RoBoIO 1.8 for Python Software Development Introduction

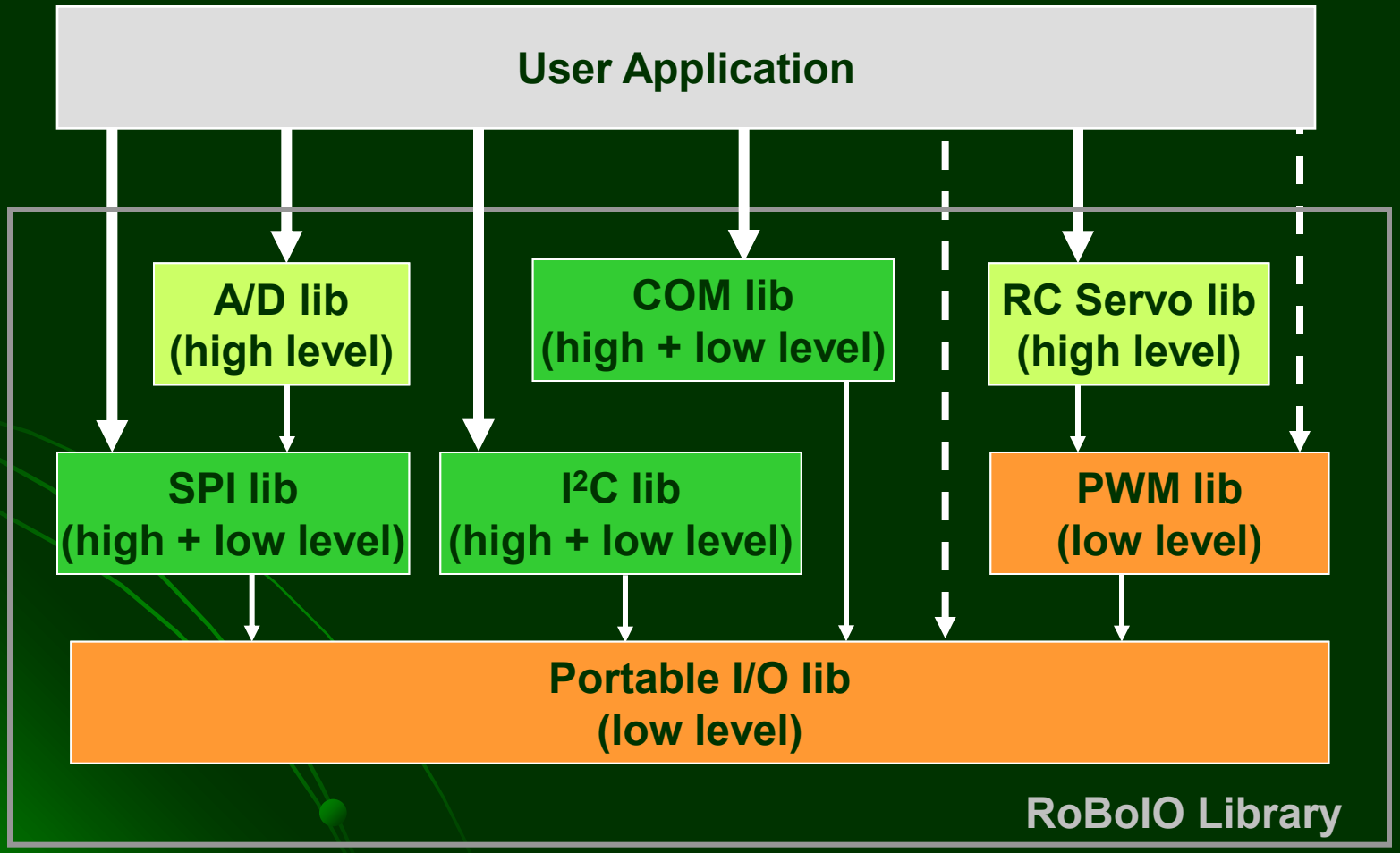
DMP Electronics Inc.
Robotics Division
June 2011

Overview

RoBoIO Library

- A open-source library for RoBoard's unique I/O functions
  - Free for academic & commercial use
- Supported I/O functions
  - PWM (Pulse-Width Modulation)
  - A/D (Analog-to-Digital Converter)
  - SPI (Serial Peripheral Interface)
  - I^2C (Inter-Integrated Circuit Interface)
  - GPIO (General-Purpose Digital I/O)
  - RC servo control (KONDO, HiTEC, …)

Architecture



Usage Overview

- Import `RoBoIO_Python` to use the RoBoIO library
- Call `roboio_SetRBVer(rb_ver)` to set your RoBoard correctly
  - select `rb_ver` = `RB_100` or `RB_100RD` or `RB_110` or `RB_050` according to your RoBoard version

```
From RoBoIO_Python import *
……
if __name__ == "__main__":
    roboio_SetRBVer(…)
    ……
    # use API of RoBoIO
    # library here
    ……
```

Usage Overview

- Error reporting of RoBoIO API
  - When any API function fails, you can always call

    `roboio_GetErrMsg()`

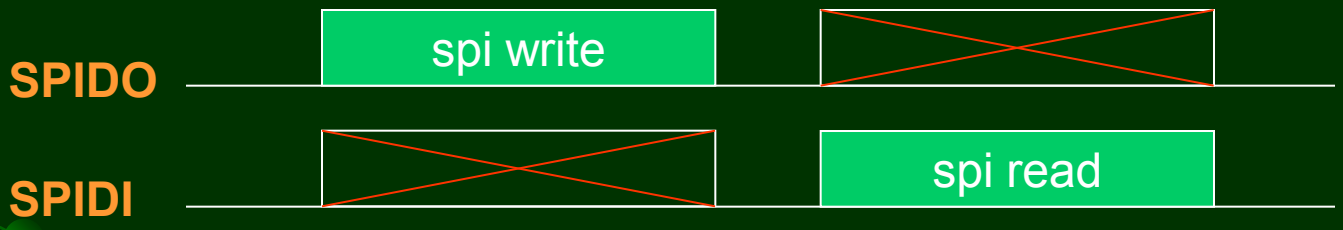
    to get the error message.
  - Example

```
……
if rcservo_Init(…) == False:
    print “Fail to initialize RC Servo lib!!!”
    print “Error message: ”, roboio_GetErrMsg()
    exit()
……
```

SPI lib

RoBoard H/W SPI Features & Limits

- Dedicated to SPI flash
- Half-Duplex

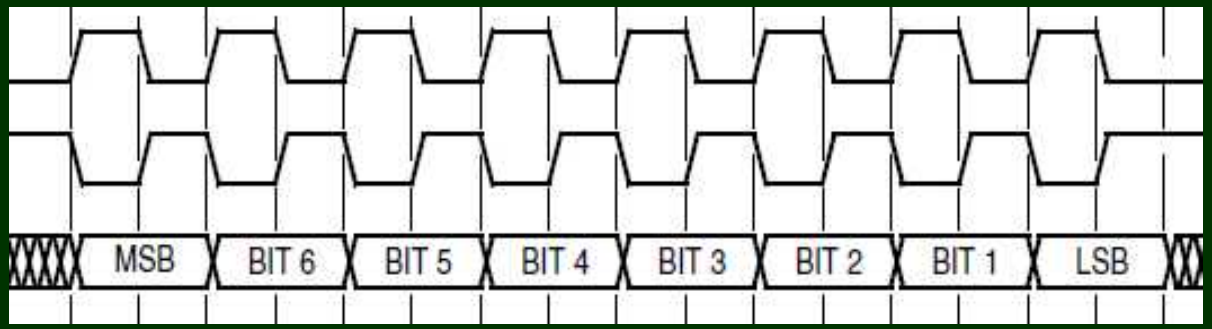


- Support only high-speed devices
  - Max: 150 Mbps
  - Min: 10 Mbps

RoBoard H/W SPI Features & Limits

- **Support only two clock modes**
  - **CPOL = 0, CPHA = 1 Mode**
  - **CPOL = 1, CPHA = 1 Mode**

CPOL = 0
CPOL = 1



- **See http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus for more information about SPI clock modes.**

RoBoard H/W SPI Features & Limits

- **Remarks**
  - **On RB-110 & RB-050, the native SPI can only be used internally to access the on-board A/D.**
  - **If you need SPI interface on RB-110, use RB-110's FTDI General Serial Port (COM6).**
    - **Refer to the application note: RB-110 SPI How-To for more information.**

Usage Overview

```
if spi_Init(clock_mode):
    ……
    val = spi_Read() #read a byte from SPI bus
    spi_Write(0x55)  #write a byte (0x55) to SPI bus
    ……
    spi_Close() #close SPI lib
```

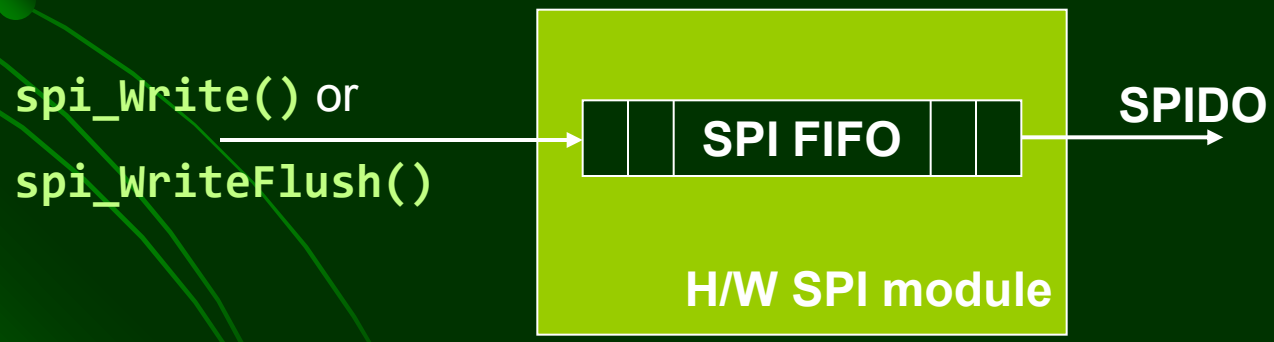
- `clock_mode` can be, e.g.,
  - `SPICLK_10000KHZ` (10 Mbps)
  - `SPICLK_12500KHZ` (12.5 Mbps)
  - `SPICLK_21400KHZ` (21.4 Mbps)
  - `SPICLK_150000KHZ` (150 Mbps)
- See the wrapper source for all available clock modes.

SPI-Write Functions

- Two different SPI-write functions:

  `spi_Write()` vs. `spi_WriteFlush()`

  - All data are written to SPI FIFO, and then transferred by Hardware.



SPI-Write Functions

- Two different SPI-write functions: (cont.)
  - `spi_Write()` does not wait transfer completion.
    - Faster
    - But must be careful about timing issue
    - Can call `spi_FIFOFlush()` to flush SPI FIFO
  - `spi_WriteFlush()` waits that SPI FIFO becomes empty.

SPISS Pin

- Control of the **SPISS** pin of RB-100/100RD
  - `spi_EnableSS()`: set **SPISS** to 0
  - `spi_DisableSS()`: set **SPISS** to 1
- **SPISS** is usually used for turning on/off SPI devices
  - If need more than one **SPISS** pin, simulate them using RoBoard's GPIO
    - For GPIO, refer to the section of RC Servo lib.

Software-Simulated SPI

- From v1.6, RoBoIO includes S/W-simulated SPI functions to support low-speed SPI devices.
- Features of S/W-simulated SPI
  - Max Speed: ~160Kbps
  - Full-Duplex

  - All SPI clock modes supported
    - For an explanation of SPI clock modes, see http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

Software-Simulated SPI

- Usage overview

```
if spi_InitSW(clock_mode, clock_delay):
    ……
    # Half-Duplex read/write
    val1 = spi_Read() #read a byte from SPI bus
    spi_Write(0x55) #write a byte (0x55) to SPI bus
    # Full-Duplex read/write
    val2 = spi_Exchange(0xaa)
                    #write a byte (0xaa) & read a byte from
                    #SPI bus at the same time
    ……
    spi_CloseSW() #close S/W-simulated SPI
```

Software-Simulated SPI

- Usage overview (cont.)
  - `clock_mode` can be
    - `SPIMODE_CPOL0 + SPIMODE_CPHA0`
    - `SPIMODE_CPOL0 + SPIMODE_CPHA1`
    - `SPIMODE_CPOL1 + SPIMODE_CPHA0`
    - `SPIMODE_CPOL1 + SPIMODE_CPHA1`
  - `clock_delay` can be any unsigned integer to control S/W-simulated SPI clock speed.
    - If `clock_delay` = 0, the clock speed is about 160Kbps.

A/D lib

RoBoard A/D Features

- Employ ADI AD7918
  - 10-bit resolution & 1M samples per second
- Share RoBoard's SPI bus
  - When accessing A/D, signals appear on SPICLK, SPIDO, SPIDI pins of RB-100/100RD.
  - So be careful about bus conflict if you have devices attached to SPI bus of RB-100/100RD.
    - Disable your SPI devices (using, e.g., SPISS pin of RB-100/100RD) when accessing A/D.

Usage Overview

```
if spi_Init(SPICLK_21400KHZ):
    ……
    val = adc_ReadCH(channel) #channel = integer 0 ~ 7
    ……
    spi_Close()
```

- To use the 8-channel A/D, we must initialize SPI lib first.
  - SPI clock must ≤ 21.4 Mbps
- Only provides the usual functions of AD7918
  - Refer to AD7918 datasheet if you want to extend A/D lib.

Usage Overview

- If need more detailed control, call `adc_ReadChannel()` instead:

```
if spi_Init(SPICLK_21400KHZ):
    ……
    val = adc_ReadChannel(channel, #channel = 0 ~ 7
                          ADCMODE_RANGE_2VREF,
                          ADCMODE_UNSIGNEDCODING)
    ……
    spi_Close()
```

Usage Overview

- Input-voltage range:
  - `ADCMODE_RANGE_2VREF`: 0V ~ 5V
    - allow higher voltage
  - `ADCMODE_RANGE_VREF`: 0V ~ 2.5V
    - allow higher resolution
- A/D value range:
  - `ADCMODE_UNSIGNEDCODING`: 0 ~ 1023
  - `ADCMODE_SIGNEDCODING`: –512 ~ 511
  - min value ⇨ lowest voltage, max value ⇨ highest voltage
- Remarks: `adc_ReadCH()` uses `ADCMODE_RANGE_2VREF` and `ADCMODE_UNSIGNEDCODING` as default settings.

Batch Mode

- `adc_ReadChannel()` is slower due to channel-addressing overhead.
- In batch mode, multiple channels are read without channel-addressing ⇨ better performance
  - `adc_InitMCH()`: open batch mode
  - `adc_ReadMCH()`: read user-assigned channels
  - `adc_CloseMCH()`: close batch mode

Batch Mode

```
if adc_InitMCH(ADC_USECHANNEL0 + ADC_USECHANNEL1 + ……,
               ADCMODE_RANGE_2VREF,
               ADCMODE_UNSIGNEDCODING):
    ……
    adc_data = adc_ReadMCH()
    for i in range(8):
        print “A/D channel ”, i, “=”, adc_data[i]
    ……
    adc_CloseMCH()
```

- Parameters ADC_USECHANNEL0 ~ ADC_USECHANNEL7
  - Indicate which A/D channels to read in batch mode

I²C lib
(Simple API)

RoBoard H/W I²C Features

- Support both master & slave modes
- Support 10-bit address (master only)
  - but not implemented in RoBoIO
- Support all I²C speed modes
  - standard mode (~100 Kbps)
  - fast mode (~400 Kbps)
    - must pull-up I2C0_SCL, I2C0_SDA pins
  - high-speed mode (~3.3 Mbps)
    - To achieve 3.3 Mbps, pull-up resisters should ≤ 1K ohm

Usage Overview: Master Mode

```
if i2c_Init(speed_mode, bps):
    ……
    #use master API of I2C lib here
    ……
    i2c_Close() #close I2C lib
```

- `speed_mode` can be
  - `I2CMODE_STANDARD`: standard mode
  - `I2CMODE_FAST`: fast mode
  - `I2CMODE_HIGHSPEED`: high-speed mode
  - `I2CMODE_AUTO`: automatically set speed mode according to `bps`
- `bps` can be any integer ≤ 3300000 (3.3 Mbps)

Master API

- `i2c_Send(addr, buf, size)`: write a byte sequence to I²C device
  - `addr`: the I²C device address
  - `buf`: the byte array to write
  - `size`: the number of bytes to write

```
buf = [0x11, 0x22, 0x33]

i2c_Send(0x30, buf, 3)  #write 3 bytes to an I²C device
                        #with address 0x30
```

Master API

- `i2c_Receive(addr, buf, size)`: read a byte sequence from I²C device
  - `addr`: the I²C device address
  - `buf`: the buffer to put read bytes
  - `size`: the number of bytes to read

```
buf = []
i2c_Receive(0x30, buf, 3)  #read 3 bytes from an I²C
                           #device with address 0x30
for i in range(3):
    print “buf[”, i, “] = ”, buf[i]
```

Master API

- `i2c_SensorRead(addr, cmd, buf, size)`: a general function used to read I²C sensor data
  - Will first write `cmd` to I²C device, and then send I²C RESTART to read a byte sequence into `buf`
  - `addr`: the I²C device address
  - `cmd`: the byte to first write
    - Usually corresponds to a command of an I²C sensor
  - `buf`: the buffer to put read bytes
  - `size`: the number of bytes to read

Master API

- `i2c_SensorReadEX(addr, cmd, csize, buf, size)`: a general function used to read I²C sensor data
  - Same as `i2c_SensorRead()` except that `cmd` is a byte array here
    - Used for the case where I²C sensor command is > 1 byte
  - `addr`: the I²C device address
  - `cmd`: the byte array to first write
  - `csize`: the number of bytes in `cmd`
  - `buf`: the buffer to put read bytes
  - `size`: the number of bytes to read

Master API

```
buf = []

# first write 0x02 to an I²C device with address 0x70
# and then restart to read 2 bytes back
i2c_SensorRead(0x70, 0x02, buf, 2)
```

```
cmd = [0x32, 0x33]
buf = []

# first write 0x32 & 0x33 to an I²C device with address 0x53
# and then restart to read 6 bytes back
i2c_SensorReadEX(0x53, cmd, 2, buf, 6)
```

Remarks on I²C Device Address

- Some vendors describes their devices' address as the form:

  [7-bit slave address, R/W bit]

  - Ex.: the SRF08 ultrasonic sensor has address `0xE0` (for read) and `0xE1` (for write) by default.
  - The LSB of these addresses are actually the R/W bit.
- When accessing such devices, you should put the 7-bit slave address in RoBoIO I²C API calls, rather than their device address.

I²C ~Reset Pin of RB-110/RB-050

- Control of the ~Reset pin on I²C connector of RB-110/RB-050
  - `i2c_SetResetPin()`: set ~Reset pin to output HIGH
  - `i2c_ClearResetPin()`: set ~Reset pin to output LOW
- By default, the BIOS will set ~Reset pin to HIGH after booting.

Software-Simulated I²C

- From v1.8, RoBoIO includes S/W-simulated I²C functions to support non-standard I²C devices (e.g., LEGO® NXT ultrasonic sensor).
  - Support only I²C master mode
  - Consider no I²C arbitration
    - i.e., assume there is only one master on the I²C bus
  - Output 3.3V as logic HIGH
    - Should ensure your devices accept 3.3V as input

Software-Simulated I²C

- Usage overview

```
if i2c_InitSW(i2c_mode, clock_delay):
    ……
    #you can use any master API here; e.g.,
    buf = [0x11, 0x22, 0x33]
    i2c_Send(0x53, buf, 3)
    i2c_SensorRead(0x53, 0x02, buf, 3)
    ……
    i2c_CloseSW() #close S/W-simulated I²C
```

Software-Simulated I²C

- Usage overview (cont.)
  - `i2c_mode` can be
    - `I2CSW_NORMAL`: simulate standard I²C protocol
    - `I2CSW_LEGO`: simulate LEGO® NXT I²C protocol
  - `clock_delay` is any unsigned integer to control S/W-simulated I²C clock speed.
    - For LEGO® NXT sensors, the suggested `clock_delay` is 46 to achieve 9600bps.
    - If `clock_delay` = 0, the clock speed is about 75Kbps.

I²C lib
(Advanced API)

Advanced Master API

- The most simple ones of all advanced I²C Master API
  - `i2c0master_StartN()`: send **START** signal to slave devices
  - `i2c0master_WriteN()`: write a byte to slave devices
  - `i2c0master_ReadN()`: read a byte from slave devices
  - Automatically send STOP signal after reading/writing the last byte

```
i2c0master_StartN(0x30, #slave address = 0x30
            I2C_WRITE, #perform write action (use I2C_READ
                       #instead for read action)
            3)         #3 bytes to write

i2c0master_WriteN(0x11)
i2c0master_WriteN(0x22)
i2c0master_WriteN(0x33) #auto send STOP after this
```

Advanced Master API

- Send RESTART instead of STOP
  - Call `i2c0master_SetRestartN()` before the first reading/writing
  - Then RESTART signal, instead of STOP, will be sent after reading/writing the last byte

```
i2c0master_StartN(0x30, I2C_WRITE, 2)

#set to RESTART for reading 1 bytes (after I²C writes)
i2c0master_SetRestartN(I2C_READ, 1)

i2c0master_WriteN(0x44)
i2c0master_WriteN(0x55)   #auto send RESTART after this

data = i2c0master_ReadN() #auto send STOP after this
```

Usage Overview: Slave Mode

```
if i2c_Init(speed_mode, bps):
    #set slave address (7-bit) as, e.g., 0x30
    i2c0slave_SetAddr(0x30)
    ……
    #Slave Event Loop here
    ……
    i2c_Close() #close I2C lib
```

- This mode allows you to simulate RoBoard as an I²C slave device.
- In Slave Event Loop, you should use Slave API (rather than Master API) to listen and handle I²C bus events.

Slave Event Loop

```
while ……:
    state = i2c0slave_Listen()
    if state == I2CSLAVE_START:  #receive START signal
        #action for START signal
    elif state == I2CSLAVE_WRITEREQUEST: #request slave to write
        #handle write request
    elif state == I2CSLAVE_READREQUEST:  #request slave to read
        #handle read request
    elif state == I2CSLAVE_END: #receive STOP signal
        #action for STOP signal

    …… //can do stuff here when listening
```

Slave Read/Write API

- Call `i2c0slave_Write()` for sending a byte to master

```
……
elif state == I2CSLAVE_WRITEREQUEST:
    i2c0slave_Write(byte_value)
……
```

- Call `i2c0slave_Read()` for reading a byte from master

```
……
elif state == I2CSLAVE_READREQUEST:
    data = i2c0slave_Read()
……
```

RC Servo lib
(with GPIO functions)

Features

- Dedicated to PWM-based RC servos
  - Employ RoBoard's PWM generator
  - So don't use RC Servo lib & PWM lib at the same time
- Can read the width of feedback pulses
  - Very accurate in DOS (±1us)
  - Occasionally miss accuracy in XP, CE, and Linux, when the OS is being overloaded
- Support GPIO (digital I/O) functions

Usage Overview

```
……
#Configure servo setting (using Servo Configuration API) here
……
if rcservo_Init(RCSERVO_USEPINS1 + RCSERVO_USEPINS2 + ……):
    ……
    #use Servo Manipulation API here
    ……
    rcservo_Close()
```

- Parameters RCSERVO_USEPINS1 ~ RCSERVO_USEPINS24
  - Indicate which PWM pins are used as Servo Mode (for RB-110/ RB-050, RCSERVO_USEPINS17 ~ RCSERVO_USEPINS24 are invalid)
  - Other unused PWM pins will be set as GPIO Mode

Usage Overview

- Servo Configuration API allows to configure various servo parameters.
  - PWM period, max/min PWM duty
  - Feedback timings for position capture
  - ……
- Servo-mode pins allow three servo manipulation modes.
  - Capture mode (for reading RC servo's position feedback)
  - Action playing mode (for playing user-defined motions)
  - PWM mode (send PWM pulses for individual channels)

Configure Servo Setting

- **Method 1:** Use built-in parameters by calling

```
rcservo_SetServo(pin, servo_model)
```

  - `pin` indicates which PWM pin to set, and can be `RCSERVO_PINS1` ~ `RCSERVO_PINS24`
    - For RB-110/RB-050, `RCSERVO_PINS17` ~ `RCSERVO_PINS24` are invalid.

Configure Servo Setting

- **Method 1:** (cont.)
  - `servo_model` indicates what servo is connected on the PWM pin, and can be
    - `RCSERVO_KONDO_KRS78X`: for KONDO KRS-786/788 servos
    - `RCSERVO_KONDO_KRS4024`: for KONDO KRS-4024 servos
    - `RCSERVO_KONDO_KRS4014`: for KONDO KRS-4014 servos
      - KRS4014 doesn't directly work on RB-100/RB-110; see later slides for remarks.
    - `RCSERVO_HITEC_HSR8498`: for HiTEC HSR-8498 servos

Configure Servo Setting

- **Method 1:** (cont.)
  - `servo_model` can be (cont.)
    - `RCSERVO_FUTABA_S3003`: for Futaba S3003 servos
    - `RCSERVO_SHAYYE_SYS214050`: for Shayang Ye SYS-214050 servos
    - `RCSERVO_TOWERPRO_MG995`, `RCSERVO_TOWERPRO_MG996`: for TowerPro MG995 & MG996 servos

Configure Servo Setting

- **Method 1:** (cont.)
  - `servo_model` can be (cont.)
    - `RCSERVO_GWS_S03T`, `RCSERVO_GWS_S777`: for GWS S03T & S777 series servos
    - `RCSERVO_GWS_MICRO`: for GWS MICRO series servos
    - `RCSERVO_DMP_RS0263`, `RCSERVO_DMP_RS1270`: for DMP RS-0263 & RS-1270 servos

Configure Servo Setting

- **Method 1:** (cont.)
  - `servo_model` can be (cont.)
    - `RCSERVO_SERVO_DEFAULT`: attempt to adapt to various servos of supporting position feedback
    - `RCSERVO_SERVO_DEFAULT_NOFB`: similar to the above option, but dedicated to servos with no feedback
      - Default option if you don't set the servo model before calling `rcservo_Init()`
  - If you don't know which model your servos match, use `RCSERVO_SERVO_DEFAULT_NOFB`

Configure Servo Setting

```
#PWM pin S1 connects KONDO servo KRS-786/788
rcservo_SetServo(RCSERVO_PINS1, RCSERVO_KONDO_KRS78X)

#PWM pin S3 connects DMP servo RS-0263
rcservo_SetServo(RCSERVO_PINS3, RCSERVO_DMP_RS0263)

#open RC Servo lib to control servos on pins S1 & S3
if rcservo_Init(RCSERVO_USEPINS1 + RCSERVO_USEPINS3):
    ……
    #use Servo Manipulation API here
    ……
    rcservo_Close()
```

Configure Servo Setting

- **Method 2:** **Call parameter-setting functions to set customized parameters**
  - **In theory, using this method, we can adapt RC Servo lib to any PWM-based RC servos.**
  - **It requires detailed servo knowledge, and we will provide a document for this in the future.**

Manipulate Servo: Capture Mode

- Call `rcservo_EnterCaptureMode()` to enter this mode
  - Capture mode is the initial mode of servo-mode pins after calling `rcservo_Init()`
  - Note: Servos with no feedback are not supported in this mode.
- Available API in Capture mode
  - `rcservo_CapOne(pin)`: read position feedback from a specified servo-mode pin
    - return `0xffffffff` if fails to read feedback, or if the pin is connected to a servo with no feedback

Manipulate Servo: Capture Mode

- Available API in Capture mode (cont.)
  - `rcservo_CapAll(frame)`: read position feedback from all servo-mode pins
    - `frame` is an array of 32 unsigned long integers, where `frame[0]` will give position feedback on pin S1; `frame[1]` on pin S2; and …
    - `frame[i]` will give `0xffffffff` if fails to read feedback on the corresponding pin, or if the servo is with no feedback
    - for RB-100/100RD, `frame[24~31]` are reserved; for RB-110/050, `frame[16~31]` are reserved.

Manipulate Servo: Capture Mode

```
rcservo_EnterCaptureMode()
……
#read position feedback from PWM pin S3
pos = rcservo_CapOne(RCSERVO_PINS3)
……
#read position feedback from all servo-mode pins
motion_frame = []
rcservo_CapAll(motion_frame)

print “position feedback on PWM pin S3 is equal to”,
print motion_frame[2], “microsecond”
```

Manipulate Servo: Capture Mode

- Available API in Capture mode (cont.)
  - `rcservo_ReadPositions()`: read position feedback from multiple specified servo-mode pins

```
#read position feedback from PWM pins S1 and S3
motion_frame = []
rcservo_ReadPositions(RCSERVO_USEPINS1 + RCSERVO_USEPINS3,
                      0, #normally = 0
                      motion_frame)
print “position feedback on PWM pins S1 and S3 are equal to”,
print motion_frame[0], “and”, motion_frame[2], “microseconds”
```

Manipulate Servo: Action Playing Mode

- Can replay the motion frames that are captured by `rcservo_CapAll()`
- Methods to enter this mode
  - `rcservo_EnterPlayMode()`: for servos with feedback
    - Will automatically capture the current pose as the initial motion frame (home position)
    - Will reject moving servos that have no feedback
  - `rcservo_EnterPlayMode_HOME(home)`: for servos with no feedback
    - `home` is an array of 32 unsigned long integers which indicates the initial motion frame.

Manipulate Servo: Action Playing Mode

- Entering Playing Mode, all servo-mode pins will send PWM pulses continuously.
  - In general, this will make all connected servos powered always.
- To stop the pulses, just leave Playing Mode by, e.g., calling `rcservo_EnterCaptureMode()`

Manipulate Servo: Action Playing Mode

- Blocking API in Action playing mode
  - `rcservo_MoveOne(pin, pos, time)`: move a servo until it reach the target position
  - `rcservo_MoveTo(frame, time)`: move all servos until they reach to the next motion frame
    - `frame[0]` indicates target position for servo on pin S1; `frame[1]` for pin S2; and …
    - `frame[i] = 0` indicates the corresponding servo to remain at its last position.

Manipulate Servo: Action Playing Mode

```
rcservo_EnterPlayMode()
……
#move servo on PWM pin S2 to position 1500us in 500ms
rcservo_MoveOne(RCSERVO_PINS2, 1500, 500)
```

```
rcservo_EnterPlayMode()
……
#move simultaneously both servos on PWM pins S1 and S3 to
#position 1500us in 500ms

motion_frame = [1500, 0, 1500, 0, ……, 0]

rcservo_MoveTo(motion_frame, 500)
```

Manipulate Servo: Action Playing Mode

- Non-blocking API in Action playing mode
  - `rcservo_SetAction(frame, time)`: set the next motion frame
    - Can be called, before the following function returns `RCSERVO_PLAYEND`, to change the target positions
  - `rcservo_PlayAction()`: push all servos to reach the frame that was set by `rcservo_SetAction()`
    - Must call `rcservo_PlayAction()` repeatedly until it returns `RCSERVO_PLAYEND` (which indicates that all servos have reached the target)

Manipulate Servo: Action Playing Mode

```
rcservo_EnterPlayMode()
……
motion_frame = [……]
#here set up the content of motion_frame[] for playing
……
rcservo_SetAction(motion_frame, 500) #play motion in 500ms
while rcservo_PlayAction() != RCSERVO_PLAYEND:
    #
    #can do stuff here when playing motion
    #
```

Manipulate Servo: Action Playing Mode

- Non-blocking API (cont.)
  - `rcservo_StopAction()`: stop playing the motion frame immediately
    - `rcservo_PlayAction()` will return `RCSERVO_PLAYEND` after calling this
  - `rcservo_GetAction(buf)`: get the current positions of all servos
    - `buf[0]` will give the position of servo on pin S1; `buf[1]` on pin S2; and ...

Manipulate Servo: Action Playing Mode

```
rcservo_EnterPlayMode()
……
buf = []
motion_frame = [……] #set up motion_frame[] for playing
……
rcservo_SetAction(motion_frame, 500) #play motion in 500ms
while rcservo_PlayAction() != RCSERVO_PLAYEND:
    rcservo_GetAction(buf)
    print “Servo on pin S1 is moving to ”, buf[0]
```

Manipulate Servo: PWM Mode

- Call `rcservo_EnterPWMMode()` to enter this mode
  - In this mode, all servo-mode pins output 0V if no pulse is sent.
- Available API in PWM mode
  - `rcservo_SendPWM()`: send a given number of pulses with specific duty and period
  - `rcservo_IsPWMCompleted()`: return true when all pulses have been sent out

Manipulate Servo: PWM Mode

```
rcservo_EnterPWMMode()
……
PWM_period = 10000 #10000us
PWM_duty   = 1500  #1500us
count      = 100
rcservo_SendPWM(pin, #RCSERVO_PINS1 or RCSERVO_PINS2 or ……
                PWM_period, PWM_duty, count)
while (not rcservo_IsPWMCompleted(pin)):
    #
    #can do stuff here when waiting for PWM completed
    #
```

Manipulate Servo: PWM Mode

- Available API in PWM mode (cont.)
  - `rcservo_SendCPWM()`: send continuous pulses with specific duty and period
  - `rcservo_StopPWM()`: stop the pulses caused by `rcservo_SendPWM()`/`rcservo_SendCPWM()`
  - `rcservo_CheckPWM()`: return the remaining number of pulses to send
    - return `0` if pulses have stopped
    - return `0xffffffff` for continuous pulses

Manipulate Servo: PWM Mode

```
rcservo_EnterPWMMode()
……
PWM_period = 10000 #10000us
PWM_duty   = 1500  #1500us

rcservo_SendCPWM(pin, #RCSERVO_PINS1 or RCSERVO_PINS2 or ……
                 PWM_period, PWM_duty)
……
#do something when sending PWM
……
rcservo_StopPWM(pin)
```

GPIO Functions

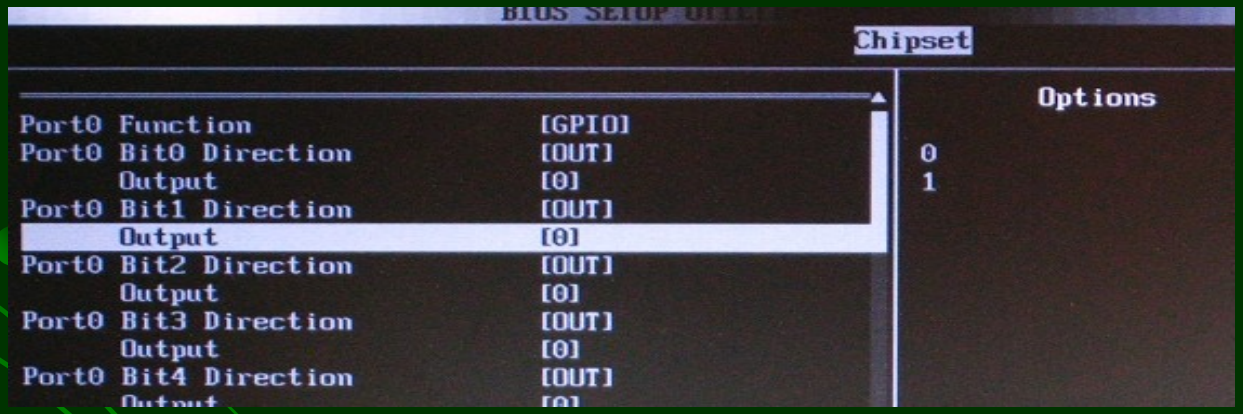
- API to control GPIO-mode pins
  - `rcservo_OutPin(pin, value)`: set GPIO pin to output HIGH or LOW
    - `pin` = `RCSERVO_PINS1` or `RCSERVO_PINS2` or ……
    - `value` = `0` (output LOW) or `1` (output HIGH)
  - `rcservo_InPin(pin)`: read input from GPIO pin
    - Return `0` if it read LOW, and `1` if it read HIGH
- The API will do nothing if `pin` is a servo-mode pin.

BIOS Setting for RC Servos

- Some RC servos (e.g., KONDO KRS-788) require the PWM input signal = LOW at power on.
- Configure RoBoard's PWM pins to achieve this
  - STEP 1: Switch the pull-up/pull-down switch to "pull-down"
  - STEP 2: Go to BIOS Chipset menu
  - STEP 3: Select `SouthBridge Configuration` → `Multi-Function Port Configuration`

BIOS Setting for RC Servos

- Configure RoBoard's PWM pins (cont.)
  - STEP 4: Set `Port0 Bit0~7`, `Port1 Bit0~7`, `Port2 Bit0~7(only for RB-100/100RD)`, `Port3 Bit6` as `Output [0]`



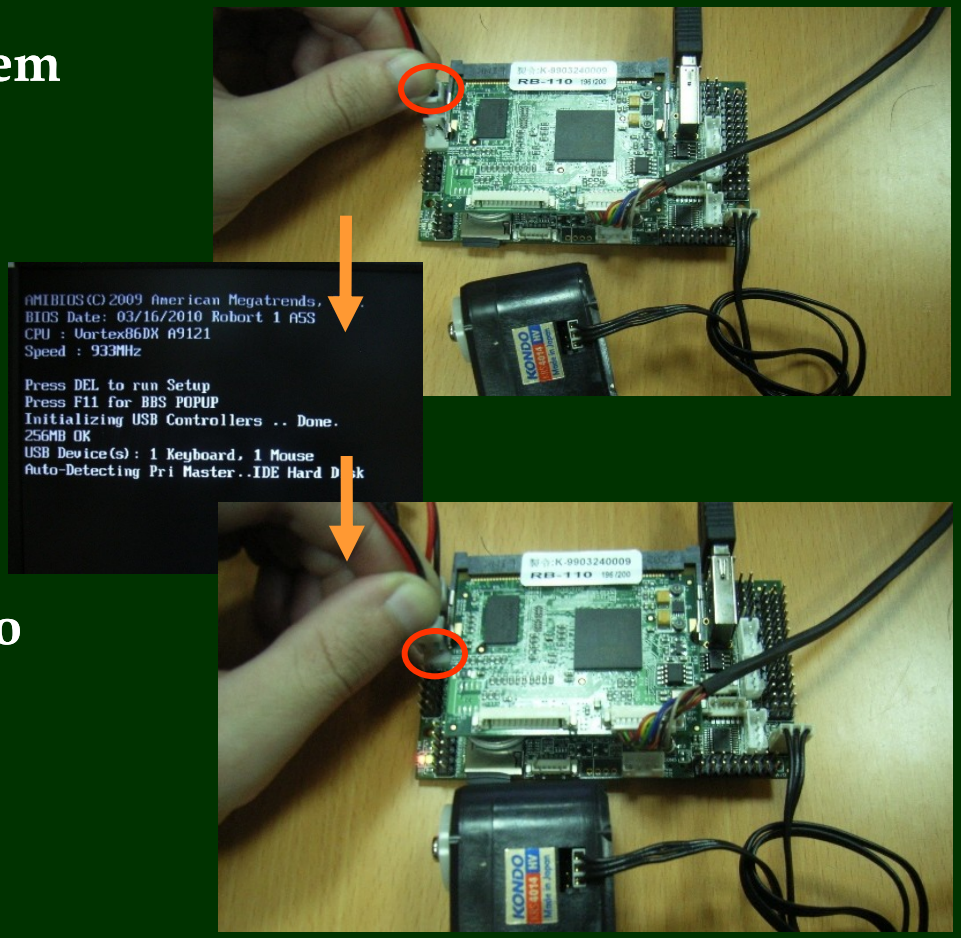
- Can also set RoBoard's PWM pins = HIGH at power on
  - Just switch the pull-up/pull-down switch to "pull-up"

Remarks for KONDO KRS-4014

- KRS-4014 servos also require PWM = LOW at power on.
  - But the former pull-up/-down setting is not enough to make KRS-4014 work on RB-100/RB-110.
  - You also need to power on KRS-4014 and RB-100/RB-110 at different time.
    - This implies that you need to power-supply the both separately.

Remarks for KONDO KRS-4014

- Example: Make KRS-4014 work on RB-110.
  - STEP 1: turn on the system power of RoBoard first
  - STEP 2: wait the BIOS screen appeared
  - STEP 3: turn on the servo power for KRS-4014



COM Ports

RoBoard Native COM Ports

- COM1~COM4 can be used as standard COM ports in WinXP, Linux, and DOS
- Max speed
  - RB-100: 115200 bps
  - RB-100RD/RB-110/RB-050: 748800 bps
- Can customize each native COM port in BIOS
  - IRQ
  - I/O base address
  - Default speed

Boosting Mode of RB-100RD/110/050 Native COM Ports

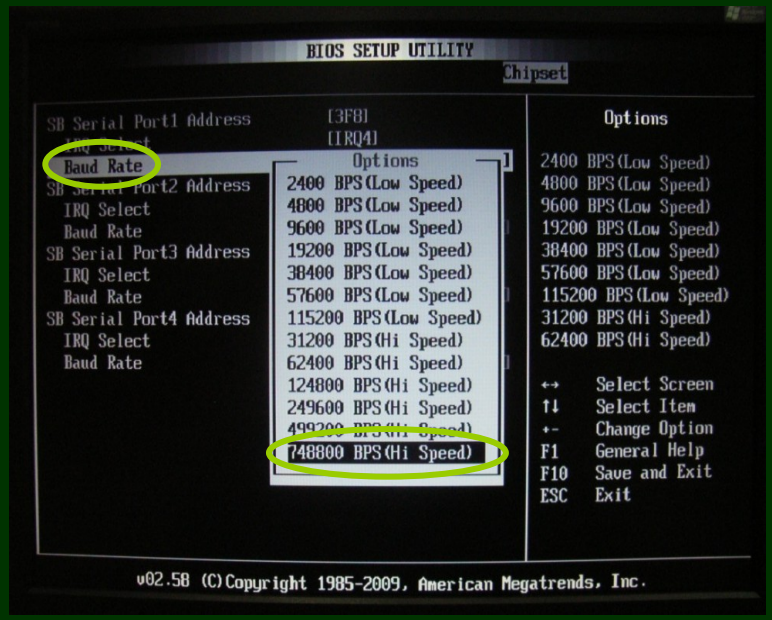
- RB-100RD/110/050's native COM ports support baudrates up to 750K bps, provided that COM boosting mode is enabled.
- When boosting mode enabled,

  the real baudrate = 13 × the original baudrate

  - For example, if boosting mode of COM3 is enabled and its baudrate is set to 38400 bps, the real baudrate is 38400 × 13 = 500K bps.
- In boosting mode, the maximum baudrate is 57600 × 13 = 750Kbps (115200 × 13 is not allowed)

How to Enable Boosting Mode of RB-100RD/110/050 Native COM Ports

- **Method 1:** Using BIOS
  - STEP 1: Go to RB-100RD/110/050 BIOS `Chipset` menu
  - STEP 2: Select `SouthBridge Configuration` → `Serial/Parallel Port Configuration`
  - STEP 3: Select the COM port that you want to boost
  - STEP 4: Set its baudrate to any speed > 115200 bps



How to Enable Boosting Mode of RB-100RD/110/050 Native COM Ports

- **Method 2:** Using `rbcom.exe` in RoBoKit.
  - Run `rbcom.exe` directly to see its usage
- **Method 3:** Using the isolated API of COM lib (refer to the later COM lib slides)

```
io_init()
……
com2_EnableTurboMode()  #enable COM2 boosting mode
……
com4_DisableTurboMode() #disable COM4 boosting mode
……
io_close()
```

RB-110 FTDI COM Ports

- COM5 & COM6 of RB-110 are realized by its on-board FTDI FT2232H chip.
  - So require to install dedicated drivers for their usage
    - See also RB-110 WinXP/Linux installation guide for more information.
  - Detailed application notes for FTDI FT2232H can be found on FTDI's web site:

    http://www.ftdichip.com/Support/FTDocuments.htm

FTDI COM vs. Native COM

- FTDI COM allows faster baudrates than RoBoard's native COM.
- But FTDI COM has also much longer latency between two packet transmission.
  - In transmitting multiple packets, FTDI COM may be slower than native COM due to its latency.
- You should experiment to see which COM meets your application.

COM lib

Usage Overview

- From RoBoIO 1.8, we add COM lib to
  - make users easier to handle H/W features (e.g., boosting mode) of RoBoard's native COM ports
  - provide a simple and unified serial API for various OS
    - Currently only support WinXP, WinCE, Linux
- Note that COM lib only tackles RoBoard's native COM, i.e., COM1~COM4.
  - So RB-110's COM5 & COM6 aren't considered.

Usage Overview

- The API has different prefixes for different COM ports.
  - `com1_...` for COM1
  - `com2_...` for COM2
  - `com3_...` for COM3
  - `com4_...` for COM4
- Following slides shall only mention COM3 API for illustration.

Usage Overview

```
if com3_Init(mode):
    com3_SetBaud(……)   #optional
    com3_SetFormat(……) #optional
    ……
    #use COM lib API here
    ……
    com3_Close()
```

- mode can be
  - COM_FDUPLEX: this port is used as a full-duplex COM (invalid for COM2 and RB-100/100RD's COM4)
  - COM_HDUPLEX: this port is used as a half-duplex COM (invalid for COM1)
    - Select this if you short the TX/RX lines of COM3

Baudrate

- `com3_SetBaud(baudrate)`: set the baudrate; `baudrate` can be
  - `COMBAUD_748800BPS`: 750Kbps (invalid for RB-100)
  - `COMBAUD_499200BPS`: 500Kbps (invalid for RB-100)
  - `COMBAUD_115200BPS`: 115200bps
  - `COMBAUD_9600BPS`: 9600bps
  - …… (See the wrapper source for all available baudrates)
- The default baudrate is 115200bps when calling `com3_Init()`.

Data Format

- `com3_SetFormat(bytesize, stopbit, parity)`: set the data format
  - `bytesize` can be
    - `COM_BYTESIZE5`: byte size = 5 bits
    - `COM_BYTESIZE6`: byte size = 6 bits
    - `COM_BYTESIZE7`: byte size = 7 bits
    - `COM_BYTESIZE8`: byte size = 8 bits
  - `stopbit` can be
    - `COM_STOPBIT1`: 1 stop bit
    - `COM_STOPBIT2`: 2 stop bit

Data Format

- `com3_SetFormat(…)`: (cont.)
  - `parity` can be
    - `COM_NOPARITY`: no parity bit
    - `COM_ODDPARITY`: odd parity
    - `COM_EVENPARITY`: even parity
- The default data format is 8-bit data, 1 stop bit, no parity when calling `com3_Init()`.

Write API

- `com3_Write(byte)`: write a byte to COM3

```
com3_Write(0x55) #write 0x55 to COM3
```

- `com3_Send(buf, size)`: write a byte sequence to COM3
  - `buf`: the byte array to write
  - `size`: the number of bytes to write

```
buf = [0x11, 0x22, 0x33]

com3_Send(buf, 3) #write 3 bytes to COM3
```

Write API

- `com3_ClearWFIFO()`: cancel all bytes in write-FIFO
- `com3_FlushWFIFO()`: wait until all bytes in write-FIFO are sent out

```
buf = [0xff, 0x01, 0x02, 0x01]

com3_Send(buf, 4) #write 4 bytes to COM3
com3_FlushWFIFO() #wait until these bytes are sent out
```

Read API

- `com3_Read()`: read a byte from COM3
  - return `0xffff` if timeout

```
data = com3_Read()
```

- `com3_Receive(buf, size)`: read a byte sequence from COM3
  - `buf`: the byte buffer to put read bytes
  - `size`: the number of bytes to read

```
buf = []
com3_Receive(buf, 3) #read 3 bytes from COM3
```

Read API

- `com3_ClearRFIFO()`: discard all bytes in read-FIFO
- `com3_QueryRFIFO()`: query the number of bytes in read-FIFO

```
buf = []
while (com3_QueryRFIFO() < 4):  #wait until there are
                                #4 bytes in read-FIFO
com3_Receive(buf, 4)   #read the 4 bytes from read-FIFO
```

Special API for AI Servos

- `com3_ServoTRX(cmd, csize, buf, size)`: send servo command to and then read feedback data from COM3
  - `cmd`: the byte array to send first
  - `csize`: the number of bytes in `cmd`
  - `buf`: the byte buffer to put read bytes
  - `size`: the number of bytes to read

Special API for AI Servos

```
cmd = [0xff, 0xff, 0x01, 0x02, 0x01, 0xfb]
buf = []

# ping Dynamixel AX-12 servo of ID 0x01
com3_ServoTRX(cmd, 6, buf, 6)

Print “The feedback of AX-12 is ”,
for i in range(6):
    print buf[i],
print
```

Isolated API

- There are isolated API that can work without `com3_Init()` & `com3_Close()`
  - `com3_EnableTurboMode()`: enable COM3's boosting mode (invalid for RB-100)
  - `com3_DisableTurboMode()`: disable COM3's boosting mode (invalid for RB-100)
- Isolated API are usually used with external serial-port libraries.

Isolated API

- Usage 1: (without `com3_Init()` & `com3_Close()`)
  - will reserve the change made by isolated API even when the program exit

```
io_init(…)
……
com3_EnableTurboMode() #set COM3 into boosting mode
……
io_close() #the boosting-mode setting would be reserved
```

  - Note that except isolated API, you shouldn't mix COM lib with other serial lib (i.e., after you call `com3_Init()`, don't use other serial lib to access COM3).

Isolated API

- Usage 2: (with `com3_Init()` & `com3_Close()`)
  - will restore the change made by the isolated API

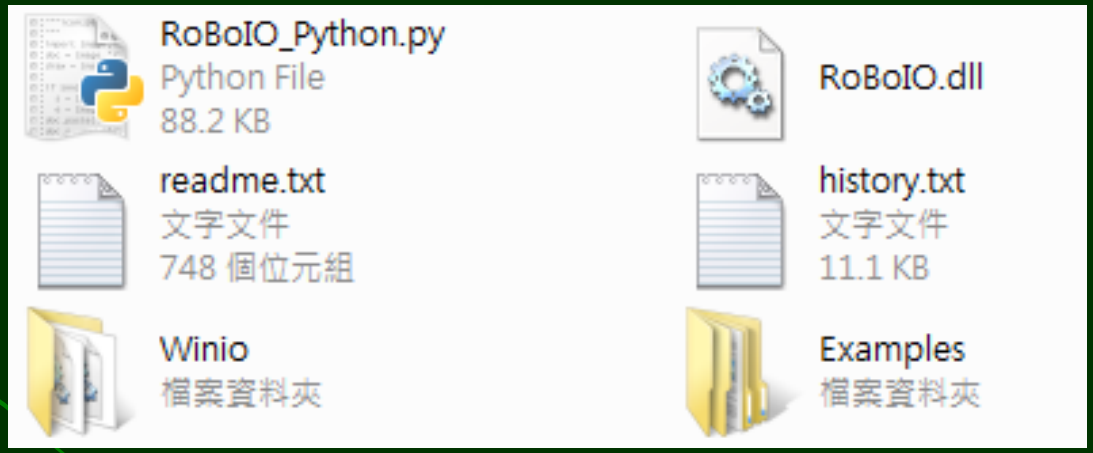
```
com3_Init(…)
……
com3_EnableTurboMode() #set COM3 into boosting mode
……
com3_Close() #will restore COM3’s original
             #boosting-mode setting after this
```

  - This is not the recommended usage of isolated API.

Installation
(for Python)

Setup RoBoIO Python Wrapper

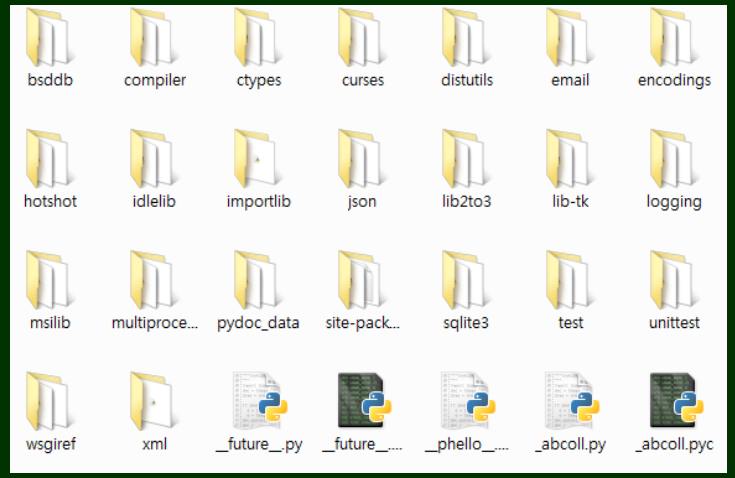
- Decompose RoBoIO Python Wrapper zip-file to, e.g., `C:\RoBoard_Python`



  - `Examples:` sample codes for RoBoIO Python Wrapper
  - `Winio:` needed when using RoBoIO under WinXP

Setup RoBoIO Python Wrapper

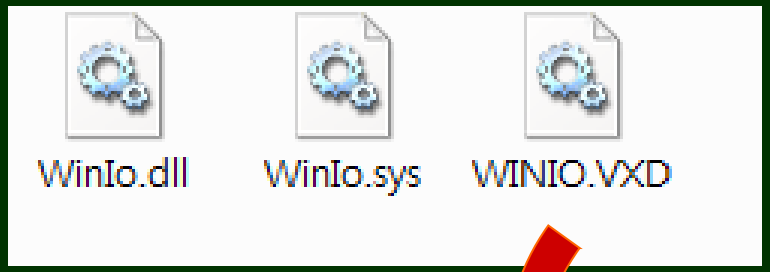
- Copy `RoBoIO_Python.py` to your Python's Lib directory.

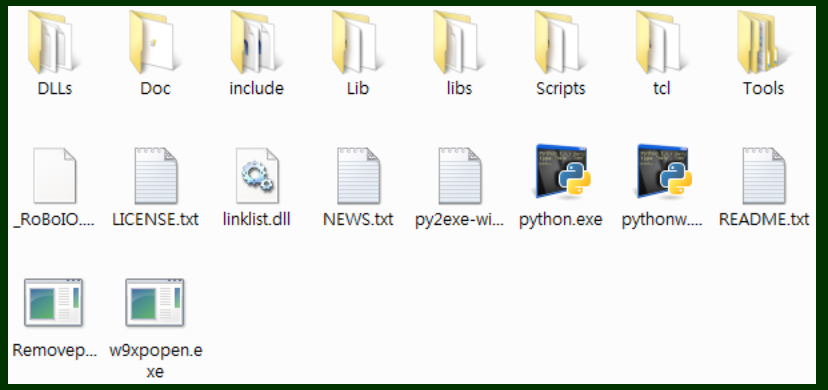


Your Python's Lib directory
(e.g., `C:\Python27\Lib`)

Setup RoBoIO Python Wrapper

- Copy all files in `\Winio` to your Python directory.

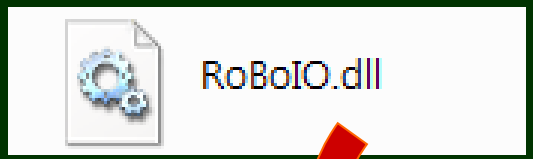


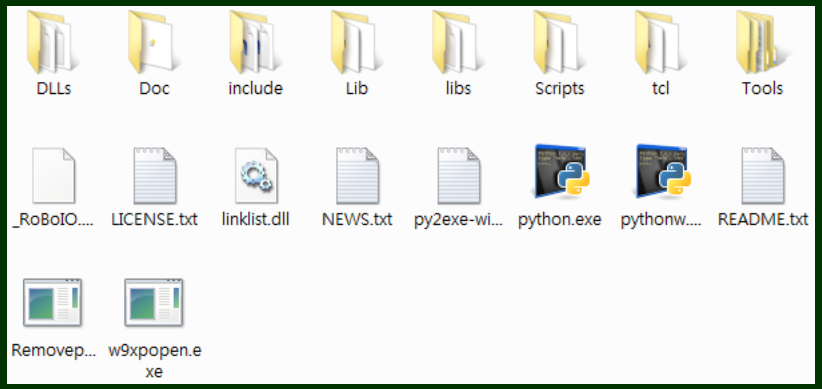


Your Python directory
(e.g., `C:\Python27`)

Setup RoBoIO Python Wrapper

- Copy `RoBoIO.dll` to your Python directory.





Your Python directory
(e.g., `C:\Python27`)

Some Remarks

- RoBoIO recognizes RoBoard's CPU, and doesn't run on other PC.
- It is suggested to login WinXP with administrator account for running RoBoIO applications.
- Don't run RoBoIO applications on Network Disk, which may fail RoBoIO.

Some Remarks

- You should install Python in RoBoard in order to run your RoBoIO Python applications.
  - Since the wrapper uses `ctypes` package, you may need Python 2.5 or higher to run it.
- The wrapper is designed for WinXP; for use in Linux, you need to modify the `RoBoIO_Python.py` file.

Applications

Introduction

- **x86-based** ⇨ Almost all resources on PC can be employed as development tools of RoBoard.
  - **Languages:** C/C++/C#, Visual Basic, Java, Python, LabVIEW, …
  - **Libraries:** OpenCV, SDL, LAPACK, …
  - **IDE:** Visual Studio, Dev-C++, Eclipse, …
  - **GUI (if needed):** Windows Forms, GTK, …

Introduction

- **Rich I/O interfaces** ⇨ **Various sensors & devices can be employed as RoBoard's senses.**
  - **A/D, SPI, I^2C:** accelerometer, gyroscope, …
  - **COM:** GPS, AI servos, …
  - **PWM:** RC servos, DC motors, …
  - **GPIO:** bumper, infrared sensors, on/off switches, …
  - **USB:** webcam, …
  - **Audio in/out:** speech interface

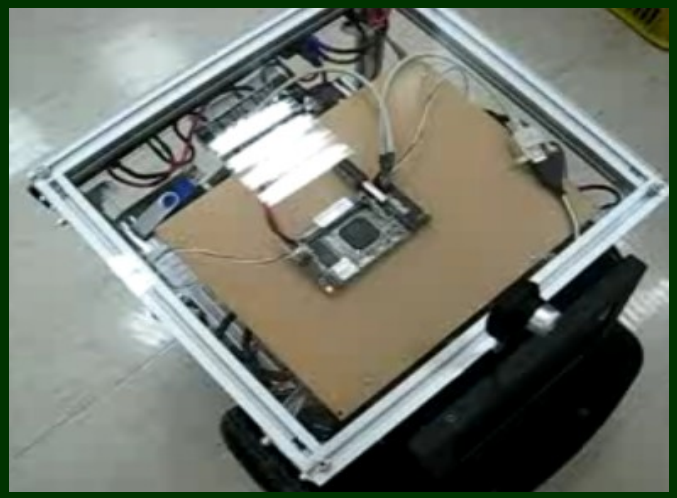
Introduction

Rich I/O (using RoBoIO) + Rich resources on PC

Can develop robots more easily and rapidly

Experiences

- Mobile robot controlled by wireless joystick



  - RoBoIO library + Allegro game library
  - Take < 20 minutes to complete the control program

Experiences

- KONDO manipulator with object tracking & face recognition

  - RoBoIO library + OpenCV library
  - Take < 3 hours to complete the program

Experiences

- KONDO humanoid (motion capture + replay, script control, MP3 sound, compressed data files)

  - RoBoIO library + irrKlang library + zziplib library
  - Take < 5 days to complete the program

Thank You

tech@roboard.com