each interrupt request source.
Interrupt Structure

This diagram illustrates the structure of the interrupt controller. All potential external interrupt sources, except one, are routed through the interrupt controller, prioritised, and where not masked, result in a single interrupt request to the core. Two interrupt sources are treated differently by connection directly to the CPU generating a machine check exception. These are the software watchdog timer and interrupt request zero.

The software watchdog timeout indicates a severe software problem, typically an endless loop or runaway code, requiring the processor to immediately stop normal operation and report or attempt to correct the error. The external interrupt request zero signal should be reserved for extreme errors that also require similar processor action to the software watchdog timeout. The remaining external inputs are controlled by the interrupt controller, and all can be masked. There is some control over relative priorities between all these inputs. Each external interrupt request is configurable to be edge triggered or active low.

Similarly, the port C pins can also be selected to edge triggered or generate a request on any change in level. The inputs shown do not indicate any priority scheme, simply all those that can generate interrupt requests. These inputs are coming from the event registers of the controllers. Remember that they only generate an output if they are not masked, so the interrupt controller only recognizes unmasked requests.

In some cases the input can reflect numerous events. There is only one input from each controller, so if that controller is capable of reporting more than one event, then work is involved in finding the original cause. When the CPU receives an interrupt request it initially only knows that it is an external interrupt request and so jumps to the external exception vector. It must then find out which of these inputs generated the request and then, in many cases, find the specific cause. The interrupt controller provides the means of determining which of these inputs generated the request.
Interrupt Recognition

Each controller has an event register where the events that occur in the normal operation of the controller are recorded. Each event register has a corresponding mask register. Only if the related bit in the mask register is set will the event be recognized by the interrupt controller. At any time there could be several events recorded, so when the interrupt request is serviced the service routine must discover which events are recorded, and deal with them accordingly. That is entirely the responsibility of the service routine.

The interrupt controller contains a register similar to the event register, where all service requests that reach it are recorded. Remember, only those events not masked reach this stage. This register is called the interrupt pending register, and it also has a corresponding mask register. For an interrupt request to be recognized for any interrupt pending, the related mask bit must be set.

When an interrupt request is sent to the CPU, the next hurdle is the machine state register of the processor. Only if the enable external interrupt bit EE is set, will an interrupt request be recognized by the processor. This simple example illustrates a possible scenario. There are three controllers in use, each with several, but not all events unmasked. The event register is shown on the left of each pair and the mask register on the right. Masked events are shown in red with a zero, and enabled ones in green with a one. Some events are enabled, but have not occurred, and so the interrupt pending bit associated with those is clear. Some events have occurred but are masked, but others are enabled and so the interrupt pending bit associated with them is set.

There are several interrupt pending bits set, but some of those are masked and so no interrupt request will result from those. Two interrupt pending bits are enabled and so an interrupt request will result from those. The next consideration is if more than one interrupt request is pending how will they be serviced? Only one can be serviced at a time, which will be the first?

The question is which would you like to be serviced first – probably the one with the highest priority.
As mentioned earlier, it is possible to control some of the relative priority between some of the controllers. There are three registers which allow the user to determine the relative priorities of related peripherals. The SIPRR provides the ability to choose the relative priorities of interrupt request pins one through five, the periodic interrupt timer, the timer counter and the PCI interface. These are known as the XSIU options.

The SCPRR_L enables the relative priorities of the SCCs to be selected. These are known as the YCC options. And SCPRR_H enables the relative priorities of the FCCs and MCCs to be selected. These are known as the XCC options. The selection of the XCC options is fixed as defined by the selection made, but there is another level of relationship for the XSIU and the YCC options.

The Interrupt configuration register, SICR, selects either group or spread priority scheme. This means that the relative priorities are either grouped together or spread out among all the other requests. If you are using all the SCCs, as well as many of the other functions available, do you want the all the SCCs to have a higher priority than most of the other functions. Or would you like some to be higher, while others are lower?

In the diagram, the thicker arrows indicated the grouped priorities, and the thinner arrows the spread, providing a rough idea of how the priorities can be organized. As you can see, all the other requests have a lower priority than those already mentioned, when not in spread priority scheme. There is just one more option. The user can select any one of the possible requests to have the highest priority. The one selected will always be serviced ahead of any others if it generates a request, effectively jumping the queue!

Having considered the choosing of priorities, the next consideration is how does it work? As previously seen, when an interrupt request is recognized by the processor, it has no idea what caused it. What must happen is that the highest priority request is serviced first, and as we have seen, there could be several pending. The processor could go through some polling routine with a table of priorities, but that would not be efficient. What is required is a method of getting a quick response for the highest priority, unmasked request, and that is obtained by the processor reading the SIU interrupt vector register.

The process of prioritization ensures that a vector number that relates to the highest priority pending interrupt is returned in the sivec register. The processor can use that vector number to jump to the necessary service routine. That service routine will be appropriate for the controller that generated the interrupt request, but as mentioned earlier, there will be work involved in discovering the event to be serviced, deciding which event must be serviced first, and probably organizing the order of multiple events. The events being serviced must also be cleared to enable subsequent ones to be recognized.

Generally, interrupt pending bits are not cleared by the service routine. The service routine must clear the controller event. If no other events are set then the interrupt pending bit will clear, but if other events are set the pending bit remains set and will generate another interrupt request.
### Interrupt configuration Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIU Interrupt Configuration Register (SIUCR)</td>
<td>Defines highest priority and Group or Spread priority</td>
</tr>
<tr>
<td>SIU Interrupt Priority Register (SIUPR)</td>
<td>Selects relative priority for each of IRQ1-5, TMCNT, PIT and PCI.</td>
</tr>
<tr>
<td>CPM High Interrupt Priority Register (CPMHPR)</td>
<td>Selects relative priority for each FCC and MCC.</td>
</tr>
<tr>
<td>CPM Low Interrupt Priority Register (CPMLPR)</td>
<td>Selects relative priority for each SCC.</td>
</tr>
<tr>
<td>SIU External Interrupt Control Register (SIUXR)</td>
<td>Defines pin activity to generate an Interrupt request from IRQ and port C pins</td>
</tr>
</tbody>
</table>

These are the registers employed in configuring the interrupt controller. The SIU Interrupt Configuration Register defines the highest priority request and Group or Spread priority schemes for the XSIU and the YCC options. The SIU Interrupt Priority Register selects the relative priority for each IRQ1-5 input pins, Timer counter, Periodic Interrupt Timer, and PCI interface.

The CPM High Interrupt Priority Register selects relative priority for each FCC and MCC. The CPM Low Interrupt Priority Register selects relative priority for each SCC. The SIU External Interrupt Control Register defines pin activity to generate an Interrupt request from interrupt request and port C pins. To view the registers in detail, click on the one of interest.
### Interrupt controller input and enable Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SIMR-H</strong></td>
<td>Enables selective requests from SIPNR-H to generate a request to the Interrupt Controller.</td>
</tr>
<tr>
<td><strong>SIMR-L</strong></td>
<td>Enables selective requests from SIPNR-H to generate a request to the Interrupt Controller.</td>
</tr>
<tr>
<td><strong>SIPNR-H</strong></td>
<td>Indicates Interrupt requests from Port C pins, IRQ pins, Timer counter, and Periodic Interrupt Timer. PCI controller for devices where it’s present.</td>
</tr>
<tr>
<td><strong>SIPNR-L</strong></td>
<td>Indicates Interrupt requests from all other functions.</td>
</tr>
<tr>
<td><strong>SIVEC</strong></td>
<td>Indicates the source of a recognized Interrupt.</td>
</tr>
</tbody>
</table>

**Reference Manual section 4.3.1**

These are the registers that capture, and enable interrupt requests, and enable the processor to service the appropriate request. The SIU Interrupt Mask Register enables selective requests from the interrupt pending register to generate a request to the Interrupt Controller. It is made up of two 32-bit registers, designated high and low, to handle all the possible requests.

The SIU Interrupt Pending Register indicates interrupt requests from all unmasked events, and is also made up of two 32-bit registers, and aligns with the mask registers. The SIU Interrupt Vector Register indicates the source of a recognized Interrupt by returning the associated vector number.
Responding to an External Interrupt

When the processor responds to an interrupt request, there is a basic process it must follow. First, it must save any necessary information required to return to normal processing. Then, it must establish what caused the interrupt by reading the SIVEC register. The SIVEC register returns a specific value related directly to the controller or function that generated the request. This value always has the two least significant bits clear, so always returns a number that is a factor of four. There are at least a couple of options on how this can be handled.

Remember the registers in dual-port RAM are effectively RAM entries, and the processor could read SIVEC as an 8-bit value. In that case, the number can be used directly as an offset into a table of 32-bit values, where each entry could be a branch to service routine instruction.

Reference Manual section 4.2.4
Responding to an External Interrupt - Routine table

- a) Save required info.
- b) Read SIVEC
- c) place table base_pointer + SIVEC[16] into entry pointer
- d) branch to table entry

SIVEC = 0x0000
  0x0400
  0x0800
  0x0C00
  0x1000
  . . . .
  . . . .

An alternative option is to read SIVEC as a 16-bit value. In that case, the value returned will always have the lower ten bits clear. If this number is used as a offset into a table, each block of the table would span one kilobyte, that could contain 256 instructions. This could be used as a table of service routines. And if a block is not large enough for certain routines, a branch out to a larger block or to several individual routines could be used.