ZIO Python API

Tutorial

1.1, May 2009

Table of Contents

Chapter 1. Introduction

ZIO is a IO framework for rapid product development. And as such it comes along with an API that can be used to access the IO interfaces provided by the board. This document shows how to use the API, to do simple tasks, which can then be used as a reference for building complex applications.

From the API's stand point, the ZIO motherboard has 5 modules.

1. GPIO
2. I2C
3. Sensor
4. PWM
5. SPI

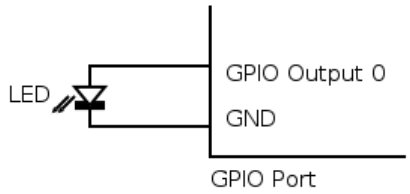
These modules on the motherboard are accessed through an agent software running on the motherboard. The agent software communicates with the PC and performs appropriate actions on the modules.

Chapter 2. API Examples

1. Controlling LEDs

LEDs can be easily connected to GPIO output pins as shown in the following circuit. When the GPIO output pin is set to high, the LED turns on and when the pin is set to low, the LED turns off. The code to blink the LED is listed below.

Figure 2.1. LED Circuit



Listing 2.1. LED Blink, Python Code

```
import zio ❶
import time

agent = zio.Agent("/dev/ttyUSB0") ❷
gpio = zio.GPIO(agent) ❸
led_pin = 0 ❹

while True:
    # Turn on the LED
    gpio.write_output_pin(led_pin, 1) ❺
    time.sleep(1)

    # Turn off the LED
    gpio.write_output_pin(led_pin, 0) ❻
    time.sleep(1)
```

❶ The ZIO API resides in a Python module called `zio`. To use the API the module has to be imported.

❷ The `Agent` class is used to establish a communication link between the host and the board. The device file that corresponds to the USB serial port of the ZIO board is passed as argument. Under Linux, it is usually `/dev/ttyUSBx`. Under Windows, it is `COMx`.

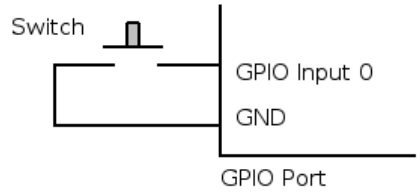
❸ The `GPIO` class is used to control the GPIO module. The `Agent` object is passed as argument to the constructor.

❹ The `GPIO` output pin to which the LED is connected.

❺❻ The pin state can be controlled using the `write_output_pin` method of the `GPIO` class. The first argument is the pin to control. The second argument is the value to be set on the pin.

2. Interfacing Switches

Switches can be connected to GPIO output pins as shown in the following circuit. When the switch is not pressed, the input pin is internally pulled up to 5V, and reads high. When the switch is pressed, the input pin is grounded, and reads low. The code to read the switch status is listed below.



Listing 2.2. Switch Status, Python Code

```
import zio
import time

agent = zio.Agent("/dev/ttyUSB0")
gpio = zio.GPIO(agent)
switch_pin = 0 ❶

while True:
    state = gpio.read_input_pin(switch_pin) ❷
    if state:
        print "Switch Off"
    else:
        print "Switch On"

    time.sleep(0.5)
```

❶ The GPIO input pin to which the switch is connected.
❷ The `read_input_pin` method of the `GPIO` class can be used to read the pin state. The pin no. is passed as argument. The method returns the state of the pin.

The problem with the above approach is that the state of the pin has to be periodically polled. If done at high rates, this can cause excessive load on the host CPU. This can be avoided by using a callback mechanism. A callback function is registered with the GPIO module. Whenever a change occurs in any of the pins, the callback is invoked by the `GPIO` class.

Listing 2.3. Switch Status with Notifications, Python Code

```
import zio

def input_changed(event): ❶
    global switch_pin

    if event.pin != switch_pin: ❷
        return
    if event.value:
        print "Switch Off"
    else:
        print "Switch On"

agent = zio.Agent("/dev/ttyUSB0")
gpio = zio.GPIO(agent)
switch_pin = 0
gpio.add_change_handler(input_changed) ❸

agent.wait_for_events() ❹
```

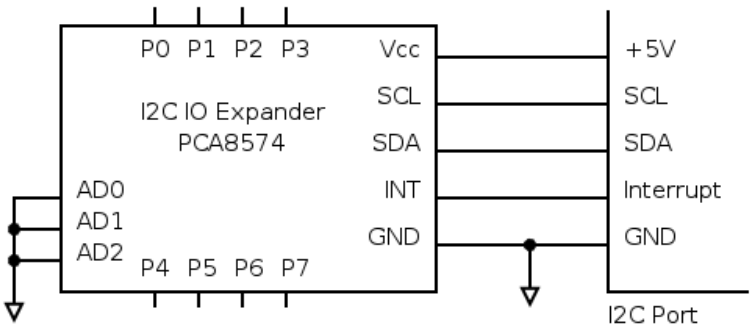
❶ The callback function takes an event object as argument. In the case of GPIO, it is a `GPIOChangeEvent` object. The event object contains information about the event — the pin in which the change occurred, the current state of the pin, etc.
❷ The callback will be called when any of the input pins change. This check filters out changes in other pins.
❸ The callback is registered with the `GPIO` object using the `add_change_handler` method. The function is passed as an argument to the method.
❹ The `wait_for_events` method on the `Agent` object is used to wait for events in an infinite loop. And when an event occurs, the corresponding callback is invoked.

3. Interfacing I2C Devices

I2C is a bi-directional two-wire (data and clock) serial bus that provides a communication link between integrated circuits. Examples of simple I2C-compatible devices found in embedded systems include EEPROMs, thermal sensors, and real-time clocks.

I2C IO Expander (PCF8574) provides 8 digitial IO lines that can be controlled, through the I2C bus. The IO Expander can be interfaced to the I2C port as show in the following circuit. The code to access the I2C IO Expander is listed below.

Figure 2.2. I2C IO Expander Circuit



Listing 2.4. I2C IO Expander, Python Code

```
import zio
import sys

agent = zio.Agent("/dev/ttyUSB0")
i2c = zio.I2C(agent) ❶
dev = 0x20 ❷

i2c.config(100) ❸
try:
    i2c.write(dev, [0xFF]) ❹
    data = i2c.read(dev, 1) ❺
except I2CNoAckException, e:
    print e
    sys.exit(1)

print "IO Expander Input: 0x%02X" % data[0];
```

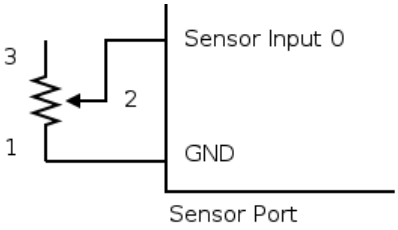
❶ The `I2C` class is used to control I2C module. The `Agent` object is passed as argument to the constructor.
❷ The 7-bit device address of the I2C device can be obtained from the data sheet, and the hardware configuration of the pins `A0`, `A1` and `A2`. In this case it happens to be `0x20`.

❸ The `config` method of the `I2C` class is used to configure the bus clock frequency. The frequency is specified in kHz.
❹ The `write` method of the `I2C` class is used to write bytes to the I2C device. The device address is specified as the first argument. The list of bytes to be written is specified as the second argument.
❺ The `read` method of the `I2C` class is used to read bytes from the I2C device. Th device address is specified as the first argument. The no. of bytes to be read is specified as the second argument. The method returns a list of bytes read from the device.

4. Interfacing a Potentiometer

A single turn potentiometer in a variable resistor connection can be used in volume control applications. The potentiometer is interfaced to the Sensor port as shown in the following circuit. The internal pull-up and the potentiometer are in voltage divider configuation. The voltage drop across the potentiometer is measured by the ADC. The code to read the voltage from the sensor input is given below.

Figure 2.3. Potentiometer Circuit



Listing 2.5. Potentiometer, Python Code

```
import zio
import time

agent = zio.Agent("/dev/ttyUSB0")
sensor = zio.Sensor(agent) ❶
pot_pin = 0 ❷

while True:
    value = sensor.read_pin(pot_pin) ❸
    print "Sensor Value: %1.2f" % value

    time.sleep(0.5)
```

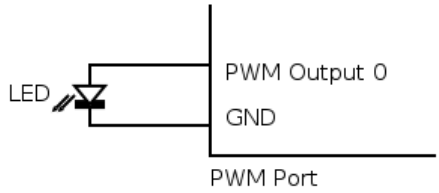
❶ The `Sensor` class is used to control Sensor module. As with other module objects, the `Agent` object is passed as argument to the constructor.
❷ The Sensor input pin to which the pot. is connected.
❸ The `read_pin` method of the `Sensor` class can be used to get the voltage at the pin. The pin no. is passed as argument to the method. The function returns the voltage on the pin as a floating point value. In case the raw ADC value is required, `read_pin_raw` method can be used. The method returns a value between `0x0000` and `0xFFFF`.

Note: Even though the ADC is 10-bit, the value is oversampled to 16-bits, to make the interface future proof.

5. Controlling LED Brightness

The brightness of an LED can be controlled by driving the LED using a PWM signal. PWM is a digital control technique wherein the processor adjusts the duty cycle of a sequence of fixed-width pulses. The LED can be interfaced to the PWM port as shown in the following circuit. When the duty cycle increases, the ON period of the PWM signal increases and the LED brightness increases. The code to control the LED brightness is given below.

Figure 2.4. LED Brightness Circuit



Listing 2.6. LED Brightness Control, Python Code

```
import zio
import time

agent = zio.Agent("/dev/ttyUSB0")
pwm = zio.PWM(agent) ❶
pins = [0] ❷

pwm.set_freq(pins, 2) ❸
pwm.set_duty(pins, 0) ❹
pwm.start(pins) ❺

while (1):
    for i in range(0, 100, 3):
        pwm.set_duty(pins, i)
        time.sleep(0.01)
```

❶ The `PWM` class is used to control PWM module. As with other module objects, the `Agent` object is passed as argument to the constructor.

❷ The PWM output pin to which the LED is connected. The PWM API accepts a list of pins, so that the PWM parameters of multiple pins can be set simultaneously.

❸ The PWM freq is set using the `set_freq` method of the `PWM` class. The list of pins is passed as the first argument. The frequency in kHz is passed as the second argument.

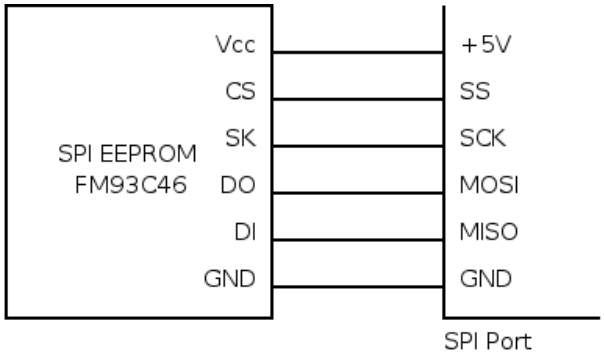
❹ The PWM duty is set using the `set_duty` method of the `PWM` class. The list of pins is passed as the first argument. The duty cycle in percentage is passed as the second argument.

❺ The PWM signal generation is started using the `start` method. And can be stopped using the `stop` method.

6. Interfacing SPI Devices

Serial Peripheral Interface (SPI) is an inexpensive chip interconnection bus, popular on circuit boards. SPI devices like the SPI EEPROM (93C46) can be interfaced to the SPI port as show in the following circuit. The code to access the SPI EEPROM is given below.

Figure 2.5. SPI EEPROM Circuit



Listing 2.7. SPI EEPROM, Python Code

```
import zio

agent = zio.Agent("/dev/ttyUSB0")
spi = zio.SPI(agent) ❶
gpio = zio.GPIO(agent) ❷
ss_pin = 4

freq = spi.config(100,
                  zio.SPI.CPOL_IDLE_LOW,
                  zio.SPI.CPHASE_LEAD_EDGE,
                  zio.SPI.ENDIAN_MSB_FIRST) ❸

gpio.write_output_pin(ss_pin, 1) ❹

try:
    addr = 0x2
    read = spi.write_read([ 0x03, addr, 0x00, 0x00 ]) ❺
finally:
    gpio.write_output_pin(ss_pin, 0) ❻

print "EEPROM has 0x%02X%02X at address 0x%04X" % (read[2], read[3], addr)
```

❶ The `SPI` class is used to control SPI module. The `Agent` object is passed as argument to the constructor.

❷ The `GPIO` class is used for controlling the slave select pin. The slave select pins available on the SPI port are GPIO output pins 4 and 5.

❸ The `config` method of the `SPI` class is used to configure the bus clock frequency, the clock polarity, the clock phase, and the data endianess.

❹❻ The slave select signal is made high while accessing the chip, using the GPIO `write_output_pin` method.

❺ The `write_read` method of the `SPI` class is used to write / read bytes to / from the slave. The bytes to be written is specified as argument. An equal no. of bytes is returned as a list. Here a 16-bit value is read from address `0x2`.