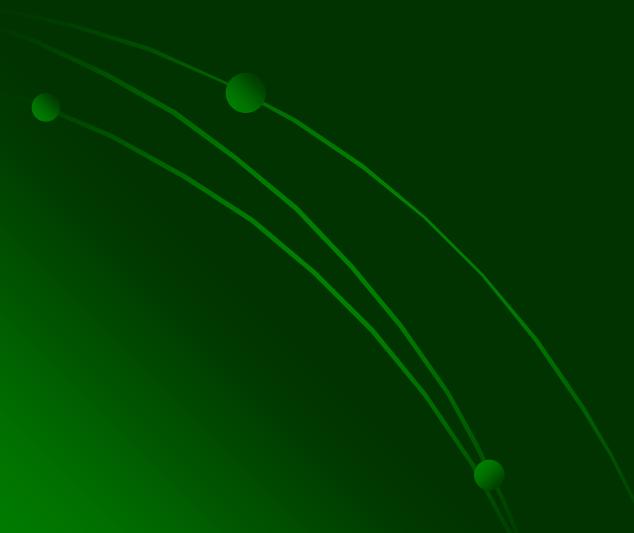


RoBoI0 1.7

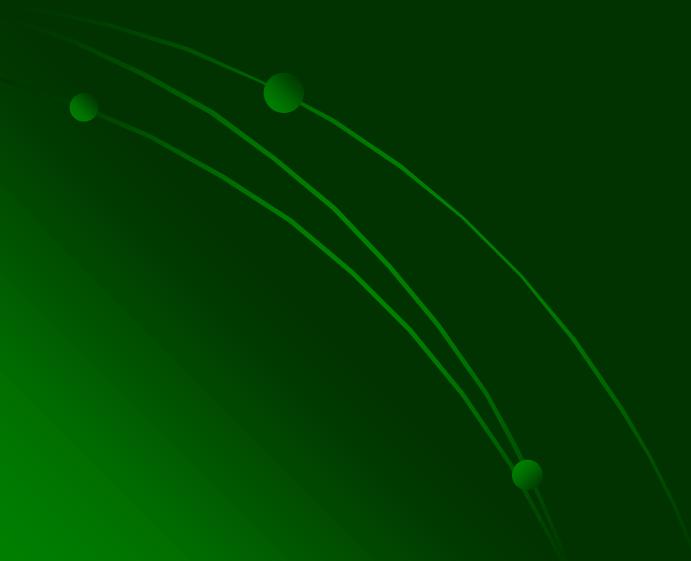
Software Development

Introduction



DMP Electronics Inc.
Robotics Division
March 2011

Overview



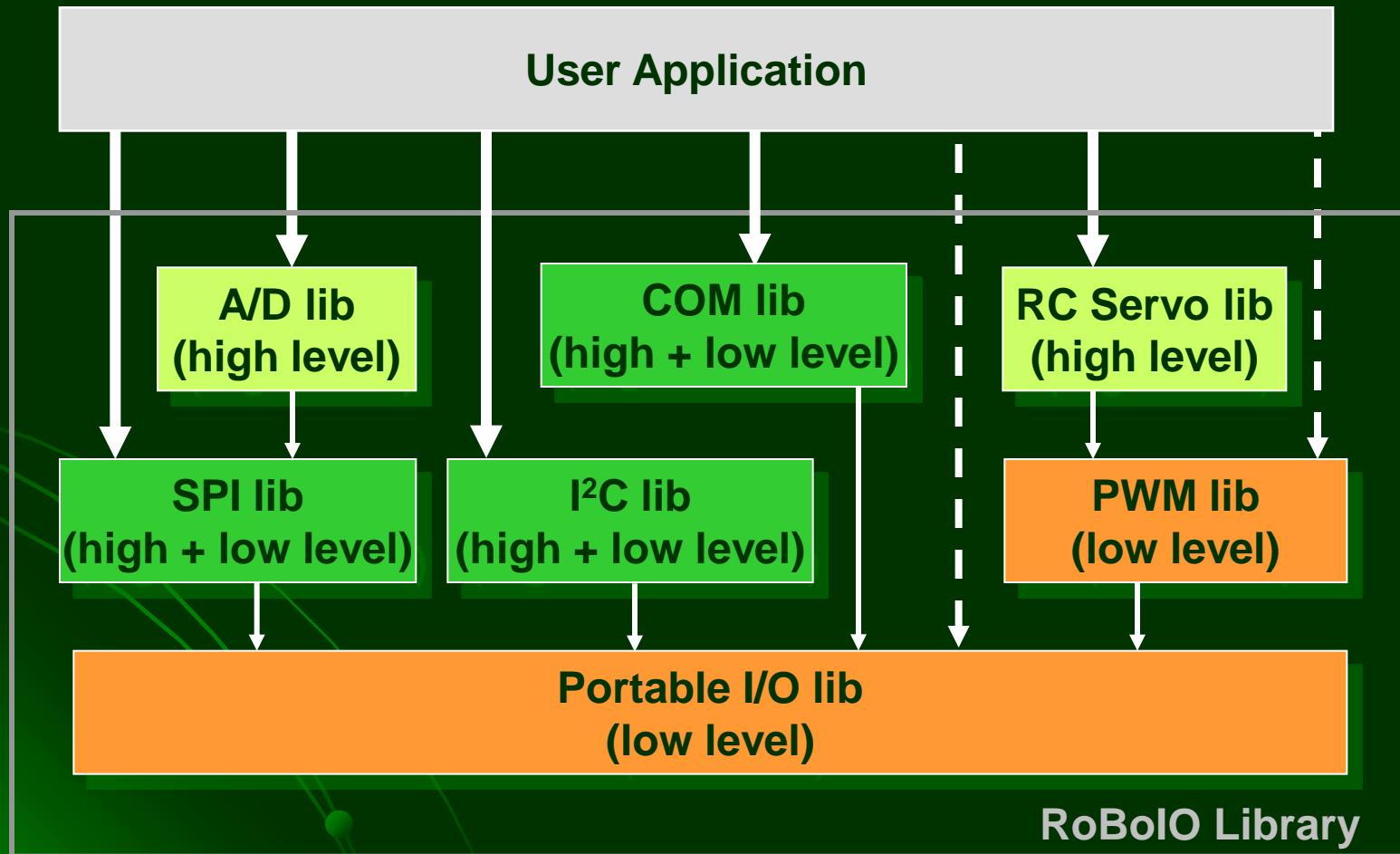
RoBoIO Library

- 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)
 - I²C (Inter-Integrated Circuit Interface)

RoBoLO Library

- Supported functions (cont.)
 - GPIO (General-Purpose Digital I/O)
 - RC servo control (KONDO, HiTEC, ...)
- Supported platforms
 - Windows XP: Visual Studio 2005/2008
 - require `WinIo` or `PciDebug` runtimes
 - Windows CE: Visual Studio 2005/2008
 - Linux: gcc
 - DOS: DJGPP, Watcom C++, Borland C++ 3.0

Architecture



Usage Overview

- Include **roboard.h** to use
 - SPI lib
 - A/D lib
 - I²C lib
 - RC Servo lib
- Call **roboio_SetRBVer(rb_ver)** to set your RoBoard correctly
 - select **rb_ver = RB_100** or **RB_110** or **RB_050** according to your RoBoard version

```
#include <roboard.h>
int main() {
    roboio_SetRBVer(...);

.....
// use API of RoBoIO
// library here
.....
return 0;
}
```

Usage Overview

- Include **roboard_dll.h** instead if you use RoBoIO DLL version
- The DLL version is only available on
 - Windows XP
 - Windows CE

```
#include <roboard_dll.h>
int main() {
    roboio_SetRBVer(...);

.....
// use API of RoBoIO
// library here

.....
return 0;
}
```

Usage Overview

- Error reporting of RoBoIO API
 - When any API function fails, you can always call **roboio_GetErrMsg()** to get the error message.
 - Example

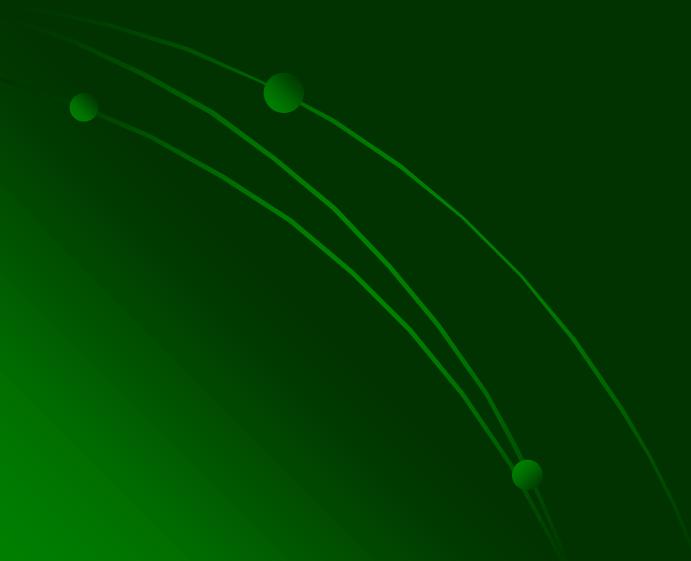
```
.....  
if (rcservo_Init(...) == false) {  
    printf("Fail to initialize RC Servo lib!!!\n");  
    printf("Error message: %s\n", roboio_GetErrMsg());  
    exit(0);  
}  
.....
```

Usage Overview

- Remarks

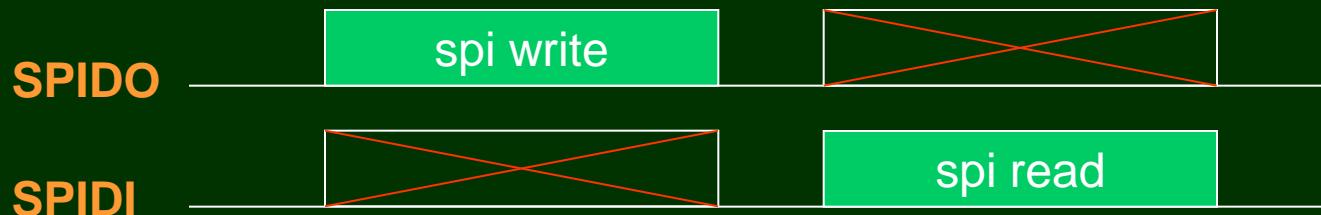
- For using PWM lib, you need to include **pwm.h** additionally
- Don't use both PWM lib and RC Servo lib at the same time
 - because PWM lib is managed within RC Servo lib

SPI lib



RoBoard H/W SPI Features & Limits

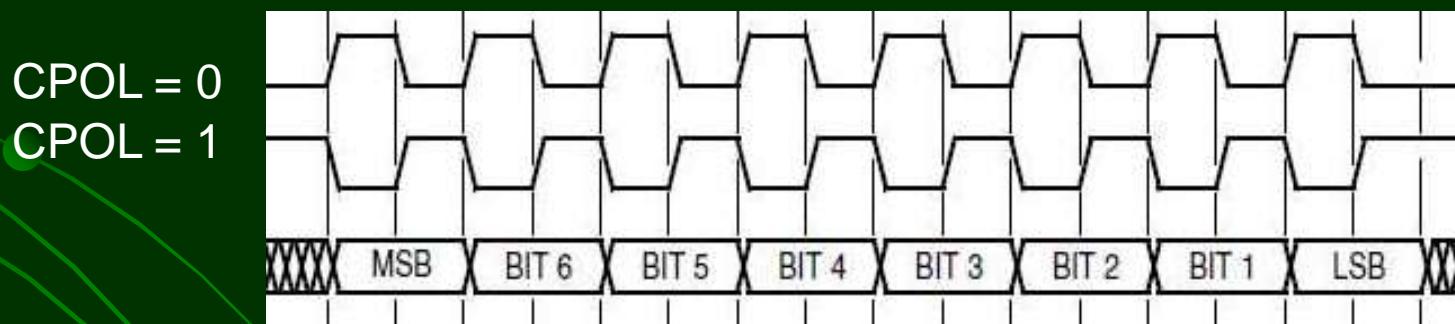
- Dedicated to SPI flash
- Half-Duplex



- 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



- 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 RB-110 H/W Introduction slide for more information.

Usage Overview

```
if (spi_Init(clock_mode)) {  
    ....  
    unsigned 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 **spi.h** 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()` or
`spi_WriteFlush()`



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_FIFOFflush()** to flush SPI FIFO
 - **spi_WriteFlush()** waits that SPI FIFO becomes empty.

SPISS Pin

- Control of the **SPISS** pin of RB-100
 - `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

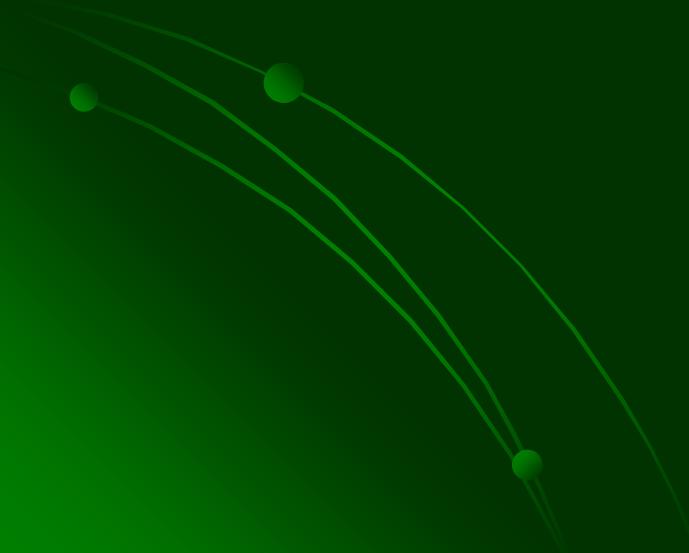
- Usage overview

```
if (spi_InitSW(clock_mode, clock_delay)) {  
    .....  
    // Half-Duplex read/write  
    unsigned val1 = spi_Read(); //read a byte from SPI bus  
    spi_Write(0x55); //write a byte (0x55) to SPI bus  
    // Full-Duplex read/write  
    unsigned 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.
 - So be careful about **bus conflict** if you have devices attached to SPI bus of RB-100.
 - Disable your SPI devices (using, e.g., **SPISS** pin of RB-100) when accessing A/D.

Usage Overview

```
if (spi_Init(SPICLK_21400KHZ)) {  
    ....  
    int 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)) {  
    ....  
    int 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 \Rightarrow better performance
 - `adc_InitMCH()`: open batch mode
 - `adc_ReadMCH()`: read user-assigned channels
 - `adc_CloseMCH()`: close batch mode

Batch Mode

```
int* ad_data;  
if (adc_InitMCH(ADC_USECHANNEL0 + ADC_USECHANNEL1 + .....,  
                  ADCMODE_RANGE_2VREF,  
                  ADCMODE_UNSIGNEDCODING)) {  
    .....  
    adc_data = adc_ReadMCH();  
    for (i=0; i<8; i++)  
        printf("A/D channel %d = %d", 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 resistors should \leq 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

```
unsigned char buf[3] = {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 byte buffer to put read bytes
 - **size**: the number of bytes to read

```
unsigned char buf[3];  
  
i2c_Receive(0x30, buf, 3); //read 3 bytes from an I2C  
                           //device with address 0x30
```

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 byte 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 byte buffer to put read bytes
 - **size:** the number of bytes to read

Master API

```
unsigned char buf[2];  
  
// first write 0x02 to an I2C device with address 0x70  
// and then restart to read 2 bytes back  
i2c_SensorRead(0x70, 0x02, buf, 2);
```

```
unsigned char cmd[2] = {0x32, 0x33};  
unsigned char buf[6];  
  
// 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);
```

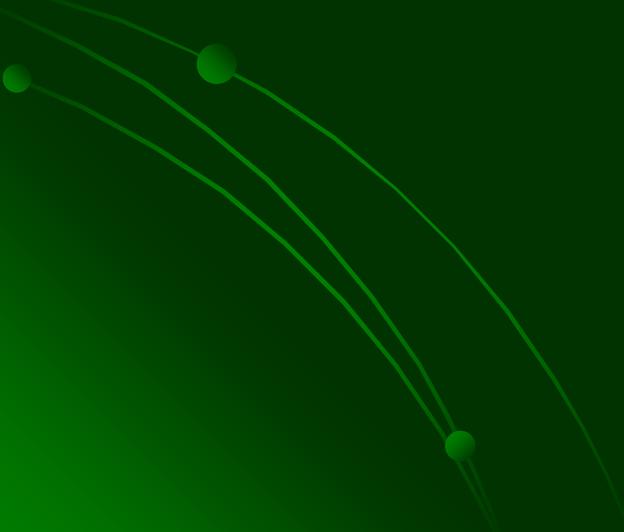
Remarks on I²C Device Address

- 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 when booting.

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);
```

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 (.....) {  
    switch (i2c0slave_Listen()) {  
        case I2CSLAVE_START: //receive START signal  
            //action for START signal  
            break;  
        case I2CSLAVE_WRITEREQUEST: //request slave to write  
            //handle write request  
            break;  
        case I2CSLAVE_READREQUEST: //request slave to read  
            //handle read request  
            break;  
        case I2CSLAVE_END: //receive STOP signal  
            //action for STOP signal  
            break;  
    }  
    ..... //can do stuff here when listening  
}
```

Slave Read/Write API

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

```
.....  
case I2CSLAVE_WRITEREQUEST:  
    i2c0slave_Write(byte_value);  
    break;  
.....
```

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

```
.....  
case I2CSLAVE_READREQUEST:  
    data = i2c0slave_Read();  
    break;  
.....
```

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 (**$\pm 1\mu s$**)
 - 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 pins are used as **Servo Mode** (for RB-110/ RB-050, **RCSERVO_USEPINS17 ~ RCSERVO_USEPINS24** are invalid)
 - Other unused 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** = **RCSERVO_PINS1** ~ **RCSERVO_PINS24**, indicating which pin to set.
- For RB-110/RB-050, **RCSERVO_PINS17** ~ **RCSERVO_PINS24** are invalid.
- **servo_model** can be
 - **RCSERVO_KONDO_KRS78X**: for KONDO KRS-786/788 servos

Configure Servo Setting

- **Method 1:** (cont.)
 - **servo_model** can be (cont.)
 - **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
 - `RCSERVO_DMP_RS0263`: for DMP RS-0263 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 configure the servo 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 `0xffffffffL` 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 `0xffffffffL` if fails to read feedback on the corresponding pin, or if the servo is with no feedback
 - for RB-100, `frame[24~31]` are reserved; for RB-110/RB-050, `frame[16~31]` are reserved.

Manipulate Servo: Capture Mode

```
rcservo_EnterCaptureMode();  
.....  
//read position feedback from PWM pin S3  
unsigned long pos = rcservo_CapOne(RCSERVO_PINS3);  
.....  
//read position feedback from all servo-mode pins  
unsigned long motion_frame[32];  
rcservo_CapAll(motion_frame);  
  
printf("position feedback on PWM pin S3 is  
      equal to %lu microsecond\n", motion_frame[2]);
```

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
unsigned long motion_frame[32];
rcservo_ReadPositions(RCSERVO_USEPINS1 + RCSERVO_USEPINS3,
                      0, //normally = 0
                      motion_frame);

printf("position feedback on PWM pins S1 and S3 are
       equal to %lu and %lu microseconds\n",
       motion_frame[0], motion_frame[2]);
```

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] = 0L** indicates the corresponding servo to remain at its position.

Manipulate Servo: Action Playing Mode

```
rcservo_EnterPlayMode();  
.....  
//move servo on PWM pin S2 to position 1500us in 500ms  
rcservo_MoveOne(RCSERVO_PINS2, 1500L, 500);
```

```
rcservo_EnterPlayMode();  
.....  
//move simultaneously both servos on PWM pins S1 and S3 to  
//position 1500us in 500ms  
  
unsigned long motion_frame[32] = {0L};  
  
motion_frame[0] = 1500L;  
motion_frame[2] = 1500L;  
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();  
.....  
unsigned long motion_frame[32] = {0L};  
//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();  
.....  
unsigned long buf[32];  
unsigned long motion_frame[32] = {0L};  
//here set up the content of motion_frame[] for playing  
.....  
rcservo_SetAction(motion_frame, 500); //play motion in 500ms  
while (rcservo_PlayAction() != RCSERVO_PLAYEND) {  
    rcservo_GetAction(buf);  
    printf("Servo on pin S1 is moving to %lu\n", 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();  
.....  
unsigned long PWM_period = 10000L; //10000us  
unsigned long PWM_duty   = 1500L; //1500us  
unsigned long count       = 100L;  
rcservo_SendPWM(pin, //RCSERVO_PINS1 or .....  
                 PWM_period, PWM_duty, count);  
while (!rcservo_IsPWMCCompleted(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 `0L` if pulses have stopped
 - return `0xffffffffL` for continuous pulses

Manipulate Servo: PWM Mode

```
rcservo_EnterPWMMode();  
.....  
unsigned long PWM_period = 10000L; //10000us  
unsigned long PWM_duty   = 1500L;  //1500us  
  
rcservo_SendCPWM(pin, //RCSERVO_PINS1 or .....  
                  PWM_period, PWM_duty);  
.....  
//do something when sending PWM  
.....  
rcservo_StopPWM(pin);
```

GPIO Functions

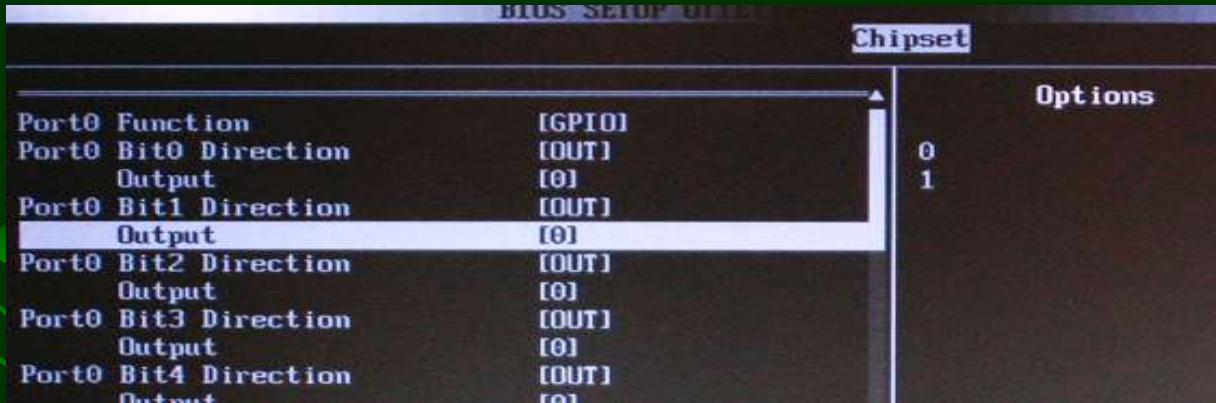
- API to control GPIO-mode pins
 - `rcservo_OutPin(pin, value)`: set GPIO pin to output HIGH or LOW
 - `pin` = `RCSERVO_PINS1` or `RCSERVO_PINS2` or
 - `value` = `0` (output LOW) or `1` (output HIGH)
 - `rcservo_InPin(pin)`: read input from GPIO pin
 - Return `0` if it read LOW, and `1` if it read HIGH
- 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), 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.

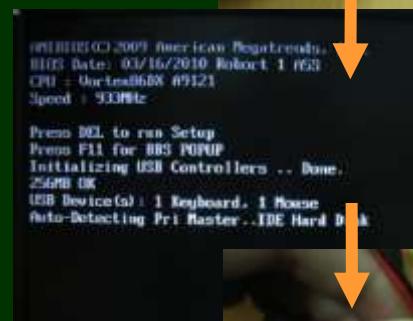
Remarks for KONDO KRS-4014

- Example: Make KRS-4014 work on RB-110.

- STEP 1: turn on the system power of RoBoard first



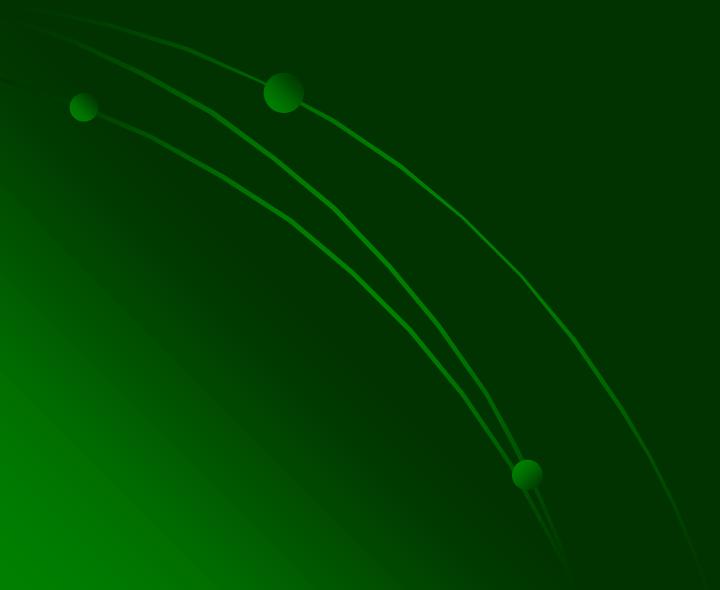
- STEP 2: wait the BIOS screen appeared



- STEP 3: turn on the servo power for KRS-4014



PWM lib



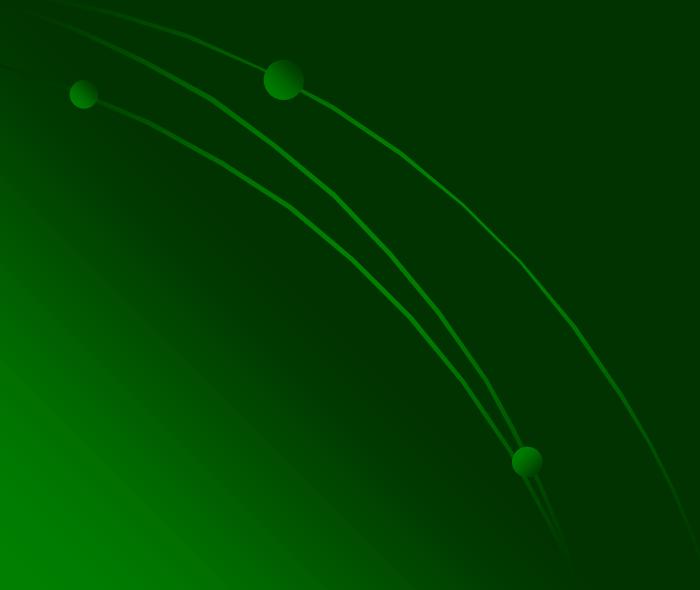
PWM lib Usage

- Allow users to employ complete RoBoard's PWM features
 - Control polarity of PWM waveform
 - Control PWM resolution (**maximum resolution = 20ns**)
 - Enable PWM interrupt
 -

PWM lib Usage

- See **pwm.h** and **pwm_dx.h** for available API.
 - To use PWM lib, a detailed understanding of RoBoard's H/W PWM functions is required.
- **WARNING!** Do not use PWM lib when RC Servo lib is in use.
 - You should use PWM functions of RC Servo lib instead.

COM Ports



RB-100/RB-110 Native COM Ports

- COM1~COM4 can be used as standard COM ports in WinXP, Linux, and DOS
- Max speed
 - RB-100: **115200** bps
 - 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-110/RB-050

Native COM Ports

- RB-110/RB-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 = $13 \times$ the original baudrate
 - For example, if boosting mode of COM3 is enabled and its baudrate is set to 38400 bps, the real baudrate is $38400 \times 13 = 500\text{K}$ bps.
 - In boosting mode, the maximum baudrate is $57600 \times 13 = 750\text{Kbps}$ (115200×13 is not allowed)

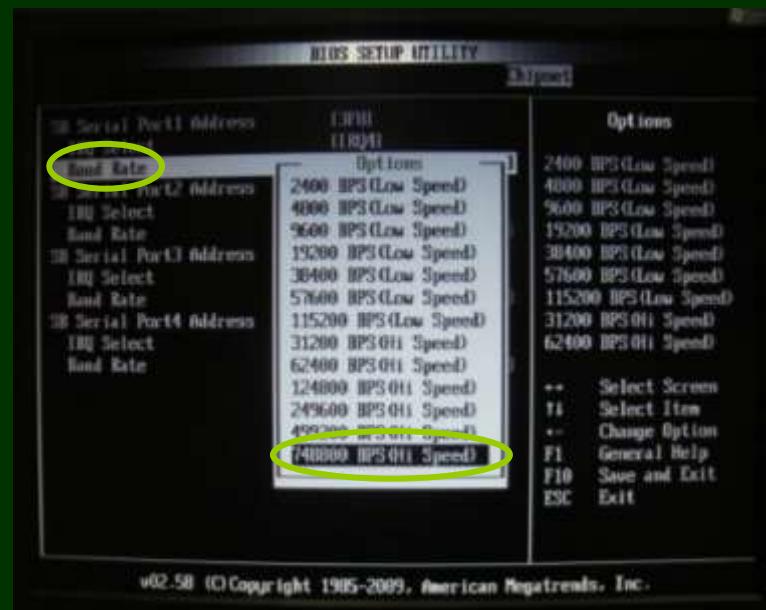
How to Enable Boosting Mode of RB-110/RB-050 Native COM Ports

- **Method 1: Using BIOS**

- STEP 1: Go to RB-110/RB-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-110/RB-050 Native COM Ports

- **Method 2:** Using `rbcn.exe` in RoBoKit.
 - Run `rbcn.exe` directly to see its usage
- **Method 3:** Using the API of COM lib

```
io_init();  
.....  
com_EnableTurboMode(0); //enable boosting mode of COM1  
com_EnableTurboMode(2); //enable boosting mode of COM3  
.....  
com_DisableTurboMode(2); //disable boosting mode of COM3  
.....  
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>

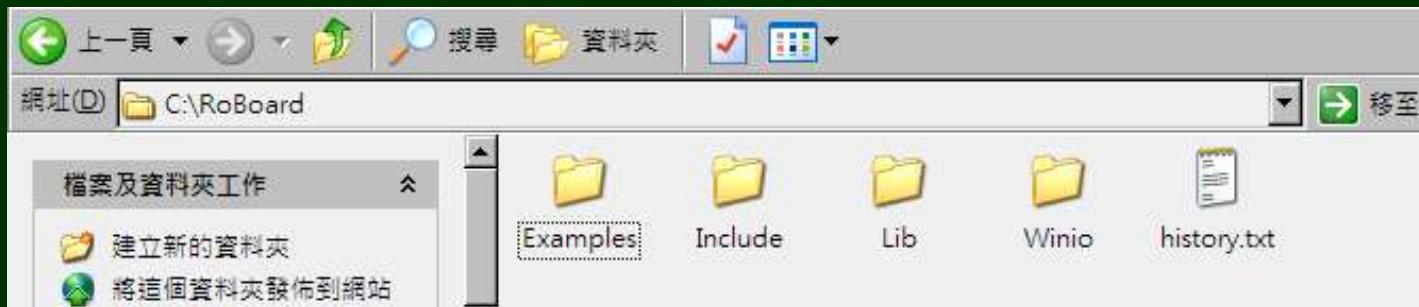
Installation

(for Visual Studio 2005/2008)



Setup in VS2005/2008

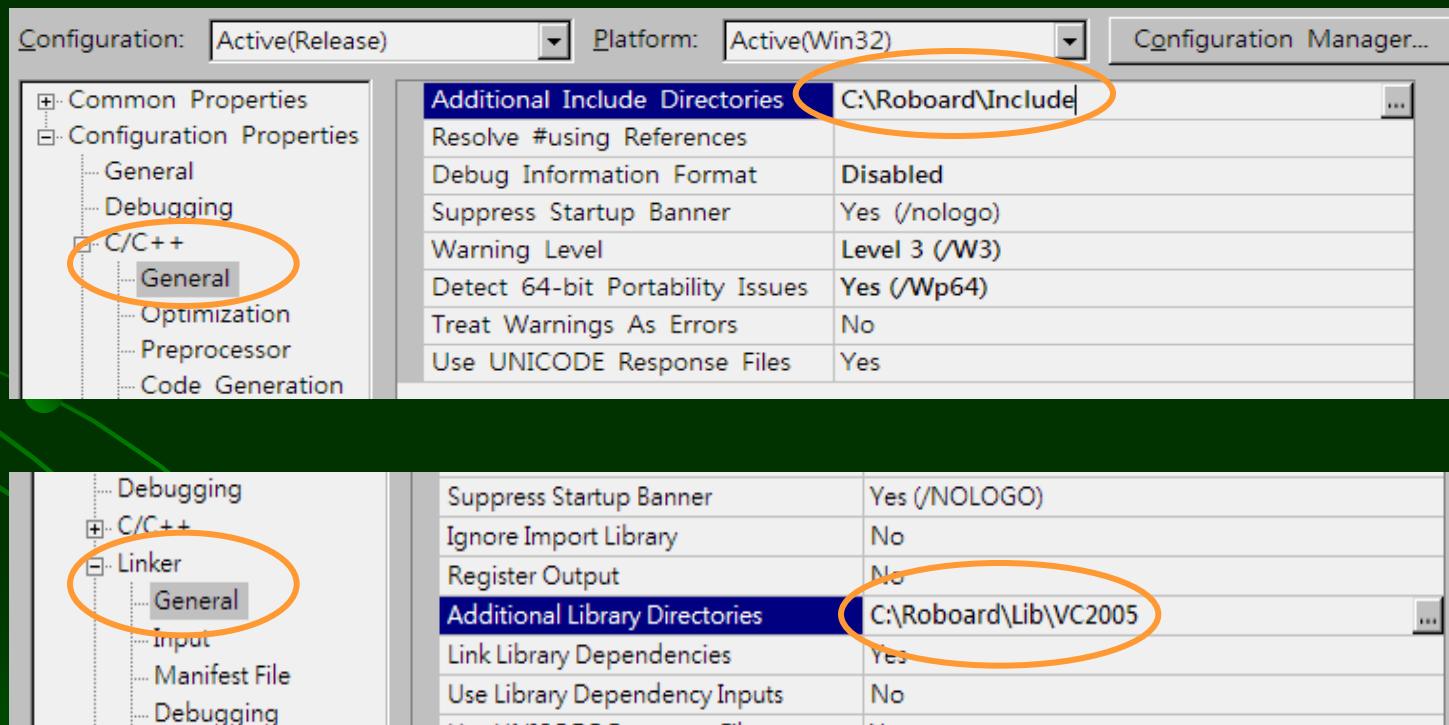
- Decompose RoBoIO bin zip-file to, e.g., C:\RoBoard



- **Examples:** sample codes for RoBoIO library
- **Include:** include files of RoBoIO library
- **Lib:** binary files of RoBoIO library
- **Winio:** needed when using RoBoIO under WinXP

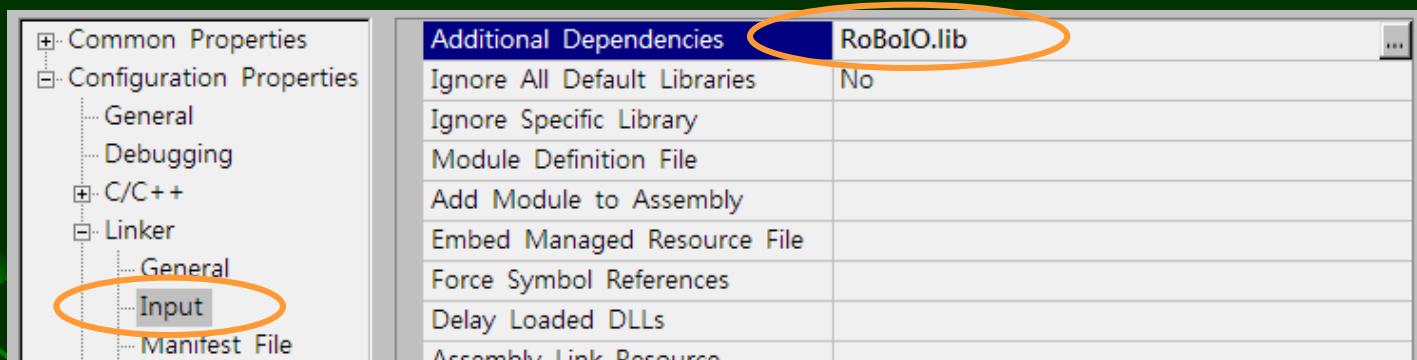
Setup in VS2005/2008

- Setting RoBoIO in your VC2005/2008 project



Setup in VS2005/2008

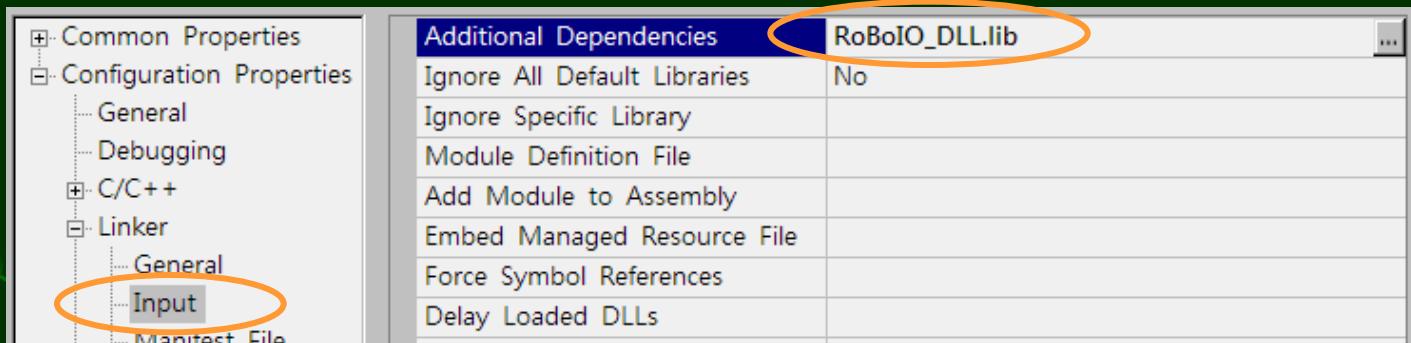
- Setting RoBoIO in your VC2005/2008 project (cont.)
 - If using the static version



- VC2005/2008 compatibility: need to use the correct version of lib files for VC2005 & VC2008

Setup in VS2005/2008

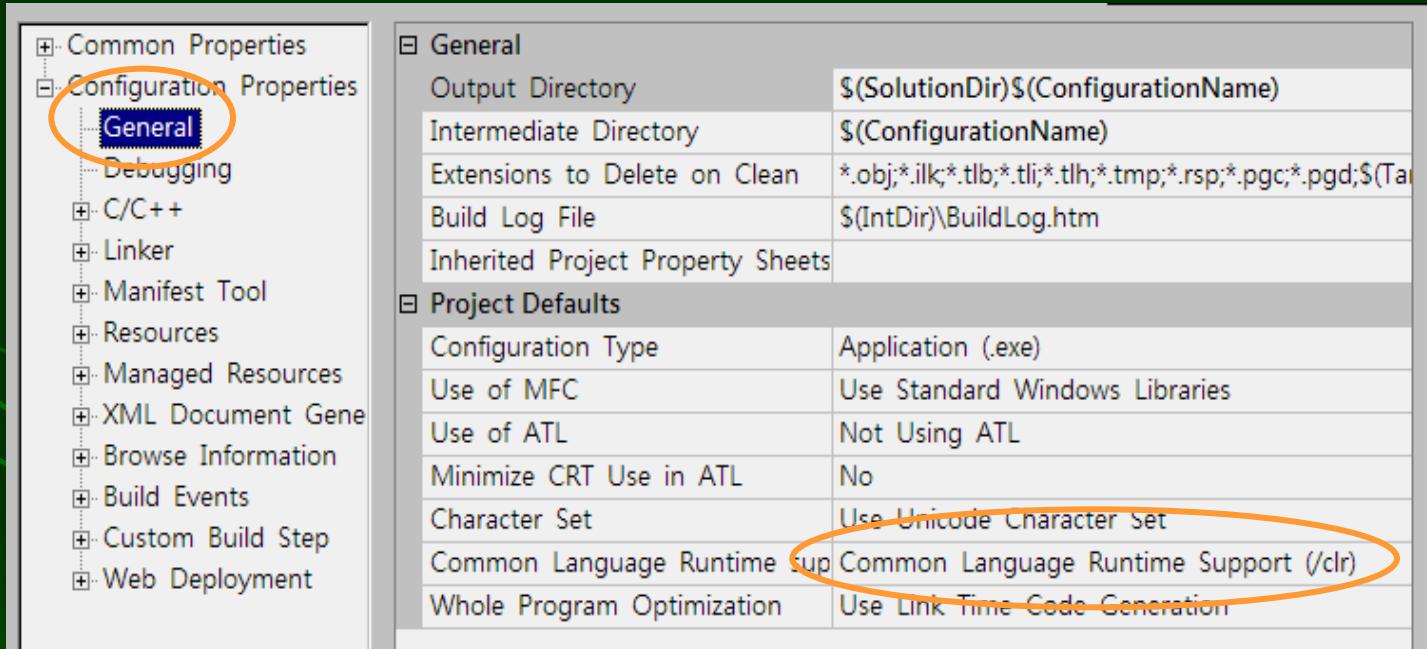
- Setting RoBoIO in your VC2005/2008 project (cont.)
 - If using the DLL version



- The DLL version uses the **stdcall** calling convention
(compatible to VB, C#, Delphi, Matlab, and)

Setup in VS2005/2008

- If you use .NET



Setup in VS2005/2008

- To run your RoBoIO application on WinXP:
 1. First install **VC2005/2008 SP1 runtime redistribution** in RoBoard
 2. Copy your application to RoBoard's storage (the MicroSD or USB storage)
 3. Copy all files in **RoBoard\Winio** to your application directory or Window's **System32** directory

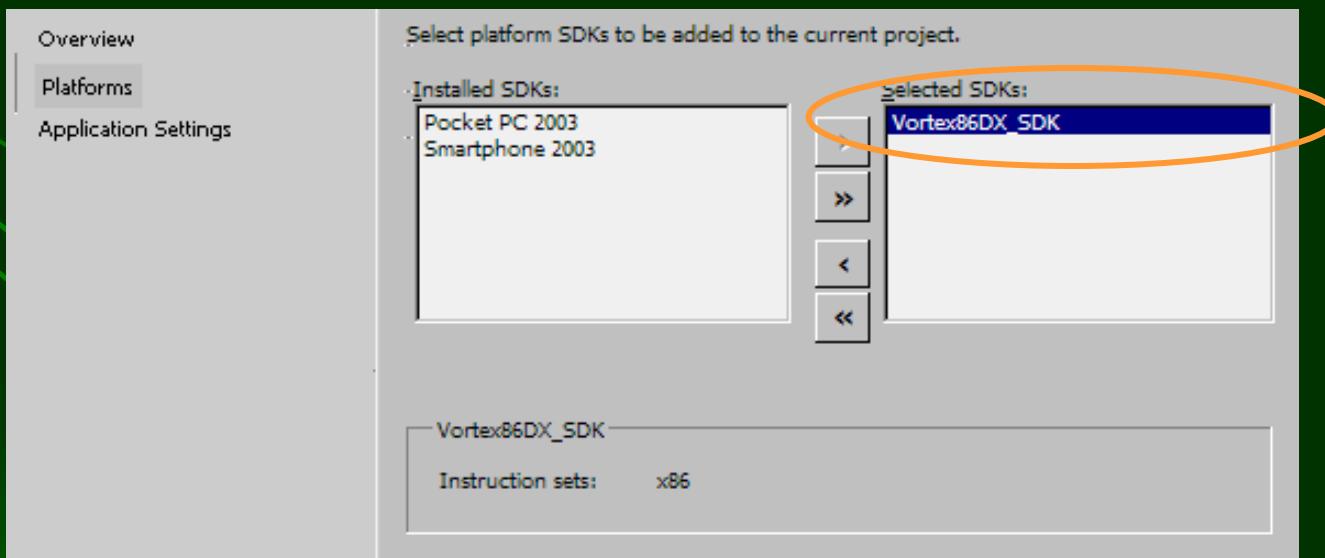
Setup in VS2005/2008

- 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.

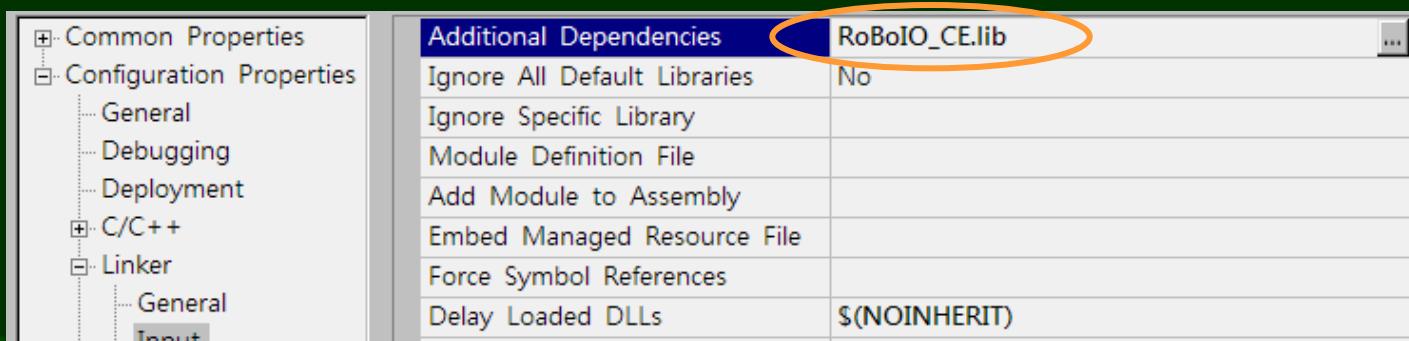
Setup in VS2005/2008

- If you want to develop WinCE RoBoIO application
 - Download **Vortex86DX WinCE 6.0 SDK** from RoBoard website, and install it.
 - In VS Smart Device Project Wizard, select **Vortex86DX_SDK**:

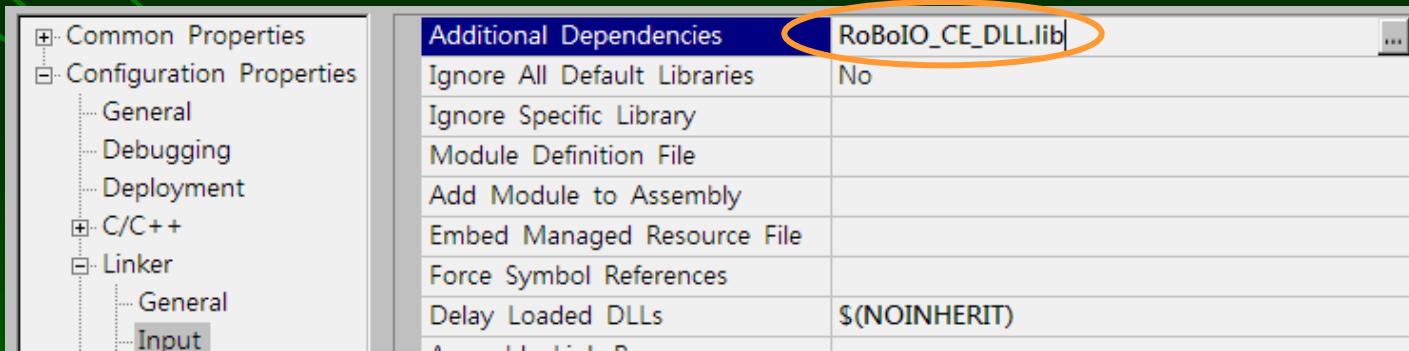


Setup in VS2005/2008

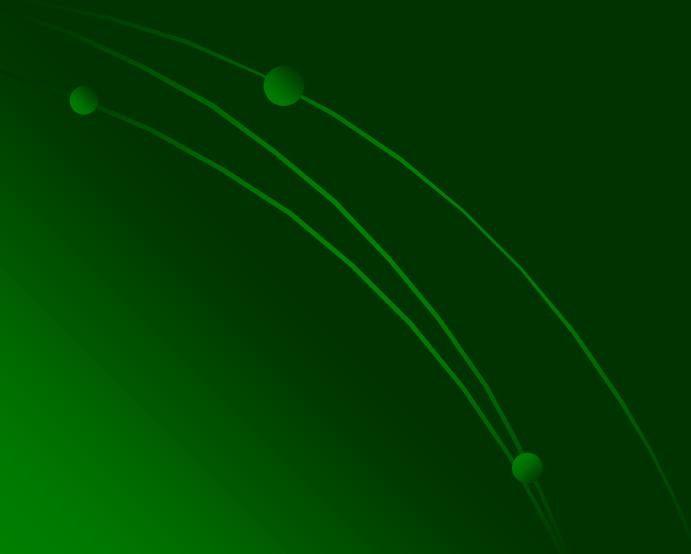
- Note that the filenames are different for WinCE.
 - static version



- DLL version



Installation (for Linux)



Setup in Linux

- Make the RoBoIO lib
 - STEP 1: Ensure the gcc environment has been installed.
 - As an example, in Ubuntu 9.0.4, you can type

```
sudo apt-get install libncurses5-dev  
sudo apt-get install gcc g++ make
```

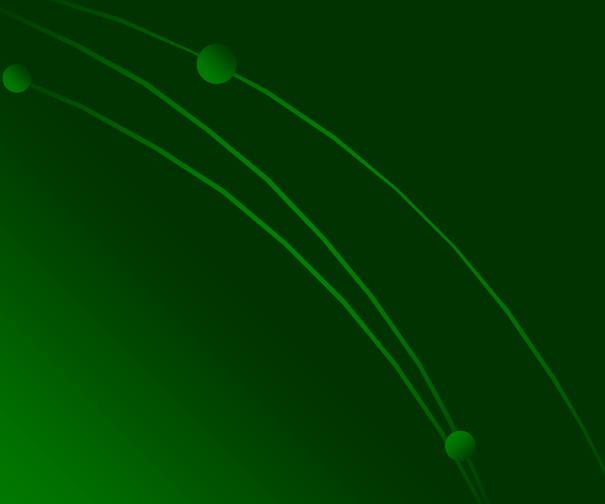
to install a gcc environment for RoBoIO compilation.



Setup in Linux

- Make the RoBoIO lib (cont.)
 - STEP 2: Decompress the RoBoIO linux src to a directory.
 - STEP 3: Going into the directory with **Makefile**, type
make
and you will get the static RoBoIO lib: **libRBI0.a**
- Remarks: You should login with root to run RoBoIO applications.

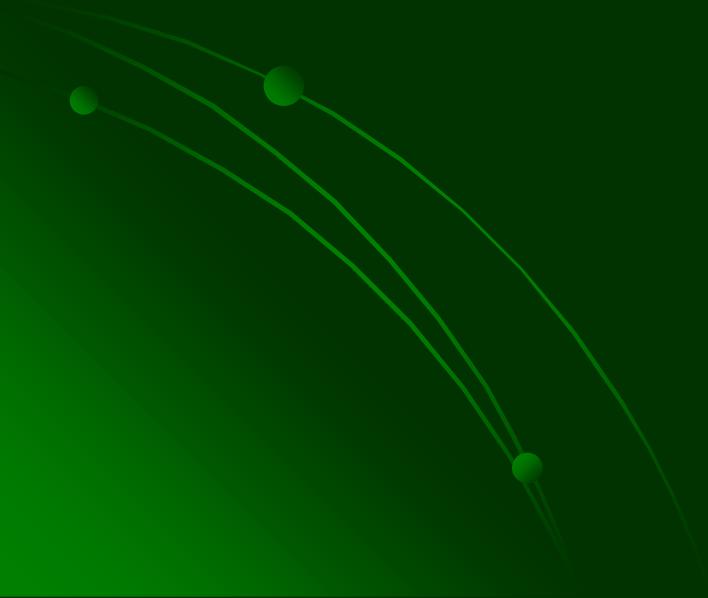
Installation (Other Platforms)



Other Supported Platforms

- If you need to setup RoBoIO in the following platforms, please email to tech@roboard.com
 - DJGPP
 - Watcom C++
 - Borland C++ 3.0~5.02

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, Matlab, ...
 - **Libraries:** OpenCV, SDL, ...
 - **IDE:** Visual Studio, Dev-C++, Code::Blocks, ...
 - **GUI (if needed):** MFC, Windows Forms, GTK, ...

Introduction

- **Rich I/O interfaces** ⇒ Various sensors & devices can be employed as RoBoard's senses.
 - **A/D, SPI, I²C:** 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

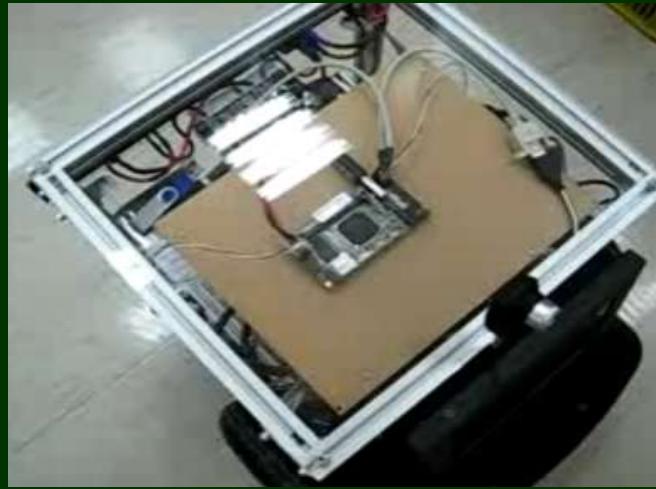
Rich I/O (using RoBoIO) + Rich
resources on PC



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