

LatticeMico32 GPIO

The LatticeMico32 GPIO is a general-purpose input/output core that provides a memory-mapped interface between a WISHBONE slave port and general-purpose I/O ports. The I/O ports can connect to either on-chip or off-chip logic.

Version

This document describes the 3.1 version of the LatticeMico32 GPIO.

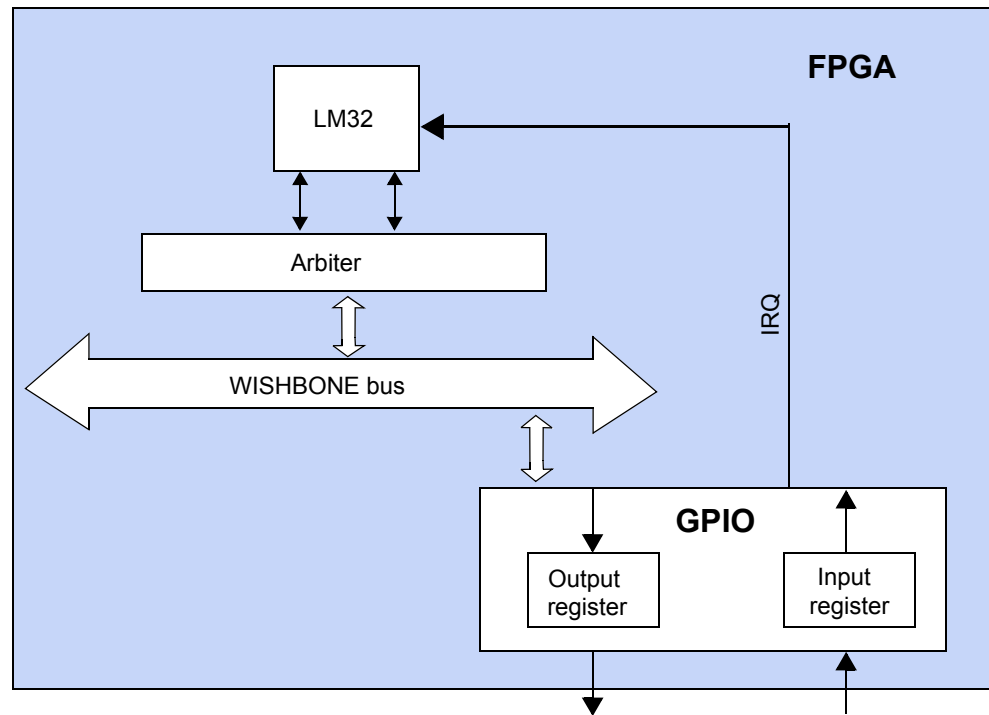
Features

The LatticeMico32 GPIO includes the following features:

- ◆ WISHBONE B.3 interface
- ◆ Four transfer port types: tristate, input, output, independent input/output
- ◆ Interrupt request (IRQ) generation on level-sensitive or edge-sensitive input
- ◆ Hardware interrupt mask register

Figure 1 shows the deployment of the LatticeMico32 GPIO within the FPGA and its input and output ports.

Figure 1: Using GPIO to Connect LatticeMico32 to FPGA I/O Pins



For additional details about the WISHBONE bus, refer to the *LatticeMico32 Processor Reference Manual*.

Functional Description

The LatticeMico32 GPIO creates an interface between the LatticeMico32 RISC processor and a simple bit-level control interface. The control bits can be connected to either internal or external logic. Control of the input or output pins is managed using four memory-mapped registers. The memory-mapped registers control reading and writing the input/output bits, tristating the I/O bits, interrupt masking, and an “edge event” status register.

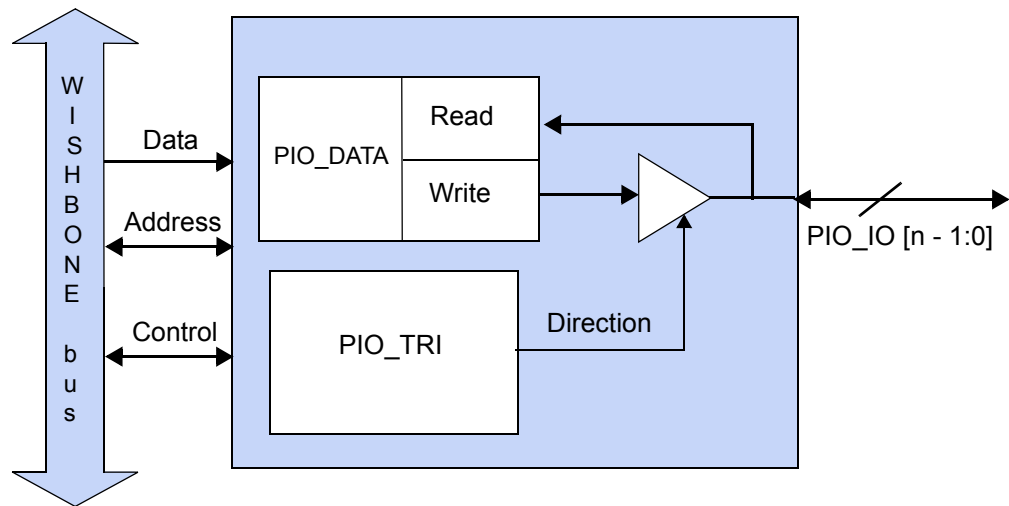
The number, width, and behavior of the control registers change on the basis of the configuration of the GPIO block. The GPIO block can be configured with inputs only, outputs only, or both inputs and outputs. A bidirectional mode with tristate control is also provided for bidirectional I/O pins that are available on the device. Bidirectional mode I/O can only be connected to FPGA I/O pins. They are not available for use with on-chip logic.

Reading the data register returns the value that is present on the input ports. See “EBR Resource Utilization” on page 16 for details. Writing the PIO_DATA register (see Table 4 on page 11) affects the value that is driven to the output

ports. Because these ports are independent, reading the data register does not return the previously written data. Bidirectional ports, which are tristate, are the exception to this behavior.

Figure 2 is an internal block diagram of the GPIO showing its bidirectional ports. It shows how the PIO_TRI register (see Table 4) determines whether the PIO_DATA register is used to transfer data in or out.

Figure 2: LatticeMico32 GPIO Core Usage with Bidirectional Ports



Input Data Path

Figure 7 on page 14 shows how the GPIO's master ports read the data in the internal register. The input port is connected to a D flip-flop that has an asynchronous reset controlled by RST_I. The flip-flop is continuously clocked by CLK_I. The output is connected directly to the PIO_DATA read register, as shown in Figure 2. The value read from the PIO_DATA register is always delayed one CLK_I cycle from the current state of the PIO_IN port. The PIO_DATA read cycle does not prevent the input flip-flop from changing state.

Output Data Path

The PIO_OUT is driven by D-type flip-flops. The D-type flip-flops are updated when the processor writes to the PIO_DATA register. The data appears on the output pins when the WISHBONE acknowledge signal (GPIO_ACK_O) goes high.

Tristate Control

After the microprocessor writes to the PIO_TRI register, the pins are tristated when the WISHBONE acknowledge signal (GPIO_ACK_O) goes high. The data becomes valid in the same cycle as the GPIO_ACK_O.

Edge Capture

The edge capture register is used to produce an interrupt, which corresponds to the input bit where the edge was detected. The edge capture register is read to identify which input bit caused the interrupt. If an edge capture interrupt is not cleared, subsequent edge events on the same input bit will be ignored. See “IRQ Generation” on page 5 for more information on interrupt request priorities.

You can configure the GPIO to capture edges on the input ports. The type of edges—which can include low-to-high transitions, high-to-low transitions, or both—are defined and configured at the time that the LatticeMico32 platform is generated. It is not possible to change the edge capture behavior at run time. When an edge is detected by the input, the condition is indicated in the edge capture register. The bit position corresponding to the input pin transitions from 0 to 1.

The edge is detected when the input signal transitions between two WISHBONE clock cycles (CLK_I).

Clearing a bit—that is, setting it to zero in the edge capture register—clears the corresponding bit in the edge capture register. Setting a bit in the interrupt mask register resets the corresponding bit in the edge capture register if it was set.

Port Width and Port Type Settings

The Width and Port Type settings define the basic functionality of the GPIO instance.

Port Width

The number of I/O pins that can be controlled by the GPIO block ranges from a minimum of one to a maximum of thirty-two. In the case of the mixed independent input/output ports, the number of input ports can be different than the number of output ports.

Port Type

The GPIO instance can be configured for output only, input only, mixed independent input/output, and shared bidirectional input/output.

Output Port Type For the output port type, the PIO ports can drive output only (PIO_OUT).

Input Port Type For the input port type, the PIO ports can capture input only (PIO_IN).

Independent Input and Output Port Type For the independent input and output port type, the input and output ports are separate unidirectional buses. The number of input pins can be set independently of the number of output ports. The PIO_DATA register is wide enough to manage the larger of the “Input Width” or “Output Width.” The input and output bits overlap in the PIO_DATA register, starting at the least-significant bit. The “Input Width” PIO_IN ports and “Output Width” PIO_OUT ports are created for connection to logic external to the GPIO block.

Tristate Port Type For the tristate port type, each PIO bit shares one device pin for driving and capturing data. The direction of each pin is individually selectable. An I/O pin becomes an input when the corresponding PIO_TRI register bit is cleared, that is, 0. Whenever the WISHBONE RST_I signal is asserted, the PIO_TRI register is cleared, forcing all PIO_IO pins to be inputs.

IRQ Generation

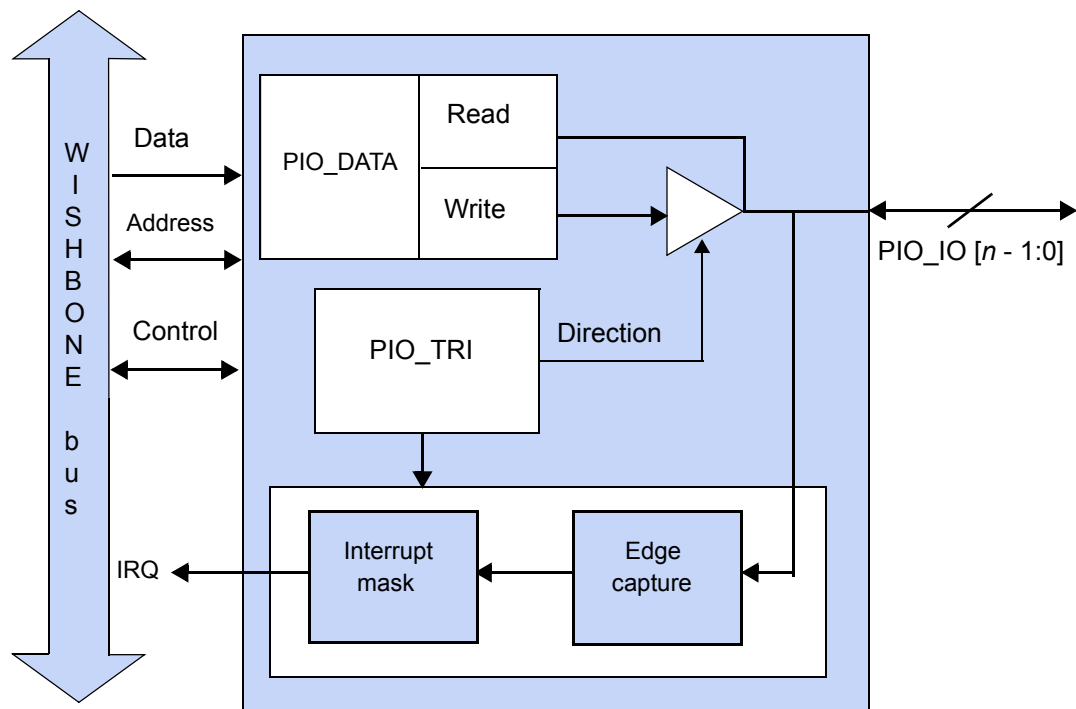
You can configure the LatticeMico32 GPIO to generate an interrupt request (IRQ) on level-sensitive or edge-sensitive input conditions.

- ◆ Level-sensitive – An IRQ is generated whenever a specific input is high and interrupt requests are enabled for that input in the IRQ_MASK register. The input pin should be held high until the interrupt condition is cleared.
- ◆ Edge-sensitive – An IRQ is generated whenever a specific bit in the edge capture register is high and interrupt requests are enabled for that bit in the IRQ_MASK register. The type of edge must be specified at platform generation: positive edge, negative edge, or either edge. Clearing a bit—that is, setting it to zero in the edge capture register—clears the corresponding bit in the edge capture register. Setting a bit in the interrupt mask register resets the corresponding bit in the edge capture register if it was set.

Any I/O in a GPIO can cause an interrupt request. The IRQ_MASK register of the GPIO defines which I/Os can cause an interrupt request.

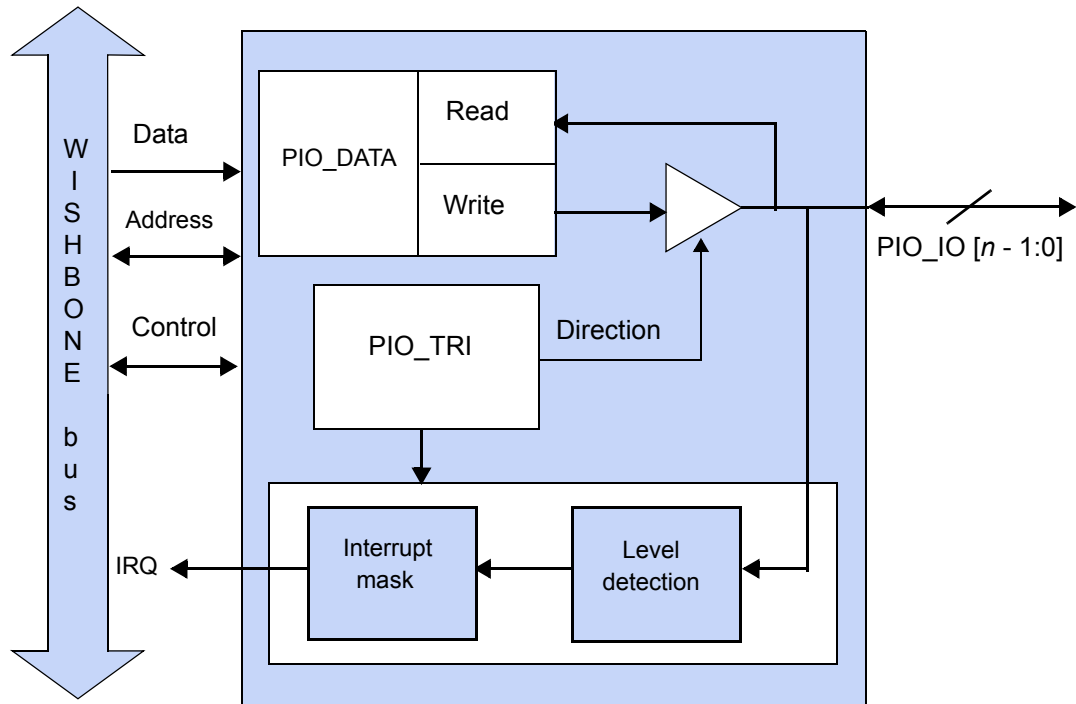
When the IRQ mode is turned off, the IRQ_MASK register does not exist.

Figure 3 shows a bidirectional LatticeMico32 GPIO similar to that shown in Figure 2 but with the ability to generate interrupt requests. The GPIO in Figure 3 can generate interrupt requests as a signal switches from 0 to 1 or 1 to 0 (edge capture).

Figure 3: LatticeMico32 GPIO Usage with IRQ (Edge Capture)

For level-sensitive input, as shown in Figure 4, only active high is directly supported.

Figure 4: LatticeMico32 GPIO Usage with IRQ (Level Triggered)



Configuration

The following sections describe the graphical user interface (UI) parameters, the hardware description language (HDL) parameters, and the I/O ports that you can use to configure and operate the LatticeMico32 GPIO.

UI Parameters

Table 1 shows the UI parameters available for configuring the LatticeMico32 GPIO through the Mico System Builder (MSB) interface.

Table 1: GPIO UI Parameters

Dialog Box Option	Description	Allowable Values	Default Value
Instance Name	Specifies the name of the GPIO instance.	Alphanumeric and underscores	gpio
Base Address	Specifies the base address for the device. The minimum byte alignment is 0X80.	0X80000000 – 0XFFFFFF80 If other components are included in the platform, the range of allowable values will vary.	0X80000000

Port Types

Table 1: GPIO UI Parameters (Continued)

Dialog Box Option	Description	Allowable Values	Default Value
Output Ports Only	Specifies the transfer mode of PIO ports as output only.	selected not selected	selected
Input Ports Only	Specifies the transfer mode of PIO ports as input only.	selected not selected	not selected
Tristate Ports	Specifies the transfer mode of PIO ports as tristate only.	selected not selected	not selected
Both Input and Output	Specifies the transfer mode of PIO ports as input and output.	selected not selected	not selected
Port Width			
Data Width	Specifies the width of the I/O port, in bits.	1 to 32	1
Input Width	Specifies the input data bus width for an independent input/output GPIO, in bits.	1 to 32	1
Output Width	Specifies the output data bus width for an independent input/output GPIO, in bits.	1 to 32	1
IRQ Mode			
IRQ Mode	When selected, provides IRQ signal output when a specified event occurs on input ports	selected not selected	not selected
Level	When selected, generates an IRQ whenever a specific input is high and interrupts have been enabled for that input in the IRQ-MASK register.	selected not selected	not selected
Edge	When selected, generates an IRQ whenever a specific bit in the edge capture register is high and interrupts have been enabled for that bit in the IRQ-MASK register.	selected not selected	selected
Edge Response			
Either Edge	When selected, generates an IRQ on either low-to-high or high-to-low transitions.	selected not selected	not selected
Positive Edge	When selected, generates an IRQ on low-to-high transitions.	selected not selected	selected
Negative Edge	When selected, generates an IRQ on high-to-low transitions.	selected not selected	not selected

HDL Parameters

Table 2 describes the parameters that appear in the HDL.

Table 2: GPIO HDL Parameters

Parameter Name	Description	Allowable Values
INPUT_PORTS_ONLY	A value of 1 defines the transfer mode for PIO ports as input only.	0 1
OUTPUT_PORTS_ONLY	A value of 1 defines the transfer mode for PIO ports as output only.	0 1
TRISTATE_PORTS	A value of 1 defines the transfer mode for PIO ports as tristate only.	0 1
BOTH_INPUT_AND_OUTPUT	A value of 1 defines the transfer mode for PIO ports as input and output.	0 1
DATA_WIDTH	Defines the width of the I/O port.	1 to 32
INPUT_WIDTH	Defines the input data bus width for an independent input/output GPIO.	1 to 32
OUTPUT_WIDTH	Defines the output data bus width for an independent input/output GPIO.	1 to 32
IRQ_MODE	A value of 1 establishes IRQ_MODE, providing IRQ signal output when a specified event occurs on input ports.	0 1
LEVEL	With a value of 1, an IRQ is generated whenever a specific input is high and interrupts have been enabled for that input in the IRQ MASK register.	0 1
EDGE	With a value of 1, an IRQ is generated whenever a specific bit in the edge capture register is high and interrupts have been enabled for that bit in the IRQ MASK register.	0 1
EITHER_EDGE_IRQ	With a value of 1, an IRQ is generated on either low-to-high or high-to-low transitions.	0 1
POSE_EDGE_IRQ	With a value of 1, an IRQ is generated on low-to-high transitions.	0 1
NEGE_EDGE_IRQ	With a value of 1, an IRQ is generated on high-to-low transitions.	0 1

I/O Ports

Table 3 describes the input and output ports of the LatticeMico32 GPIO.

Table 3: GPIO I/O Ports

Port Name	Active	Direction	Initial State	Description
System Clock and Reset				
CLK_I	—	I	X	System clock
RST_I	High	I	X	System reset
WISHBONE Interface				
GPIO_ADR_I	—	I	X	Address input array, the address generated by the master
GPIO_CYC_I	High	I	X	Cycle input. When asserted, it indicates that a bus cycle is in progress.
GPIO_DAT_I	—	I	X	Data input array, valid for a write request
GPIO_LOCK_I	High	I	X	Lock input. If asserted, the current cycle becomes uninterruptible.
GPIO_SEL_I	High	I	X	Select input array, which indicates where the valid data is expected on a data bus.
GPIO_STB_I	High	I	X	Strobe input. When asserted, indicates that the SLAVE is selected.
GPIO_WE_I	—	I	X	Write signal. Value of 1 is used for a write and 0 for a read.
GPIO_ACK_O	High	O	0	Acknowledge output. When asserted, indicates normal cycle termination.
GPIO_DAT_O	—	O	0	Data output array
PIO Interface Ports				
PIO_IN	—	I	X	Appears in input mode or both input and output mode. The GPIO's number of input bits is configurable.
PIO_OUT	—	O	0	Appears in output mode or both input and output mode. The GPIO's number of output bits is configurable.
PIO_IO	—	I/O	0/X	Appears in tristate mode only. Each GPIO bit shares one device pin for driving and capturing data. The direction of each pin is individually selectable. The PIO_IO is an input when the corresponding PIO_TRI register bit is cleared (0).
Other Auto-connected Internal Signals				
IRQ_O	High	O	0	Interrupt request outputs. The GPIO can be configured to generate an IRQ on certain input conditions.

User Impact of Initial State

For the output port, the initial state is low after reset.

Register Definitions

The LatticeMico32 GPIO includes the registers shown in Table 4.

Table 4: Register Map

Register Name	Offset	31-0
I/O data	0x00	PIO_DATA
Tristate control	0x04	PIO_TRI
IRQ mask	0x08	IRQ_MASK
Edge capture	0x0C	EDGE_CAPTURE

Table 5 through Table 8 provide details about each register in the LatticeMico32 GPIO.

Table 5: PIO_DATA Register Bit Definition

Register Name	Bit	Access Mode	Description
PIO_DATA	DATA_WIDTH -1:0	Read/write	<ul style="list-style-type: none"> ◆ R – Input only I/O data latched and read ◆ W – Output only I/O data asserted ◆ R/W – Bidirectional I/O data read/written

Table 6: PIO_TRI Register Bit Definition

Register Name	Bit	Access Mode	Description
PIO_TRI	DATA_WIDTH -1:0	Read/write	PIO read/write tristate enable mask Setting a bit to 1 enables the driver for that output port.

Table 7: IRQ_MASK Register Bit Definition

Register Name	Bit	Access Mode	Description
IRQ_MASK	INPUT_WIDTH -1:0	Read/write	Setting a bit to 1 enables interrupt generation for that input port. Setting a bit to 0 disables interrupt generation. A change from 0 to 1 clears the corresponding EDGE_CAPTURE register.

Table 8: EDGE_CAPTURE Register Bit Definition

Register Name	Bit	Access Mode	Description
EDGE_CAPTURE	INPUT_WIDTH - 1:0	Read/write	A bit that is set to 1 indicates that an edge capture event has occurred for that input port. The bit is cleared by writing a 0 to the corresponding bit in the EDGE_CAPTURE register or by disabling and then enabling the corresponding bit in the IRQ_MASK register. After an edge is detected, the EDGE_CAPTURE bit is held at a 1 until explicitly cleared.

The structure shown in Figure 5 depicts the register map layout for the GPIO component. The elements are self-explanatory and are based on the register, as shown in Table 4. This structure, which is defined in the MicoGPIO.h header file, enables you to directly access the GPIO registers, if desired. It is used internally by the device driver for manipulating the GPIO.

Figure 5: GPIO Register Map Structure

```

/*
 * GPIO REGISTER MAPPING
 */

typedef struct st_MicoGPIO_t{
    /* read/write: r-only for in-only GPIO, w-only for out-only
    GPIO, r/w for tristates */
    volatile unsigned int data;

    /* read/write: tristate enable register for tristate GPIOs
    */
    volatile unsigned int tristate;

    /* read/write: sets irq mask for interrupt-enabled GPIOs */
    volatile unsigned int irqMask;

    /* read/write: applicable to GPIOs with edge-capture
    capability */
    volatile unsigned int edgeCapture;
}MicoGPIO_t;

```

Timing Diagrams

The timing diagrams featured in Figure 6 through Figure 10 show the timing of the GPIO's WISHBONE and external signals.

Figure 6 shows how the GPIO's master ports update the data in the internal register.

Figure 6: WISHBONE Master Writes Data in the Internal Register

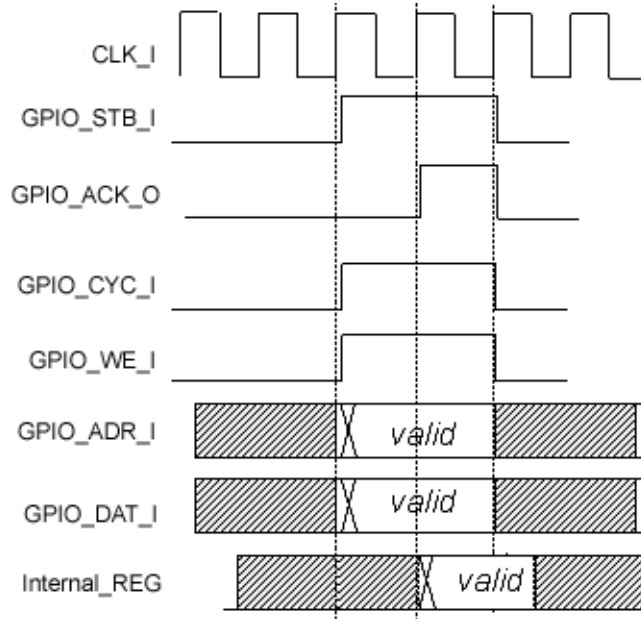


Figure 7 shows how the GPIO's master ports read the data in the internal register.

Figure 7: WISHBONE Master Reads Data in the Internal Register

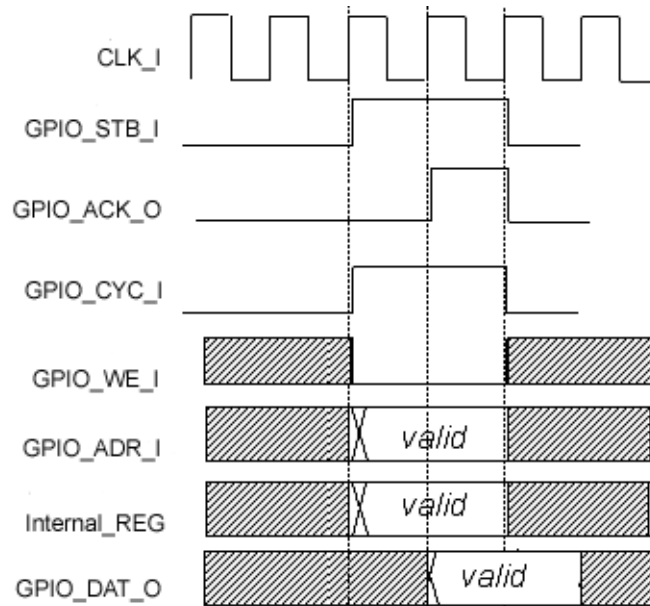


Figure 8 shows how the GPIO generates interrupt requests when a signal is high or low.

Figure 8: IRQ Generation (Level)

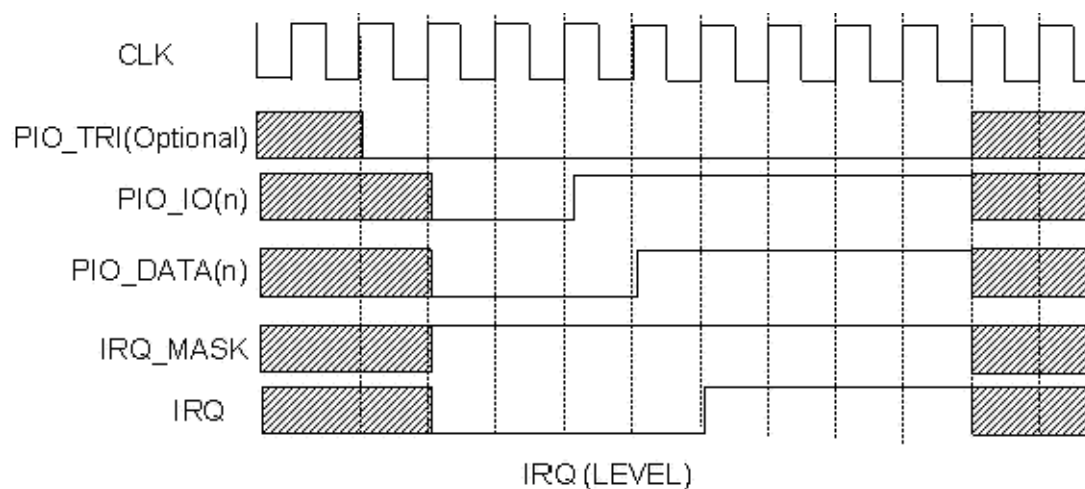


Figure 9 shows how the GPIO generates interrupt requests when the PIO_DATA signal transitions from low to high. After an edge is detected, the Edge_Capture bit is held at a 1 until cleared in the IRQ_MASK register.

Figure 9: IRQ Generation (Rising Edge)

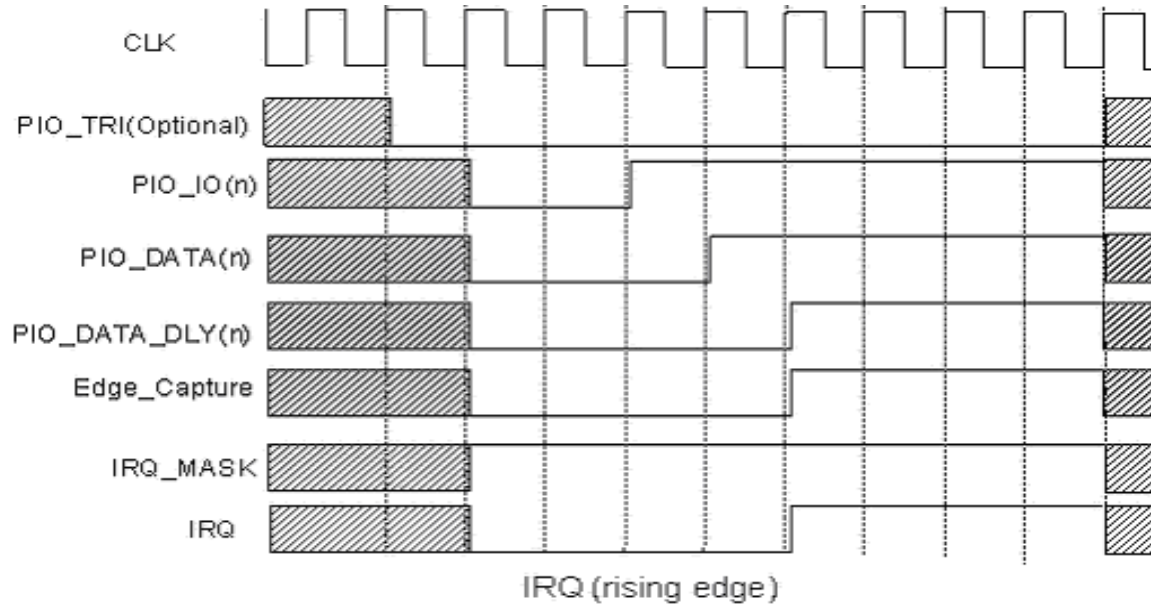
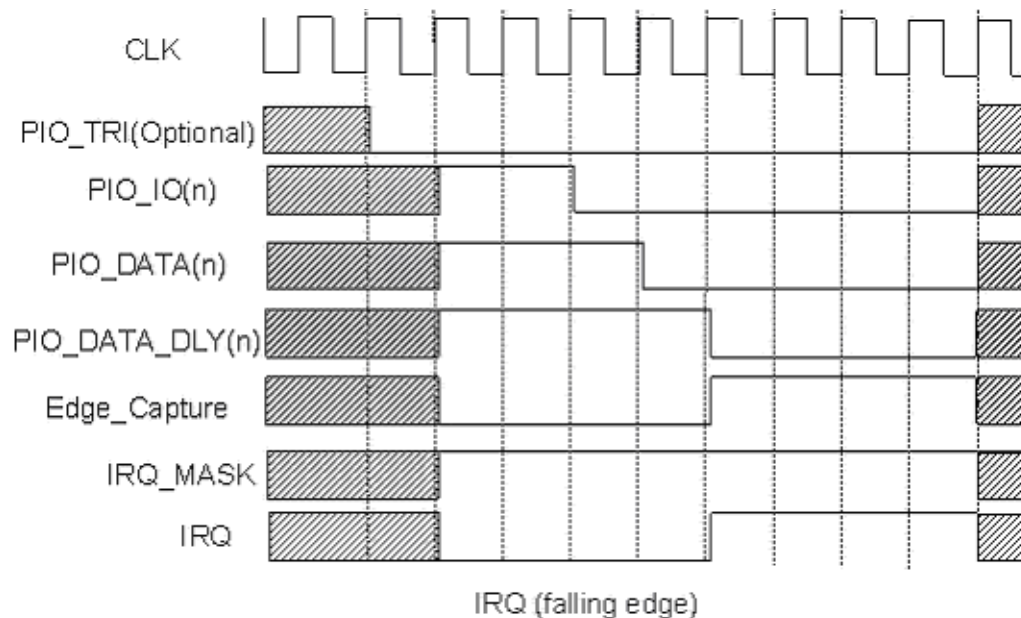


Figure 10 shows how the GPIO generates interrupt requests when the PIO_DATA signal transitions from high to low.

Figure 10: IRQ Generation (Falling Edge)



EBR Resource Utilization

The LatticeMico32 GPIO uses no EBRs.

Software Support

This section describes the software support provided for the LatticeMico32 GPIO component, including the GPIO usage model, its relationship with the LatticeMico32 microprocessor, the device driver, and services. It also provides software usage examples.

The support routines for the GPIO component are for use in a single-threaded environment. If used in a multi-tasking environment, re-entrance protections must be provided.

Usage Model

The GPIO component, as the name suggests, is general-purpose. This component can be used for the following:

- ◆ An output device (output mode only)
- ◆ An input device (input mode only)
- ◆ An output and input device (output and input mode), which means using separate input and output pins
- ◆ A bidirectional device, which means that the same sets of pins are used for input and output by controlling the direction of the pins

Additionally, this device is capable of generating interrupt requests when the following are detected on an input port:

- ◆ High level
- ◆ Positive edge
- ◆ Negative edge

The usage scenario depends on the end-user application and does not fit within a well-known usage model. To handle interrupt from a GPIO that is configured to trigger interrupts on edge detection, you can disable the interrupt and perform processing later, since the edge-capture register remains set on detecting the appropriate condition. Or, you can perform the processing in the ISR and write a zero to the corresponding bit of the edge-capture register to clear the condition. In the former case, the edge-capture register is cleared when the interrupt is enabled again; in the latter case, the edge capture register is explicitly cleared by your writing a zero to the corresponding bit of the edge capture register.

Effect of Endianness

The LatticeMico32 microprocessor is a big-endian processor. It is important to understand the impact of endianness when the microprocessor interacts with a GPIO instance using less than a 32-bit memory access. The software performs 32-bit memory accesses when using “unsigned int” or “signed int” variables.

For example, assume that you have a design that has a GPIO, and the Mico System Builder (MSB) has assigned a base address of 0x80000100. The GPIO data register for input and output of data is located at byte offset 0 from the base address, and it is accessible by reading the contents of the base address. For a 32-bit GPIO, access would be as simple as the code shown in Figure 11:

Figure 11: Access to a 32-Bit GPIO

```
unsigned int GPIOData;  
GPIOData = *(volatile unsigned int *)0x80000100;
```

Now assume that this GPIO is really an 8-bit GPIO, so that it uses only 8 bits for data input/output. From a programmer’s perspective, only bits 7 through 0 are significant. Although it would be tempting to treat the 0x80000100 location as a pointer to an unsigned character, as shown in Figure 12, the results would be incorrect.

Figure 12: Incorrect Access to an 8-Bit GPIO

```
/*  
 * INCORRECT ACCESS TO AN 8-BIT GPIO,  
 * FOR LatticeMico32 (Big Endian)  
 */  
unsigned char GPIOData;  
GPIOData = *(volatile unsigned char*)0x80000100;
```

Since the microprocessor is big-endian, this code would not yield the correct results, because it would actually receive bits 31 through 24 instead of bits 7 through 0.

The best alternative is to treat the 0x80000100 address as a pointer to an unsigned integer location and read it as shown in Figure 13:

Figure 13: Correct Access of an 8-Bit GPIO

```
/*
 * Read unsigned int at location 0x80000100
 * and ignore the upper bits.
 */
unsigned int GPIOData;
GPIOData = *(volatile unsigned int *)0x80000100;

/*
 * If you must pass this data on to unsigned char,
 * use the following:
 */
unsigned char GPIOData;
GPIOData = (unsigned char) (*(unsigned int *)0x80000100);
```

Device Driver

This section describes the type definitions for instance-specific structures and the GPIO device context structure.

Instance-Specific Structures

The MSB managed build process instantiates a unique structure per instance of the GPIO in the platform. These instances are defined in DDStructs.c. The information for these instance-specific structures is filled in by the managed build process, which extracts GPIO component-specific information from the platform definition file. The members should not be manipulated directly because the structure is used exclusively by the device driver. You can retrieve a pointer to the instance-specific GPIO device context structure by using the MicoGetDevice function call of the LatticeMico32 device lookup service. Refer to the *LatticeMico32 Software Developer User Guide* for more information on the device lookup service.

GPIO Device Context Structure

This structure, shown in Figure 14, contains GPIO component-specific information and is dynamically generated in the DDStructs.h header file. This information is largely filled in by the MSB managed build process, which extracts the GPIO component-specific information from the platform definition file. The members should not be manipulated directly, because this structure is for exclusive use by the device driver.

Figure 14: GPIO Device Context Structure

```
typedef struct st_MicoGPIOctx_t {
    const char*    name;
    unsigned int   base;
    unsigned int   intrLevel;
    unsigned int   output_only;
    unsigned int   input_only;
    unsigned int   in_and_out;
    unsigned int   tristate;
    unsigned int   data_width;
    unsigned int   input_width;
    unsigned int   output_width;
    unsigned int   intr_enable;
    DeviceReg_t    lookupReg
    void *         prev;
    void *         next;
} MicoGPIOctx_t;
```

Table 9 describes the parameters of the GPIO device context structure shown in Figure 14.

Table 9: GPIO Device Context Structure Parameters

Parameter	Data Type	Description
name	const char *	GPIO instance name
base	unsigned int	MSB-assigned base address

Table 9: GPIO Device Context Structure Parameters

Parameter	Data Type	Description
intrLevel	unsigned int	MSB-assigned interrupt, if interrupts are used. If interrupts are not used, this value is greater than 31. If interrupts are used, the value is 0-31.
output_only	unsigned int	This value is 1 if the GPIO is configured as output only. Otherwise, it is 0.
input_only	unsigned int	This value is 1 if the GPIO is configured as input only. Otherwise, it is 0.
in_and_out	unsigned int	This value is 1 if the GPIO is configured as input and output. Otherwise, it is 0.
tristate	unsigned int	This value is 1 if the GPIO is configured as a tristate device. It is 0 if the GPIO is not configured as a tristate device.
data_width	unsigned int	This value represents the instance-configured data width. It should be treated as a valid value only if the GPIO is configured as input only, output only, or tristate only.
input_width	unsigned int	This value represents the input width if the GPIO is configured for input and output mode.
output_width	unsigned int	This value represents the output width if the GPIO is configured for input and output mode.
intr_enable	unsigned int	This value is set to 0 if the GPIO is not configured to generate interrupts. Otherwise, the value is 1.
lookupReg	DeviceReg_t	Used by the device driver to register the GPIO component instance with the LatticeMico32 lookup service. Refer to the <i>LatticeMico32 Software Developer User Guide</i> for a description of the DeviceReg_t data type.
prev	void *	Used internally by the lookup service
next	void *	Used internally by the lookup service

Note

You may need to access the GPIO device registers directly, but some of these registers are write-only. Implementing shadow registers in RAM can be an effective way to replace this missing capability. Figure 15 provides an example of “shadow” register code for handling write-only registers in LatticeMico32 System.

Figure 15: Example Shadow Register Code Fragment

```

MicoGPIOCtx_t *pGPIO_context;
MicoGPIO_t      gpioState, *pGPIO;

// The GPIO is an OUTPUT only instance
pGPIO = (pMicoGPIO_t *) (pGPIO_context->base);
// initialize the "shadow" copy in RAM
gpioState.gpioData = 0x1fff;
// write the "shadow" values out to the I/O pins
pGPIO->gpioData = gpioState.gpioData;

// do a read of the "shadow" value and then clear the lsb
gpioState->gpioData &= 0xFE;
// write the new value to the I/O pins
pGPIO->gpioData = pGpioState->gpioData;

// ...

```

Functions

Since the GPIO is a general-purpose device and does not fit a well-defined usage scenario, there are no predefined functions. However, there are numerous macros provided in the MicoGPIO.h file that allow easy access to the various GPIO registers using the GPIO context structure. These macros are listed here, and their usage is illustrated in “Software Usage Examples” on page 22.

Figure 16: Macros for Accessing GPIO Registers

```

/*
 * MACROS FOR ACCESSING GPIO REGISTERS
 *
 * NOTE: For the macros, the following rules apply:
 * X is a pointer to a valid MicoGPIOCtx_t structure.
 * Y is an unsigned int variable.
 */

/* reads data register */
#define MICO_GPIO_READ_DATA(X,Y) \
    (Y)=((volatile MicoGPIO_t *) ((X)->base))->data

/* writes data-register */
#define MICO_GPIO_WRITE_DATA(X,Y) \
    ((volatile MicoGPIO_t *) ((X)->base))->data=(Y)

/* reads tristate register */
#define MICO_GPIO_READ_TRISTATE(X,Y) \
    (Y) = ((volatile MicoGPIO_t *) ((X)->base))->tristate

/* writes tristate register */
#define MICO_GPIO_WRITE_TRISTATE(X,Y) \
    ((volatile MicoGPIO_t *) ((X)->base))->tristate = (Y)

```

Figure 16: Macros for Accessing GPIO Registers

```

/* reads irq-mask register */
#define MICO_GPIO_READ_IRQ_MASK(X,Y) \
    (Y) = ((volatile MicoGPIO_t *) (X)->base)->irqMask

/* writes irq-mask register */
#define MICO_GPIO_WRITE_IRQ_MASK(X,Y) \
    ((volatile MicoGPIO_t *) (X)->base)->irqMask = (Y)

/* reads edge-capture register */
#define MICO_GPIO_READ_EDGE_CAPTURE(X,Y) \
    (Y) = ((volatile MicoGPIO_t *) (X)->base)->edgeCapture

/* writes to the edge-capture register */
#define MICO_GPIO_WRITE_EDGE_CAPTURE(X,Y) \
    ((volatile MicoGPIO_t *) (X)->base)->edgeCapture = (Y)

```

Services

The GPIO device driver registers GPIO instances with the LatticeMico32 lookup service, using their instance names for device names and "GPIODevice" as the device type.

For more information about using the lookup service, refer to the *LatticeMico32 Software Developer User Guide*.

Software Usage Examples

This section provides two code examples that demonstrate how to access the GPIO registers.

Using the GPIO Register Structure

The code example shown in Figure 17 shows how to locate a GPIO device that is instantiated in the platform and how to directly access the registers using the GPIO register structure.

Figure 17: Locating a GPIO and Accessing Its Registers

```

/* Fetch GPIO instance named "LED" */
volatile MicoGPIO_t *pGPIO;
MicoGPIOCtx_t *leds = (MicoGPIOCtx_t *)MicoGetDevice("led");
if(leds == 0) {
    /* failed to find a component named "leds" */
    return(-1);
}

/* get access to the GPIO registers */
pGPIO = (volatile MicoGPIO_t *) (leds->base);

/* write 0x80 to the data register */
pGPIO->data = 0x80;

```

Using Provided Macros

The code example shown in Figure 18 shows how to locate a GPIO device that is instantiated in the platform and how to directly access the data register using the macros provided in MicoGPIO.h header file.

Figure 18: Locating a GPIO and Accessing Its Data Register

```
/* Fetch GPIO instance named "LED" */
unsigned int iValue;
MicoGPIOCtx_t *leds = (MicoGPIOCtx_t *)MicoGetDevice("led");
if (leds == 0) {
    /* failed to find a component named "leds" */
    return(-1);
}

/* write 0x80 to the data register */
MICO_GPIO_WRITE_DATA(leds, 0x80);

/* read back the value in the data register */
MICO_GPIO_READ_DATA(leds, iValue);
```
