Parallel Input/Output (PIO) and Interrupt

Updated: January 2014

1. **Introduction**

This document explains the PIO core with Altera’s Avalon Memory-Mapped (Avalon MM) interface. This IP can be used to connect to on-chip user logic or to I/O pins such as LEDs, switches, etc.

2. **PIO Core**

Altera provides a set of commonly used I/O peripherals that can be integrated into an embedded system using Qsys integration tool. We will examine the PIO core that can be used to interface with general input and output peripherals.

The PIO core provides data transfer in either input or output (or both) directions. The transfer is done in parallel and it may involve from 1 to 32 bits. The number of bits and the direction of transfer are specified by the user through Altera’s Qsys tool (at the time a Nios II based system is being implemented). A Nios-II processor can interface with these ports by reading and writing register-mapped Avalon MM interface.

2.1. **PIO Core Register Map**

The PIO core has a number of options for customizing general-purpose I/O interfaces. PIO interfaces can be specified as input only, output only, or bidirectional. If bidirectional is selected, then the direction of each pin must be set in the direction register at run-time via software. Input PIO interfaces can also have various interrupt and edge capture capabilities including the capturing of either or both edges and edge or level sensitive interrupt triggers. In general, a PIO interface contains various controlling registers shown in Fig. 1.

Each register is 32 bits long; however, not all bits may be valid. The four main registers have the following purpose:

- **Data**: holds the \( n \)-bit data that are transferred between the PIO interface and the Nios II processor. It can be implemented as an input, output, or a bidirectional register by the Qsys tool.
• **Direction**: defines the direction of transfer for each of the \( n \) data bits when a bidirectional interface is generated. This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.

• **Interrupt-mask**: is used to enable interrupts from the input lines connected to the PIO. This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.

• **Edge-capture**: register indicates when a change of logic value is detected in the signals on the input lines connected to the PIO. A logic ‘1’ indicates a change is detected for that bit. **Writing any value to Edge-capture register clears all bits.**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>R/W</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>data</td>
<td>R</td>
<td>(n-1)</td>
</tr>
<tr>
<td></td>
<td>read access</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write access</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>direction ( (1) )</td>
<td>R/W</td>
<td>Data value currently on PIO inputs.</td>
</tr>
<tr>
<td>2</td>
<td>interruptmask ( (1) )</td>
<td>R/W</td>
<td>Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.</td>
</tr>
<tr>
<td>3</td>
<td>edgecapture ( (1), (2) )</td>
<td>R/W</td>
<td>IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.</td>
</tr>
<tr>
<td>4</td>
<td>outset</td>
<td>W</td>
<td>Specifies which bit of the output port to set.</td>
</tr>
<tr>
<td>5</td>
<td>outclear</td>
<td>W</td>
<td>Specifies which output bit to clear.</td>
</tr>
</tbody>
</table>

**Notes**

(1) This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.

(2) Writing any value to edgecapture clears all bits to 0.

Fig. 1. PIO Core Register Map (source: Altera).

Not all of these registers are generated in a given PIO interface. For example, the Direction register is included only when a bidirectional interface is specified. The Interrupt-mask and Edge-capture registers are included if interrupt-driven input/output is used. The PIO registers are accessible as memory locations. Any base address that has the four least significant bits equal to 0 can be assigned to a PIO. This becomes the address of the Data register. The addresses of the other three registers have offsets of 4, 8, or 12 bytes (1, 2, or 3 32-bit words) from this base address. Note that the offset numbers shown in Fig. 1 are word offset (integer type = 32-bit).
2.2. PIO Core Example

Basic operation for these PIO ports including reading and writing data to the register at the base address (offset of 0 byte) of the component. For example, let’s consider a \textit{nios\_system} that contains:

- an 8-bit input PIO port for the slider switches
- an 8-bit output PIO port for the red leds, and
- a 4-bit input PIO port for the 4 pushbuttons on a DE2-70 board.

Fig. 2 shows the PIO components in the \textit{nios\_system} generated by the Qsys.

**Notes:**
- The base address for the switches PIO, \textit{switches}, is at 0x00011000
- The base address for the red leds PIO, \textit{red\_leds}, is at 0x00011020
- The base address for the pushbuttons, \textit{KEYs}, is at 0x000110b0

**Question:** What is the address of the \textit{Edge-capture} register of the \textit{switches} PIO?

**Question:** What is the address of the \textit{Interrupt-mask} register of the \textit{KEYs} PIO?

Once the \textit{nios\_system} has been appropriately instantiated in the design, user application executed on a Nios-ii processor can directly access peripherals via the control registers. For example, consider the following code snippet:

```c
1. volatile int * SWITCHES_ptr = (int *) 0x00011000;
2. volatile int * RED_LEDS_ptr = (int *) 0x00011020;
3. *(RED_LEDS ptr) = *(SWITCHES ptr);
```

Listing 1
• Line 1 defines a pointer, `SWITCHES_ptr`, that points to the base address of `switches`.
• Line 2 defines a pointer, `RED_LEDS_ptr`, that points to the base address of `red_leds`.
• Line 3 shows a simple assignment operation. The value stored in the register at the base address (pointed to by `SWITCHES_ptr`) of `switches` is read. This value is then written to the register at base address (pointed to by `RED_LEDS_ptr`) of `red_leds`.

PIO Core with Polling Method

Polling is a technique to monitor an I/O port and to trigger an appropriate action when a change is detected. The general idea is that the processor periodically reads the I/O port and determines if there is a change. If there is a change, the processor will execute the subset of code to deal specifically with this change. Otherwise, the processor will continue with normal operations. The change is the edge transition of the external signals which can be configured in Qsys. For example, if we want to detect the falling edge of the pushbuttons (KEYs) in the `nios_system`, we can configure the PIO core as shown in Fig. 3. Note that we can also configure the PIO core so that it will generate an interrupt request when the change occurs.

![Fig. 3. PIO Core configuration.](image-url)
With polling technique, we would check the edge capture register regularly to determine if a falling edge of one of the pushbuttons has been detected. For example, let’s consider the following code snippet:

```c
1. volatile int * SWITCHES_ptr = (int *) 0x00011000;
2. volatile int * RED_LEDS_ptr = (int *) 0x00011020;
3. volatile int * KEYS_ptr = (int *) 0x000110b0;
4. if (*((KEYs_ptr + 0x3) > 0x0)) // edge capture register
5. {
6.    *(RED_LEDS_ptr) = *(SWITCHES_ptr);
7.    *(KEYs_ptr+0x3) = 0x0;
8. }
```

Listing 2

In this example, a value represented by `switches` is assigned to `red_leds` only when one of the pushbuttons is pressed.

- Line 1 defines a pointer, `SWITCHES_ptr`, that points to the base address of `switches`.
- Line 2 defines a pointer, `RED_LEDS_ptr`, that points to the base address of `red_leds`.
- Line 3 defines a pointer, `KEYS_ptr`, that points to the base address of `KEYs`.
- Line 4 examines if any bit in the Edge-capture register of the pushbutton port is set. If it is, the condition `(*(KEYs_ptr + 0x3) > 0x0)` will be true and line 6 and 7 will be executed next. Note that the address of the Edge-capture register is located at (base address + 12 bytes) or (base address + 3 words). Since `KEYS_ptr` is defined as an integer pointer, word offset can be used as shown in line 4.
- Line 6 shows a simple assignment operation as in previous example.
- Line 7 indicates that 0 (or any value) is written to the Edge-capture register which clears all bits. This clearing process has to be done before a new change can be detected.

**Question:** What is value of the edge capture register of the KEYS PIO core after one of the pushbuttons is pressed?
KEY(3) pressed: Edge-capture register =
KEY(2) pressed: Edge-capture register =
KEY(1) pressed: Edge-capture register =
KEY(0) pressed: Edge-capture register =

Question: How would you modify line 4 from the previous example to check if KEY(2) was pressed?

Question: How would you modify line 4 from the previous example to check if KEY(2), KEY(1), or KEY(0) was pressed?

Question: What is value of the Edge-capture register of the KEYS PIO core after the following statement is executed?
   *(KEYs_ptr+0x3) = 0x0F;

Question: If line 3 in the previous example was written as
   volatile char* KEYS_ptr = (char*) 0x000110b0;

   Does the program still work as intended?
3. NIOS-II Interrupt with C

The NIOS-II processor supports non-vector interrupts. It means that when an interrupt occurs, the program jumps to a fixed memory location (specified by user in Qsys tool at system generation time). Interrupt is an exception caused by an explicit request signal from an external device. When the internal interrupt controller is implemented, a peripheral device can request hardware interrupt by asserting one of the Nios II processor’s 32 interrupt-request inputs, irq0 through irq31. A hardware interrupt is generated if and only if all three of these conditions are true:

- The PIE bit of the status control register is one.
- An interrupt-request input, irqn, is asserted.
- The corresponding bit n of the ienable control register is one.

3.1. Nios-II Control Registers

The Nios-II control registers (32 bit-wide) involved in interrupt processing are:

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Register Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>status</td>
<td>This register controls/holds the state of the NIOS-ii processor. This register contains the PIE bit.</td>
</tr>
<tr>
<td>1</td>
<td>estatus</td>
<td>This register holds a copy of status register during non-break interrupt processing.</td>
</tr>
<tr>
<td>2</td>
<td>bstatus</td>
<td>This register holds a copy of status register during break interrupt processing.</td>
</tr>
<tr>
<td>3</td>
<td>ienable</td>
<td>This register holds interrupt-enable bits for irq0 through irq31.</td>
</tr>
<tr>
<td>4</td>
<td>ipending</td>
<td>This register holds pending-interrupt bits that correspond to irq0 through irq31.</td>
</tr>
</tbody>
</table>

Upon hardware interrupt, the processor clears the PIE bit to zero, disabling further interrupts, and performs the interrupt service routine to respond to the interrupt. The value of the ipending control register shows which interrupt requests (IRQ) are pending. By peripheral design, an IRQ bit is guaranteed to remain asserted until the processor explicitly responds to the peripheral and clears the interrupt.

In general, in order to successfully implement an interrupt-driven system based on a NIOS-processor, users need to make sure that:

1. Hardware controllers request an interrupt by asserting irqn. This is mainly done by Qsys tool at system generation time.
2. The memory locations for “reset vector” and “exception vector” for the CPU are defined. This is done by Qsys tool at system generation time.

3. The enable bit(s) is (are) set in the interrupt enable control register which is a part of the memory-mapped interface for that peripheral. This is done by the user program. This step enables interrupt for the peripheral.

4. The corresponding bit(s) is (are) set in the ienable (register 3) register. For example, if irq0 is the desired interrupt event, then bit 0 in the ienable register should be set to ‘1’. This is done by the user program. This step enables interrupt for the specific peripherals in the NIOS-II side.

5. The global interrupt enable (PIE or bit 0) bit is set in the status register (register 0). This is done by the user program. This step enables the overall interrupt system in the NIOS-II processor.

6. The appropriate functions are provided to handle the reset and interrupt events. These functions will be fixed and given by your instructor.

7. The user interrupt service routines are provided to handle a specific interrupt event. This is done in user program.

Fig. 4 shows the relationship between ipending, ienable, PIE, and the generation of an interrupt.

Fig. 4. Relationship between ipending, ienable, PIE, and hardware interrupt (source: Altera).
3.2. **Interrupt Example with PIO Core**

It’s easier to explain the steps to set up an interrupt-driven system with an example. So, let’s consider a nios_system with a Nios-II processor, on-chip memory, and 3 PIO cores:
- A 4-bit PIO output core *keys* that is connected to 4 pushbuttons (i.e. KEYS).
- An 8-bit PIO input core *switches* that is connected to 8 slider switches.
- An 8-bit PIO output core *leds* that is connected to 8 red LEDs.

The steps to set up an interrupt-capable PIO cores are:
1. Configuring PIO cores to detect falling/rising edges and to generate interrupt request (Qsys tool).
2. Enabling interrupt in the PIO cores (C program).
3. Enabling interrupt handling system in the Nios-II processor (C program).
4. Developing interrupt service routine (ISR) to handle interrupt events.

**a. Configuring PIO Cores**

The input core *keys* is configured to detect the falling edges of the pushbuttons and to generate an interrupt request. An example of the configuration window is shown in Fig. 3.

The input core *switches* is configured to detect the rising edges of the switches and to generate an interrupt request.

The nios_system created by Qsys tool is shown in Fig. 5.

![Fig. 5. An example of a nios_system.](image)

Notes:
- The interrupt level for the PIO core *switches* is **IRQ0** as shown in the IRQ column
- The interrupt level for the PIO core *keys* is **IRQ1** as shown in the IRQ column
b. Enabling Interrupt in the KEYs PIO Core

Enabling a core to generate an interrupt request is done in the user program. This step may be slightly different for different peripherals. For a PIO core, this is achieved by setting the corresponding bit(s) in the interrupt-mask register of the core. For example, the code snippet to enable interrupt for KEY(3) is:

```
#define KEYS_BASE_ADR 0x03010
...
volatile int * KEYS_ptr = (int *) KEYS_BASE_ADR;
*(KEYS_ptr + 2) = 0x08;
```

Listing 3

Question: How would you enable interrupt for KEY(3) and KEY(2)?

c. Enabling the Interrupt System in the Nios-II Processor

Enabling the interrupt system in the Nios-II processor is achieved by a two-step procedure:

- Setting the corresponding interrupt enable bit (determined by the IRQn level created in the Qsys tool) in the ienable control register (register 3) of the Nios-II processor. For example, code snippet to enable irq1:

  ```
  __builtin_wrctl(3, 2); //enable irq1 in the ienable register
  ```

- Setting the global interrupt enable PIE bit in the status control register (register 0) of the Nios-II processor. Note that this step should be done after all cores have been enabled. For example:

  ```
  __builtin_wrctl(0, 1); //Set the PIE bit in the status register
  ```

Note that the built-in function `__builtin_wrctl(v, r)` is implemented with machine instructions to write a value $v$ to a control register $r$. We will use this function to set up the interrupt system.
**Question:** Modify the code to enable interrupt for both the *keys* and *switches* PIO cores.

Recall that the interrupt levels for *keys* and *switches* PIO cores are *IRQ1* and *IRQ0* respectively.

d. Handling the Interrupt Event

An *interrupt* is a special condition known as an *exception* that requires a processor’s immediate attention. The Nios-II processor provides a simple, non-vectored interrupt controller. It means that when an interrupt occurs, the execution is transferred to the same known exception address for all types of interrupts. This exception address is defined when the *nios_system* is generated in the Qsys tool.

**the_exception function:**
A function (*the_exception*) to deal with an exception which includes external interrupts is shown below. The function represents a general exception handler that can be used with any C program. It includes assembly language code to save the contents of the registers and to call an interrupt service routine named *interrupt_handler*. If the interrupt service routine is named *interrupt_handler*, this function can be used as is without modification.

```c
/*
 * The assembly language code below handles CPU exception processing. This code should not be modified; instead, the C language code in the function interrupt_handler() can be modified as needed for a given application.
 */

void the_exception (void) __attribute__ ((section (".exceptions")));

void the_exception (void)

Exceptions code. By giving the code a section attribute with the name ".exceptions" we allow the linker program to locate this code at the proper exceptions vector address. This code calls the interrupt handler and later returns from the exception.
```
***********/
{
asm ( "set noat" );  // Magic, for the C compiler
asm ( "set nobreak" );  // Magic, for the C compiler
asm ( "subi sp, sp, 128" );
asm ( "stw et, 96(sp)" );
asm ( "rdctl et, ctl4" );
asm ( "beq et, r0, SKIP_EA_DEC" ); // Interrupt is not external
asm ( "subi ea, ea, 4" ); /* Must decrement ea by one instruction for external interrupts, so that the interrupted instruction will be run */
asm ( "SKIP_EA_DEC:" );
asm ( "stw r1, 4(sp)" ); // Save all registers
asm ( "stw r2, 8(sp)" );
asm ( "stw r3, 12(sp)" );
asm ( "stw r4, 16(sp)" );
asm ( "stw r5, 20(sp)" );
asm ( "stw r6, 24(sp)" );
asm ( "stw r7, 28(sp)" );
asm ( "stw r8, 32(sp)" );
asm ( "stw r9, 36(sp)" );
asm ( "stw r10, 40(sp)" );
asm ( "stw r11, 44(sp)" );
asm ( "stw r12, 48(sp)" );
asm ( "stw r13, 52(sp)" );
asm ( "stw r14, 56(sp)" );
asm ( "stw r15, 60(sp)" );
asm ( "stw r16, 64(sp)" );
asm ( "stw r17, 68(sp)" );
asm ( "stw r18, 72(sp)" );
asm ( "stw r19, 76(sp)" );
asm ( "stw r20, 80(sp)" );
asm ( "stw r21, 84(sp)" );
asm ( "stw r22, 88(sp)" );
asm ( "stw r23, 92(sp)" );
asm ( "stw r25, 100(sp)" ); // r25 = bt (skip r24 = et, because it is saved above)
asm ( "stw r26, 104(sp)" ); // r26 = gp
// skip r27 because it is sp, and there is no point in saving this
asm ( "stw r28, 112(sp)" ); // r28 = fp
asm ( "stw r29, 116(sp)" ); // r29 = ea
asm ( "stw r30, 120(sp)" ); // r30 = ba
asm ( "stw r31, 124(sp)" ); // r31 = ra
asm ( "addi fp, sp, 128" );

asm ( "call interrupt_handler" ); // Call the C language interrupt handler
asm ( "ldw r1, 4(sp)" ); // Restore all registers
asm ( "ldw r2, 8(sp)" );
asm ( "ldw r3, 12(sp)" );
asm ( "ldw r4, 16(sp)" );
asm ( "ldw r5, 20(sp)" );
asm ( "ldw r6, 24(sp)" );
asm ( "ldw r7, 28(sp)" );
asm ( "ldw r8, 32(sp)" );
asm ( "ldw r9, 36(sp)" );
asm ( "ldw r10, 40(sp)" );
asm ( "ldw r11, 44(sp)" );
asm ( "ldw r12, 48(sp)" );
asm ( "ldw r13, 52(sp)" );
asm ( "ldw r14, 56(sp)" );
asm ( "ldw r15, 60(sp)" );
asm ( "ldw r16, 64(sp)" );
asm ( "ldw r17, 68(sp)" );
asm ( "ldw r18, 72(sp)" );
asm ( "ldw r19, 76(sp)" );
asm ( "ldw r20, 80(sp)" );
asm ( "ldw r21, 84(sp)" );
asm ( "ldw r22, 88(sp)" );
asm ( "ldw r23, 92(sp)" );
asm ( "ldw r24, 96(sp)" );
asm ( "ldw r25, 100(sp)" ); // r25 = bt
asm ( "ldw r26, 104(sp)" ); // r26 = gp
// skip r27 because it is sp, and we did not save this on the stack
asm ( "ldw r28, 112(sp)" ); // r28 = fp
asm ( "ldw r29, 116(sp)" ); // r29 = ea
asm ( "ldw r30, 120(sp)" ); // r30 = ea
asm ( "ldw r31, 124(sp)" ); // r31 = ra
asm ( "addi sp, sp, 128" );
asm ( "eret" );
**interrupt_handler function:**

The `interrupt_handler` code determines which exception has occurred, by determining which bit in the `pending` register is set. It then performs the appropriate operations or calls the appropriate function to respond to the interrupt. In this example, the `interrupt_handler` function determines if KEY(3) is pressed (causing an interrupt request `IRQ1` to be asserted) and calls `keys_isr()` function to respond to this interrupt. Note that code in `keys_isr()` function can be nested inside the `interrupt_handler` function also.

```c
/* **********************************************
Interrupt Service Routine
   Determines what caused the interrupt and calls the appropriate subroutine.
   ipending - Control register 4 which has the pending external interrupts
***********************************************/
void interrupt_handler(void)
{
    int ipending;
    ipending = __builtin_rdctl(4);  //Read the ipending register
    if ((ipending & 0x02) > 0) //If irq1 is high, call keys_isr()
    {
        keys_isr();
    }
    return;
}
```

Listing 5

Note the syntax for
```
ipending = __builtin_rdctl(4);
```
The `ipending` register (or register 4) is read and stored in the variable `ipending`.

**Question:** If both `keys` and `switches` PIO cores are enabled to generate an interrupt, add code to handle the `switches` interrupt.

```c
void interrupt_handler(void)
{
    int ipending;
    ipending = __builtin_rdctl(4);  //Read the ipending register
    if ((ipending & 0x02) > 0) //If irq1 is high, call keys_isr()
    {
        keys_isr();
    }
}
```
Listing 6

Example of keys_isr() function:
This function performs tasks that deal with the interrupt when KEY(3) is pressed. Each peripheral should have an interrupt service routine (isr) to deal with the interrupt events. In this example, the keys_isr() function reads and assigns the switch value to red LEDs.

```
void keys_isr(void)
{
    int * red_leds = (int *) LEDS_BASE_ADR;
    volatile int * pushbuttons = (int *) KEYS_BASE_ADR;
    volatile int * switches = (int *) SWITCHES_BASE_ADR;

    *(red_leds) = *(switches);
    *(pushbuttons+3) = 0;
    return;
}
```

Listing 7

the_reset function:
Reset is another special exception but not related to the interrupts. A function (the_reset) to deal with a reset event is shown below. The function provides a simple reset mechanism by performing a branch to the main program. Note that this function includes nested assembly language in a C function. This function can be used as is without modification.

```
/* The assembly language code below handles CPU reset processing */
void the_reset (void) __attribute__ ((__section__ (".reset")));

/**************************************************************************************
*************
Reset code. By giving the code a section attribute with the name ".reset" we allow the linker program to locate this code at the proper reset vector address. This code just calls the main program.
***************************************************************************************/
4. Using Altera Monitor Program with Interrupt

Note that when you create an Altera Monitor Program project for your example/lab/project and if interrupt is used, you will need to reserve space for the exception/interrupt handler function at the top of the memory space. You can do this by specifying the offset value for .text section and .data section. A value of 0x400 will be enough as shown in Fig. 6.

Fig. 6. Memory settings with offset for interrupt/exception handler function.