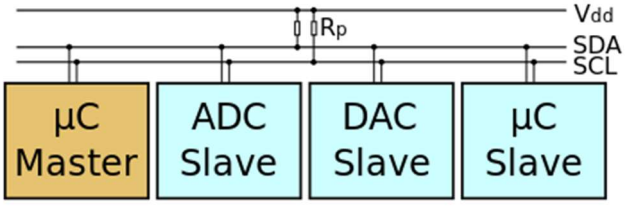
ปฏิบัติการบน RaspberryPi ครั้งที่ 6: การเชื่อมต่อกับอุปกรณ์ผ่าน i2c

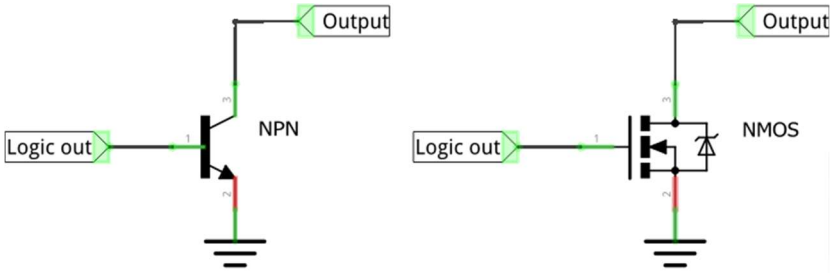
หลักการพื้นฐานของบัส i2c

i2c (อ่านว่า ไอ-สแควร์-ซี) เป็นบัสสื่อสารอนุกรมแบบซิงโครนัส ที่รองรับการเชื่อมต่อระหว่างอุปกรณ์ควบคุม(master) และอุปกรณ์ที่ถูกควบคุม(slave) สามารถมีอุปกรณ์หลายตัวต่ออยู่บนบัสเดียวกัน i2c ต้องการสัญญาณ 2 เส้น SDA ใช้สำหรับรับส่งข้อมูล ส่วน SCL ใช้เพื่อให้อุปกรณ์ควบคุมส่งสัญญาณนาฬิกาเพื่อกำหนดความเร็วในการรับส่งข้อมูล

ขาสัญญาณทั้งสองของอุปกรณ์ควบคุมและอุปกรณ์ที่ถูกควบคุมมีลักษณะเป็น open-collector ซึ่งเป็นทรานซิสเตอร์แบบ NPN หรือ CMOS แบบ NMOS โดยเมื่อมีกระแสที่ขา base ของทรานซิสเตอร์ หรือมีความต่างศักย์ตกคร่อมที่ขา gate และ source ของ NMOS จะทำให้เกิดกระแสไหลผ่านขา collector (หรือขา drain ของ NMOS) หากเราต่อตัวต้านทาน (ปกติจะใช้ค่า 4.7kohm ที่ 5v-VCC) เข้ากับขาเอาต์พุตและไฟเลี้ยง (VCC) เราจะได้ลอจิกขาออกเป็น 0 แต่ถ้าไม่มีกระแสที่ base (หรือความต่างศักย์ที่ gate) เราจะได้ลอจิกขาออกเป็น 1 และในจังหวะที่ไม่มีกระแส/ความต่างศักย์นี้ วงจรภายในสามารถใช้ขาสัญญาณ output เป็นขาอินพุตเพื่อรับข้อมูลได้ด้วย ทำให้ขา SDA และ SCL ของบัส i2c นี้สามารถทำหน้าที่ได้ทั้งเป็นอินพุตและเอาต์พุตในเวลาเดียวกัน ระดับสัญญาณลอจิกในขณะที่ไม่มีข้อมูลส่งเป็น 1



ลักษณะการเชื่อมต่ออุปกรณ์บนบัส i2c



ลักษณะวงจรสัญญาณขาออกแบบ open-collector (ซ้าย) และ open-drain (ขวา)

Mode	*Maximum speed*	*Drive/Direction*
Standard mode (SM)	100 kbps	open-drain / bidirectional
Fast mode (FM)	400 kbps	
Fast mode plus (FM+)	1Mbps	
High-speed mode (HS)	3.4Mbps	
Ultra-fast mode (UFM)	5Mbps	push-pull / unidirectional

ความเร็วสัญญาณนาฬิกาของบัส i2c ในมาตรฐานต่างๆ

ในการรับส่งข้อมูลนั้น อุปกรณ์ควบคุมมีหน้าที่ให้สัญญาณนาฬิกา SCL แก่อุปกรณ์ที่ถูกควบคุม ดังนั้นวงจรของทั้งสองฝั่งจะต้องรองรับความเร็วที่อุปกรณ์ควบคุมทำหน้าที่ส่งออกไปได้ สำหรับ MCU ของ Raspberry Pi สามารถส่งสัญญาณนาฬิกา SCL สูงสุดได้ที่ 400kbps (ปกติอยู่ที่ 100kbps)

อุปกรณ์แต่ละตัวที่เชื่อมต่ออยู่บนบัส i2c นั้น จะมีการกำหนดตัวเลขที่อยู่ (address) ซึ่งมีขนาด 7 บิต หรือ 10 บิต โดยจะประกอบอยู่ในไบต์เลขที่อยู่ที่มีขนาด 8 หรือ 16 บิต (กรณีที่เลขที่อยู่แบบ 10 บิต) โดยบิตต่ำสุด (LSB) เป็นบิตกำหนดการเขียน/อ่าน โดยใช้ 0 เพื่อเขียน (ส่งข้อมูล) หรือ 1 เพื่ออ่าน (รับข้อมูล)

การส่งข้อมูลจะเริ่มต้นจากอุปกรณ์ควบคุม ส่งไบต์เลขที่อยู่ออกไป อุปกรณ์ปลายทางที่มีเลขที่อยู่ตรงกันก็จะส่งสัญญาณตอบรับ (acknowledge) กลับมา จากนั้น อุปกรณ์ควบคุมก็จะส่งข้อมูลไปทีละไบต์ (ซึ่งในแต่ละครั้งอุปกรณ์ปลายทางก็จะยืนยันกลับมา) จนกระทั่งหมดชุดข้อมูล ในกรณีที่อุปกรณ์ปลายทางมีเรจิสเตอร์ในการทำงานหลายตัว ไบต์ถัดจากไบต์เลขที่อยู่ จะเป็นไบต์กำหนดตัวเรจิสเตอร์ซึ่งเรียกว่าไบต์คำสั่ง (command byte) จากนั้นจึงตามด้วยชุดข้อมูลที่จะเขียน ซึ่งโดยทั่วไปก็มักจะมีขนาด 8 หรือ 16 บิต

สำหรับการอ่านข้อมูลนั้น อุปกรณ์บางตัวที่มีกลไกการทำงานไม่ซับซ้อน มีเพียงเรจิสเตอร์ตัวเดียวเพื่อใช้ในการพักข้อมูล การอ่านจะเริ่มต้นจากอุปกรณ์ควบคุมส่งไบต์เลขที่อยู่ออกไป จากนั้นก็จะรอให้อุปกรณ์ปลายทางส่งข้อมูลในเรจิสเตอร์ของตนกลับออกมาจนครบทุกไบต์ เป็นอันจบการติดต่อ แต่สำหรับอุปกรณ์อีกจำนวนหนึ่งที่มีเรจิสเตอร์หลายตัวภายใน การอ่านข้อมูลจะเริ่มต้นด้วยอุปกรณ์ควบคุมส่งไบต์เลขที่อยู่ และตามด้วยคอมมานต์ไบต์ ซึ่งกำหนดตัวเรจิสเตอร์ที่ต้องการจะเขียนหรืออ่าน จากนั้นจึงเป็นการรอรับข้อมูลจากเรจิสเตอร์ที่อุปกรณ์ปลายทางจะส่งกลับมา

การจัดการบัส i2c บน Raspberry Pi

ตัว raspberry pi โมเดล 3B มีบัส i2c ทั้งหมด 2 ช่องด้วยกัน โดยมีการใช้ขาร่วมกันกับพอร์ตดังต่อไปนี้

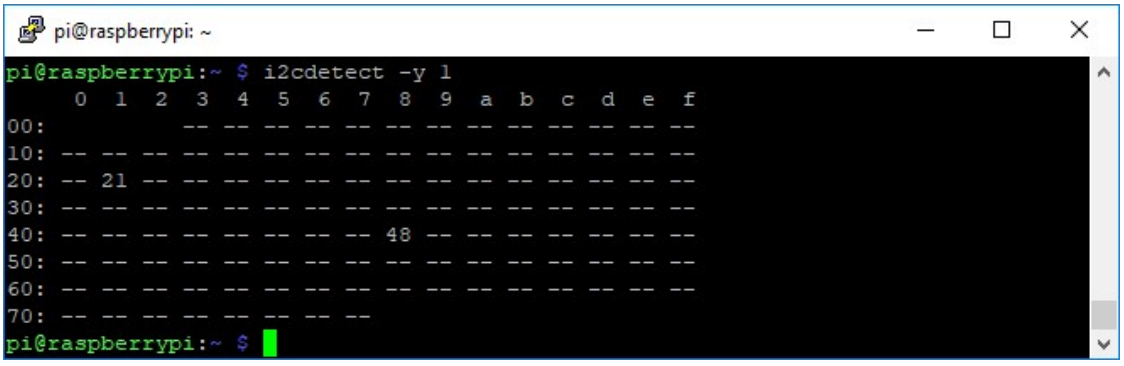
	SDA	SCL
i2c บัส 0	GPIO 0	GPIO 1
i2c บัส 1	GPIO 2	GPIO 3

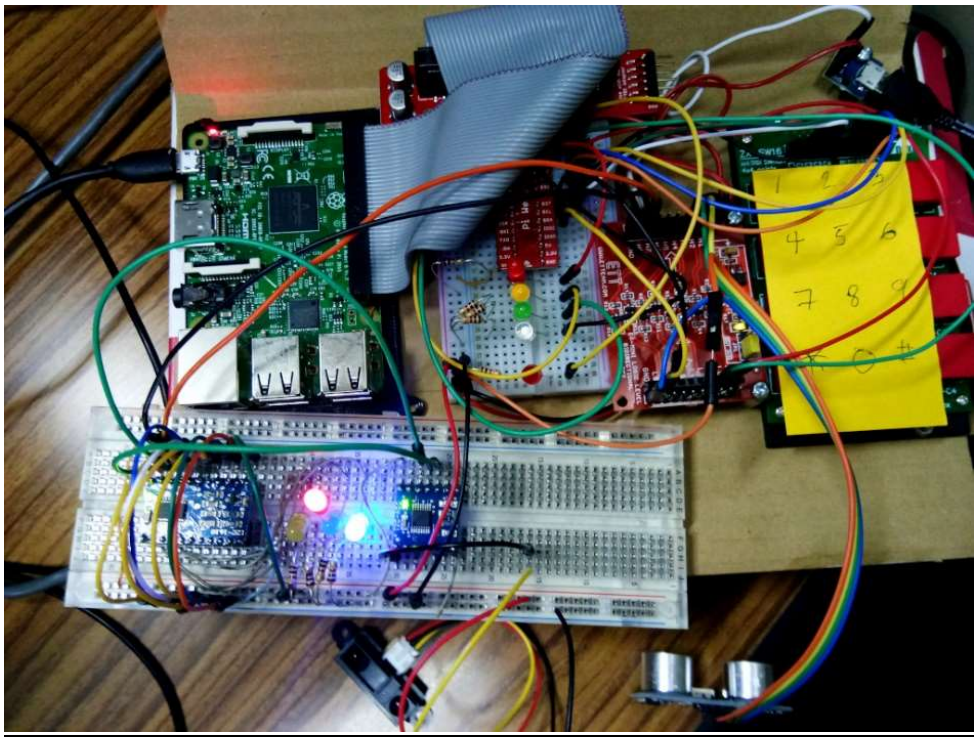
หมายเหตุ *บัส i2c ต้องการ R pull-up เนื่องจากพอร์ต i2c ของอุปกรณ์เป็น open-collector แต่ในปฏิบัติการครั้งนี้ไม่มีการต่อ R pull-up เนื่องจากบอร์ด Raspberry Pi มีการต่อ R pull-up ขนาด 1.8kohm เข้ากับ 3.3v ไว้แล้ว*

ทั้งนี้ ตัว Raspberry Pi OS นั้น รองรับการเชื่อมต่อพอร์ต i2c ในรูปแบบของการจัดการ I/O ตามปกติของลินุกซ์ ซึ่งจะมองเห็นอุปกรณ์ i2c อยู่ในโครงสร้างไฟล์ของระบบ เราจึงสามารถขอเชื่อมอุปกรณ์ และเขียนอ่านกับอุปกรณ์ในลักษณะเช่นเดียวกันกับ ไฟล์และอุปกรณ์รับส่งข้อมูลโดยทั่วไปได้อย่างสะดวก

สำหรับ Raspberry Pi OS ที่นักศึกษาใช้งานอยู่นั้น ได้มีการติดตั้งไลบรารี pigpio ซึ่งรองรับการเชื่อมต่อแบบ i2c มาแล้ว เพื่อตรวจสอบว่าตัว raspberry pi สามารถติดต่อกับอุปกรณ์ปลายทางผ่าน i2c ได้หรือไม่ เมื่อนักศึกษาต่อวงจรเชื่อมระหว่างตัว raspberry pi และอุปกรณ์ปลายทางเสร็จแล้ว ให้ใช้คำสั่ง i2cdetect เพื่อตรวจสอบค้นหาอุปกรณ์ที่เชื่อมต่อด้วย สำหรับ raspberry pi โมเด็ลที่นักศึกษาใช้ทำการทดลอง ให้กำหนดอาร์กิวเมนต์เป็น -y 1 ดังรูป

!!!ข้อควรระวัง raspberry pi ใช้ VCC 3.3 โวลต์ หากวงจรปลายทางใช้ 5 โวลต์ จะต้องมีการเชื่อมต่อวงจร Logic Shifter เพื่อป้องกันไม่ให้บัส i2c และบอร์ด raspberry pi เสียหาย





จากรูปข้างบน เป็นการทดลองต่อวงจร GPIO ขนาด 16 บิต และวงจร ADC ขนาด 8 ช่องสัญญาณผ่านบัส i2c ตัววงจร GPIO จากโรงงานกำหนดค่าเลขที่อยู่เริ่มต้นอยู่ที่ 0x20 (และยังสามารถกำหนดเลขที่อยู่ย่อยได้อีก 3 บิต เพื่อให้สามารถใช้วงจรนี้ได้ถึง 8 ชุดบนบัสเดียวกัน โดยในรูปตัวอย่างข้างบนนี้ได้มีการเปลี่ยนเลขที่อยู่ย่อย โดยการบัดกรีแก้ไขบนลายวงจรในจุดที่กำหนด ให้เป็น 0x21 เปลี่ยน A0 เป็น 1 แทนจากเดิมที่เป็น 0) และวงจร ADC มีการกำหนดเลขที่อยู่เริ่มต้นอยู่ที่ 0x48 (วงจร ADC ตัวที่เห็นนี้สามารถกำหนดเลขที่อยู่ย่อยได้อีก 1 บิต)

การเขียนโปรแกรมจัดการบัส i2c ด้วย pigpio

pigpio มีฟังก์ชันจัดการ i2c กับพอร์ต i2c ของ Raspberry Pi ตามปกติ และยังมีฟังก์ชันจัดการกับพอร์ต GPIO ธรรมดาของ Raspberry Pi เพื่อจำลองให้เป็น i2c (เนื่องจากเป็นกลไกทางซอฟต์แวร์ จึงเหมาะสมในการทำงานที่ความเร็วต่ำๆ เท่านั้น)

ฟังก์ชันจัดการกับพอร์ต i2c ของ Raspberry Pi เฉพาะที่น่าสนใจ มีดังนี้

```
int i2cOpen(unsigned i2cBus, unsigned i2cAddr, unsigned i2cFlags)
```

เริ่มต้นการใช้งานพอร์ต i2c ที่ต้องการ

i2cbus	หมายเลขบัส i2c ที่ต้องการเปิดใช้งาน (ตัวอย่าง Raspberry Pi 3 จะมีบัส0 และ บัส1)	
i2cAddr	หมายเลขประจำอุปกรณ์บนบัส ซึ่งสามารถตรวจสอบได้จากคำสั่ง i2cdetect ในเทอร์มินัล	
i2cFlags	(ในที่นี้ยังมิได้กำหนดมาใช้งาน) กำหนดให้เป็น 0	
ค่ากลับคืน	ค่า 0 หรือมากกว่า	หมายเลขแฮนเดิลประจำอุปกรณ์
	PI_BAD_I2C_BUS	ไม่พบหมายเลขพอร์ต
	PI_BAD_I2C_ADDR	ไม่พบหมายเลขประจำอุปกรณ์
	PI_BAD_I2C_FLAGS	ค่าแฟลกไม่ถูกต้อง
	PI_NO_HANDLE หรือ PI_I2C_OPEN_FAILED	เปิดใช้งานไม่สำเร็จ

```
int i2cClose(unsigned handle)
```

เริ่มต้นการใช้งานพอร์ต i2c ที่ต้องการ

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
ค่ากลับคืน	OK	จบการทำงานได้ตามปกติ
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

```
int i2cWriteByte(unsigned handle, unsigned bVal)
```

เขียนข้อมูลขนาด 1 ไบต์ไปยังอุปกรณ์

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
bVal	ข้อมูลที่จะเขียน	
ค่ากลับคืน	OK	จบการทำงานได้ตามปกติ
	PI_BAD_PARAM	ข้อมูลที่ส่งไม่ถูกต้อง
	PI_I2C_WRITE_FAILED	ไม่สามารถส่งข้อมูลไปยังอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

```
int i2cReadByte(unsigned handle)
```

อ่านข้อมูลขนาด 1 ไบต์จากอุปกรณ์

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
ค่ากลับคืน	ค่าศูนย์หรือมากกว่า	ข้อมูลแบบ unsigned ขนาด 8 บิต
	PI_I2C_READ_FAILED	ไม่สามารถอ่านข้อมูลจากยังอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

```
int i2cWriteByteData(unsigned handle, unsigned i2cReg, unsigned bVal)
```

เขียนข้อมูลขนาด 1 ไบต์ไปยังอุปกรณ์ ใช้สำหรับอุปกรณ์ i2c ที่มีไบต์คำสั่ง (command byte) กำกับข้อมูลที่จะต้องเขียน

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
i2cReg	ไบต์คำสั่งที่อุปกรณ์กำหนด	
bVal	ข้อมูลที่จะเขียนลงไปยังเรจิสเตอร์(ที่กำหนดตำแหน่งด้วยไบต์คำสั่ง) ขนาด 8 บิต	
ค่ากลับคืน	OK	จบการทำงานได้ตามปกติ
	PI_BAD_PARAM	ข้อมูลที่ส่งไม่ถูกต้อง
	PI_I2C_WRITE_FAILED	ไม่สามารถส่งข้อมูลไปยังอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

```
int i2cWriteWordData(unsigned handle, unsigned i2cReg, unsigned wVal)
```

เขียนข้อมูลขนาด 2 ไบต์ไปยังอุปกรณ์ ใช้สำหรับอุปกรณ์ i2c ที่มีไบต์คำสั่ง (command byte) กำกับข้อมูลที่จะต้องเขียน

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
i2cReg	ไบต์คำสั่งที่อุปกรณ์กำหนด	
wVal	ข้อมูลที่จะเขียนลงไปยังเรจิสเตอร์(ที่กำหนดตำแหน่งด้วยไบต์คำสั่ง) ขนาด 16 บิต	
ค่ากลับคืน	OK	จบการทำงานได้ตามปกติ
	PI_BAD_PARAM	ข้อมูลที่ส่งไม่ถูกต้อง

	PI_I2C_WRITE_FAILED	ไม่สามารถส่งข้อมูลไปยังอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

```
int i2cReadByteData(unsigned handle, unsigned i2cReg)
```

อ่านข้อมูลขนาด 1 ไบต์จากอุปกรณ์ ใช้สำหรับอุปกรณ์ i2c ที่มีไบต์คำสั่ง (command byte) กำกับข้อมูลที่จะต้องอ่าน

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
i2cReg	ไบต์คำสั่งที่อุปกรณ์กำหนด	
ค่ากลับคืน	ค่าศูนย์หรือมากกว่า	ข้อมูลที่อ่านได้แบบ unsigned ขนาด 8 บิต
	PI_BAD_PARAM	ข้อมูลที่ส่งไม่ถูกต้อง
	PI_I2C_READ_FAILED	ไม่สามารถอ่านข้อมูลจากอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

```
int i2cReadWordData(unsigned handle, unsigned i2cReg)
```

อ่านข้อมูลขนาด 2 ไบต์จากอุปกรณ์ ใช้สำหรับอุปกรณ์ i2c ที่มีไบต์คำสั่ง (command byte) กำกับข้อมูลที่จะต้องอ่าน

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
i2cReg	ไบต์คำสั่งที่อุปกรณ์กำหนด	
ค่ากลับคืน	ค่าศูนย์หรือมากกว่า	ข้อมูลที่อ่านได้แบบ unsigned ขนาด 16 บิต
	PI_BAD_PARAM	ข้อมูลที่ส่งไม่ถูกต้อง
	PI_I2C_READ_FAILED	ไม่สามารถอ่านข้อมูลจากอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

```
int i2cProcessCall(unsigned handle, unsigned i2cReg, unsigned wVal)
```

เขียนข้อมูลขนาด 2 ไบต์จากอุปกรณ์ ใช้สำหรับอุปกรณ์ i2c ที่มีไบต์คำสั่ง (command byte) กำกับข้อมูลที่จะต้องเขียน และอ่านข้อมูลผลลัพธ์กลับมาในคราวเดียว (สำหรับกลไกการติดต่อ i2c กับอุปกรณ์ที่ต้องการคำสั่งเพื่อใช้เขียนข้อมูลไปและรับข้อมูลกลับในคราวเดียว

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
i2cReg	ไบต์คำสั่งที่อุปกรณ์กำหนด	
wVal	ข้อมูลขนาด 16 บิตที่จะเขียนไป	
ค่ากลับคืน	ค่าศูนย์หรือมากกว่า	ข้อมูลที่อ่านได้แบบ unsigned ขนาด 16 บิต
	PI_BAD_PARAM	ข้อมูลที่ส่งไม่ถูกต้อง
	PI_I2C_READ_FAILED	ไม่สามารถอ่านข้อมูลจากอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

```
int i2cWriteBlockData(unsigned handle, unsigned i2cReg, char *buf, unsigned count)
```

เขียนบล็อกข้อมูลขนาดไม่เกิน 32 ไบต์ไปยังอุปกรณ์ ใช้สำหรับอุปกรณ์ i2c ที่มีไบต์คำสั่ง (command byte) กำกับข้อมูลที่จะต้องเขียน

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
i2cReg	ไบต์คำสั่งที่อุปกรณ์กำหนด	
buf	ตัวชี้รับค่าอ้างอิงไปยังพื้นที่อะเรย์เก็บข้อมูลที่จะใช้ส่ง	
count	จำนวนไบต์ที่จะเขียน	
ค่ากลับคืน	ค่าศูนย์หรือมากกว่า	ข้อมูลที่อ่านได้แบบ unsigned ขนาด 16 บิต

	PI_BAD_PARAM	ข้อมูลที่ส่งไม่ถูกต้อง
	PI_I2C_WRITE_FAILED	ไม่สามารถส่งข้อมูลไปยังอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

```
int i2cReadBlockData(unsigned handle, unsigned i2cReg, char *buf)
```

อ่านบล็อกข้อมูลขนาดไม่เกิน 32 ไบต์ไปยังอุปกรณ์ ใช้สำหรับอุปกรณ์ i2c ที่มีไบต์คำสั่ง (command byte) กำกับข้อมูลที่จะต้องอ่าน

handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
i2cReg	ไบต์คำสั่งที่อุปกรณ์กำหนด	
buf	ตัวชี้รับค่าอ้างอิงไปยังพื้นที่อะเรย์เก็บข้อมูลที่จะรับ	
ค่ากลับคืน	ค่าศูนย์หรือมากกว่า	จำนวนไบต์ข้อมูลที่อ่านได้
	PI_BAD_PARAM	ข้อมูลที่ส่งไม่ถูกต้อง
	PI_I2C_READ_FAILED	ไม่สามารถรับข้อมูลจากอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

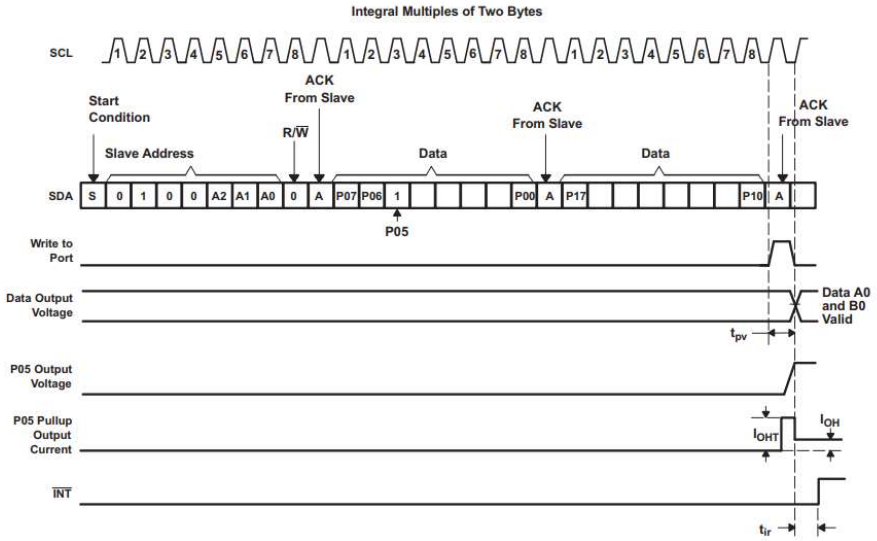
```
int i2cBlockProcessCall(unsigned handle, unsigned i2cReg,
                        char *buf, unsigned count)
```

เขียนบล็อกข้อมูลขนาดไม่เกิน 32 ไบต์ไปยังอุปกรณ์ที่มีการกำหนดไบต์คำสั่ง (command byte) กำกับข้อมูลที่จะต้องเขียน และรับบล็อกข้อมูลผลลัพธ์ขนาดไม่เกิน 32 ไบต์กลับมาจากอุปกรณ์ในคราวเดียว

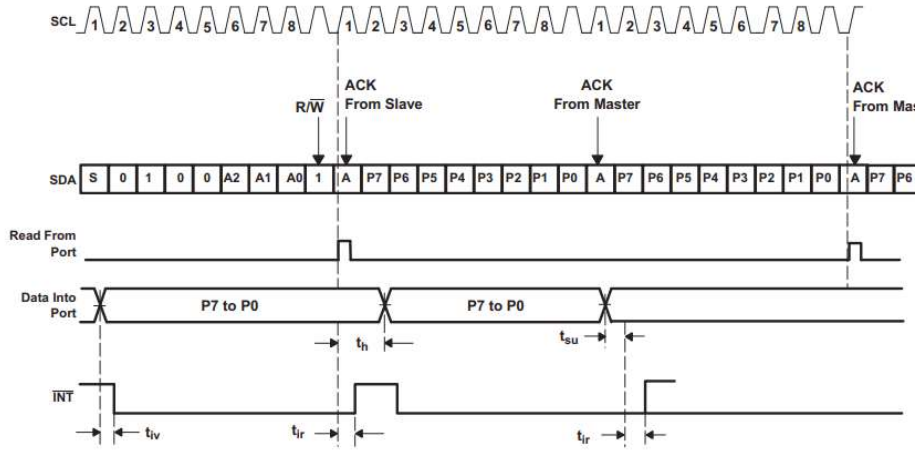
handle	หมายเลขแฮนเดิลประจำอุปกรณ์ที่ได้จาก i2cOpen()	
i2cReg	ไบต์คำสั่งที่อุปกรณ์กำหนด	
buf	ตัวชี้รับค่าอ้างอิงไปยังพื้นที่อะเรย์เก็บข้อมูลที่จะส่งและรับกลับ	
count	จำนวนไบต์ข้อมูลที่จะเขียน	
ค่ากลับคืน	ค่าศูนย์หรือมากกว่า	จำนวนไบต์ข้อมูลที่อ่านได้
	PI_BAD_PARAM	ข้อมูลที่ส่งไม่ถูกต้อง
	PI_I2C_READ_FAILED	ไม่สามารถรับข้อมูลจากอุปกรณ์ได้
	PI_BAD_HANDLE	เลขแฮนเดิลไม่ถูกต้อง

ปฏิบัติการ: การติดต่อกับไอซี PCF8575 16bit-GPIO

ไอซี PCF8575 ทำหน้าที่เป็น GPIO ขนาด 16 บิต ซึ่งประกอบไปด้วยเรจิสเตอร์ขนาด 16 บิต ต่อกับพอร์ต 8 บิต 2 พอร์ตด้วยกัน สำหรับการเขียนข้อมูลกระทำได้อย่างง่ายๆ โดยการส่งค่าทั้ง 16 บิตออกไปยังตัวไอซีโดยตรง (ไม่มีไบต์คำสั่ง) แต่ถ้าจะต้องการอ่านค่า จะต้องเซ็ตบิตขาออก ของขาที่ต้องการอ่านให้เป็น 1 เสียก่อน จึงจะสามารถอ่านค่าได้ (โดยถ้าหากอินพุตปล่อยลอย หรือต่อ VCC จะได้ลอจิก 1 แต่ถ้าต่อลงกราวนด์จะได้ลอจิก 0)

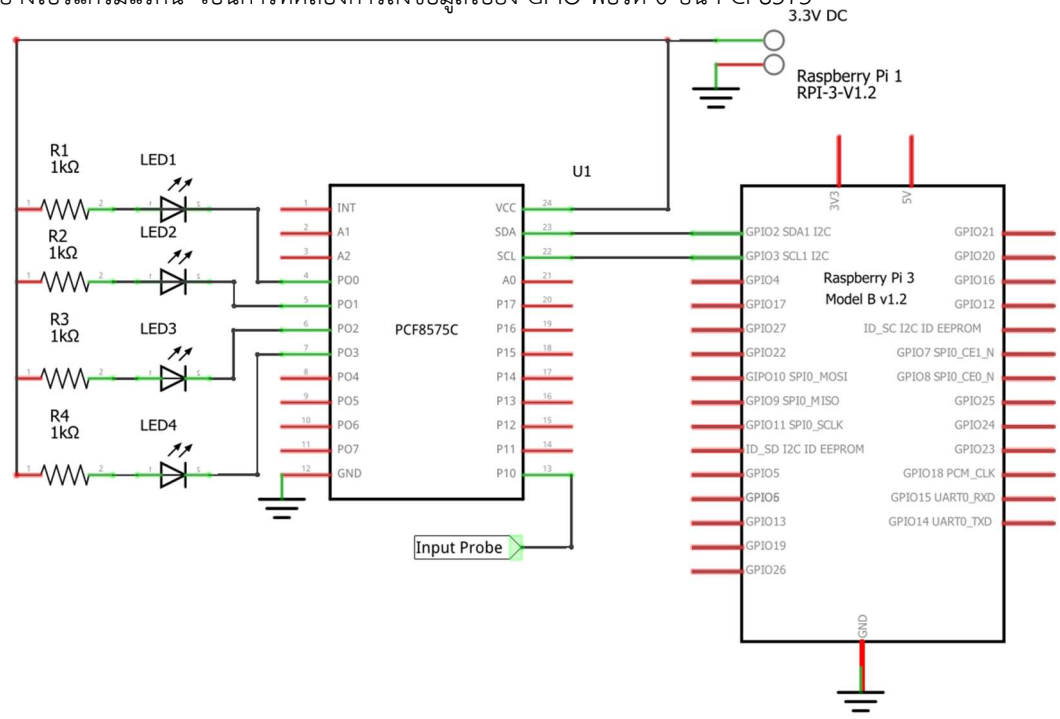


โหมดส่งค่าออก ไบต์ข้อมูลแรกที่ตามหลังไบต์แอดเดรสคือไบต์ที่ส่งไปยังพอร์ต 0 จากนั้นจึงตามด้วยพอร์ต 1



โหมดรับค่าเข้า ไบต์ข้อมูลแรกที่ตามหลังไบต์แอดเดรสคือไบต์ที่รับมาจากพอร์ต 0 แล้วตามด้วยพอร์ต 1

ตัวอย่างโปรแกรมแรกนี้ เป็นการทดลองการส่งข้อมูลไปยัง GPIO พอร์ต 0 บน PCF8575



อุปกรณ์ที่ต้องการ

- บอร์ด Raspberry Pi พร้อมอแดปเตอร์
- บอร์ด PCF8575
- โปรโตบอร์ด และสายจัมพ์อีกตามต้องการ
- LED 4 ดวง และ R 1kohm 4 ตัว

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <pigpio.h>

int running=true;
void gpio_stop(int sig);

int main(){
    int fd,i;

    if(gpioInitialise() < 0) exit(-1);
    if((fd= i2cOpen(1,0x20,0)) < 0) exit(-1);
    signal(SIGINT,gpio_stop);

    printf("I2C  PCF8575 16bit GPIO testing...\n");

    while(running){
        for(i=0;i<16;i++){
            i2cWriteByteData(fd,i,0xff);
            usleep(100000);

        }
        printf("."); fflush(stdout);
    }
    i2cClose(fd);
    gpioTerminate();
    return 0;
}

void gpio_stop(int sig){
    printf("Exiting..., please wait\n");
    running = false;
}
```

จากโปรแกรมตัวอย่าง เราจะพบว่าได้มีการใช้ฟังก์ชัน i2cWriteByteData() มาประยุกต์ใช้เพื่อส่งข้อมูล ทั้งนี้เนื่องจาก pigpio ไม่มีฟังก์ชันที่ใช้ส่งข้อมูลสองไบต์แบบที่ไม่มีคอมมานด์ไบต์หรือไบต์คำสั่ง แต่เนื่องจากเราทราบว่าไบต์คำสั่งจะเป็นไบต์แรกที่ถูกส่งไปตามหลังไบต์แสดงอุปกรณ์ที่จะรับสั่งการ ดังนั้นเราจึงส่งค่าสำหรับพอร์ต 0 ผ่านทางอาร์กิวเมนต์ที่ปกติจะใช้เป็นไบต์คำสั่ง แล้วจึงส่งค่าพอร์ต 1 ผ่านทางอาร์กิวเมนต์ที่ปกติจะใช้ส่งข้อมูล (ในตัวอย่างเราส่ง 0xff ซึ่งไม่ได้มีผลเพราะพอร์ต 1 เราไม่ได้ต่ออุปกรณ์ใดๆ ใช้งาน)

โปรแกรมถัดไป เป็นการทดลองการอ่านค่ามาจาก GPIO เพื่อนำไปทำงานต่อ โดยในที่นี้เราอ่านข้อมูลของทั้งสองพอร์ตออกมา แล้วดึงเอาค่าจากพอร์ต 0 มาใช้งาน จากโปรแกรมจะเห็นว่าเราเลื่อนบิตไปทางขวา 8 บิต หรือในอีกนัยหนึ่งคือ เรานำเอาค่าไบต์บนมาใช้งาน เหตุที่กระทำเช่นนี้เนื่องจากข้อมูลที่ส่งกลับมาจะมีค่าพอร์ต 0 นำมาก่อนพอร์ต 1 แต่เวลา i2cReadWordData() ประกอบค่าส่งกลับคืนนั้น จะพิจารณาไบต์ที่ส่งก่อนเป็นไบต์ต่ำ ทำให้ค่าจากพอร์ต 1 กลายไปเป็นไบต์สูง ค่าจากพอร์ต 1 นี้เรานำมา Bitwise AND กับค่าที่เราจะนำไปแสดงผล เนื่องจากการทำงานของไอซี PCF8575 เวลาที่ไม่ต่อสายเข้าขาอินพุต จะได้ลอจิก 1 แต่เมื่อต่อลงกราวนด์จะได้ลอจิก 0 ดังนั้น การ Bitwise AND กับข้อมูลใดๆ จะทำให้บิตข้อมูลที่ตรงกับขา P10 นั้นจะกลายเป็นศูนย์เสมอ เมื่อเราต่อ P10 ลงกราวนด์ โดยจะเห็นได้จาก LED ดวงที่ต่อกับขา P00 จะติดตลอดเวลาในขณะที่เราต่อ P10 ลงกราวนด์ (ทั้งนี้เนื่องจากเราต่อ LED ในลักษณะที่เราจ่ายไฟเลี้ยงให้กับ LED ผ่าน R เพื่อลดกระแส และต่อเข้าขาอินพุตของไอซี PCF8575 การสั่งให้ดึงสัญญาณลง 0 จึงเป็นผลทำให้ LED สว่าง ซึ่งตรงกันข้ามกับการต่อวงจร LED ที่นักศึกษาได้เคยทดลองมาในปฏิบัติการก่อนหน้านี้)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <pigpio.h>

int running=true;
void gpio_stop(int sig);

int main(){
    int fd,i;
    uint16_t j;

    if(gpioInitialise() < 0) exit(-1);
    if((fd= i2cOpen(1,0x20,0)) < 0) exit(-1);
    signal(SIGINT,gpio_stop);

    printf("I2C  PCF8575 16bit GPIO In/Out testing...\n");
    while(running){
        j=i2cReadWordData(fd,0xff);
        printf("%.2X\n",j);
        j>>=8;
        for(i=0;i<16;i++){
            i2cWriteByteData(fd,i&j,0xff);
            usleep(100000);
        }
        printf("."); fflush(stdout);
    }
    i2cClose(fd);
    gpioTerminate();
    return 0;
}

void gpio_stop(int sig){
    printf("Exiting..., please wait\n");
    running = false;
}
```

จากโปรแกรมตัวอย่างข้างบน ให้สังเกตว่าเรามีการส่งค่า 0xff ผ่านไปทางอาร์กิวเมนต์ที่ใช้ส่งข้อมูล ซึ่งในทางปฏิบัติ ค่านี้จะถูกไปเซ็ตให้กับพอร์ต 1 ซึ่งเรานำมาใช้เป็นพอร์ตอินพุต การเซ็ตค่านี้มีความจำเป็น เนื่องจากไอซี PCF8575 นี้กำหนดให้ผู้ใช้ต้องส่งค่าลอจิก 1 ให้กับบิตของพอร์ตที่จะใช้รับข้อมูลด้วย (ไม่เช่นนั้นจะอ่านแต่ได้ลอจิก 0 ตลอดเวลา)

โปรแกรมต่อไปนี้ เป็นการทดลองการติดต่อ i2c โดยใช้กลไกพื้นฐานของลินุกซ์ โปรแกรมนี้จะส่งผลการทำงานเช่นเดียวกับโปรแกรมที่แล้ว โดยเราจัดการเรื่องชุดแพ็กเก็ตข้อมูลทั้งหมดด้วยตนเอง สำหรับการใช้กับไอซี PCF8575 ซึ่งไม่มีไบต์คำสั่ง การรับส่งข้อมูลก็จะกระทำในรูปแบบง่ายๆ คือส่งข้อมูลสองไบต์ หรืออ่านข้อมูลสองไบต์แบบตรงๆ และสังเกตการณ์ส่งค่า 0xff ให้กับพอร์ต 1 ผ่านทาง i2cwrite() ตัวแรกในวนรอบ while

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>
#include <fcntl.h>

int i2cInit(int busID);
void i2cError(int rw,int size);
void i2cWrite(int fd,const void *data,int size);
void i2cRead(int fd,void *data,int size);

int main(){
    int fd,i;
      uint8_t data[2];
      uint8_t flag;

      if((fd=i2cInit(0x20))==-1) exit(-1);
```

```
    printf("I2C  PCF8575 16bit GPIO In/Out (using linux functions) testing...\n");
    while(1){
       data[0]=data[1]=0xff;
       i2cWrite(fd,data,2);
        i2cRead(fd,data,2);
        flag = data[1];

        printf("%.2X%.2X\n",data[0],data[1]);
        for(i=0;i<16;i++){
                    data[0]=i&flag;
                    i2cWrite(fd,data,2);
            usleep(100000);
        }
        printf("."); fflush(stdout);
    }
    return 0;
}

int i2cInit(int busID){
    char devname[] = "/dev/i2c-1"; //Raspberry pi 2/3
    int fd;

    if((fd=open(devname,O_RDWR))<0){
        fprintf(stderr,"Fail to connect to the i2c device.\n");
        exit(-1);
    }
    if(ioctl(fd, I2C_SLAVE, busID)<0){
        fprintf(stderr,"Unable to communicate to the i2c device.\n");
        exit(-1);
    }
    return fd;
}

void i2cError(int rw,int size){
    if(rw){
        fprintf(stderr,"Error reading %d byte%s from i2c.\n",size,(size>1)?"s":"");
        exit(-1);
    }
    fprintf(stderr,"Error writing %d byte%s to i2c.\n",size,(size>1)?"s":"");
    exit(-1);
}

void i2cWrite(int fd,const void *data,int size){
    if((write(fd,data,size))!=size)
        i2cError(0,size);
}

void i2cRead(int fd,void *data,int size){
    if((read(fd,data,size))!=size)
        i2cError(1,size);
}
```

จากโปรแกรมตัวอย่างข้างบน เราจะเห็นว่าลินุกซ์มองบัสอุปกรณ์ i2c เป็นไฟล์ภายใต้มาตรฐาน POSIX โดยตัวบัส i2c ช่อง 1 มองเห็นเป็น /dev/i2c-1 เมื่อเราเปิดใช้งานไฟล์ก็จะได้ไฟล์แฮนเดิลมาใช้งาน (ในที่นี้เก็บไว้ในตัวแปร fd) จากนั้นเราใช้ฟังก์ชัน ioctl เพื่อสั่งการขอติดต่ออุปกรณ์ตามหมายเลขที่ต้องการ (ในที่นี้คือ 0x20 ซึ่งเป็นหมายเลขอุปกรณ์ที่เราใช้งานในตัวอย่างก่อนหน้าและตัวอย่างนี้) จากนั้นเป็นการเขียนหรืออ่านข้อมูลตามกลไกปกติของการจัดการข้อมูลแบบสตรีมนั่นเอง

ปฏิบัติการ: การติดต่อกับไอซี ADS1115 4-CH ADC convertor และ 2Y0A21 Proximity sensor

2Y0A21 และ 2Y0D21 เป็นอุปกรณ์วัดระยะด้วยอินฟราเรด โดยตัวที่เลือกมาใช้ในการทดลองนี้ใช้รุ่น 2Y0A21 ซึ่งส่งสัญญาณอแนล็อกออกมา โดยมีค่าตั้งแต่ 0v ถึง VCC ตามระยะทางที่วัดจากช่วง 80 เซนติเมตร ถึง 10 เซ็นติเมตร ทั้งนี้เรานำเอาสัญญาณดังกล่าวมาใช้ทดลองป้อนให้ไอซี ADS1115 ซึ่งเป็นไอซีแปลงสัญญาณจากอแนล็อกเป็นดิจิทัลขนาด 16 บิต

ไอซี ADS1115 เป็นไอซีที่ประกอบไปด้วยวงจรแปลงสัญญาณอแนล็อกเป็นดิจิทัลจำนวน 4 ช่องสัญญาณ และภายในมีเรจิสเตอร์จำนวน 4 ตัว ดังรูป

BIT 1	BIT 0	REGISTER
0	0	Conversion register
0	1	Config register
1	0	Lo_thresh register
1	1	Hi_thresh register

เรจิสเตอร์ทั้ง 4 ตัวใน ADS1115

การสั่งการอาศัยการส่งคอมมานด์ไบต์กำหนดตัวเรจิสเตอร์ จากนั้นตามด้วยข้อมูลขนาด 16 บิต สำหรับการเขียนก็จะทำในลักษณะคล้ายคลึงกัน กล่าวคือเมื่อส่งคอมมานต์ไบต์ไปแล้ว ก็จะสามารถอ่านค่ากลับคืนมาได้

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NAME	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Conversion register มีขนาด 16 บิตแบบมีเครื่องหมาย สำหรับใช้อ่านค่าระดับสัญญาณที่แปลงเข้ามาได้

BIT	15	14	13	12	11	10	9	8
NAME	OS	MUX2	MUX1	MUX0	PGA2	PGA1	PGA0	MODE
BIT	7	6	5	4	3	2	1	0
NAME	DR2	DR1	DR0	COMP_MODE	COMP_POL	COMP_LAT	COMP_QUE1	COMP_QUE0

Default = 8583h.

Config register มีขนาด 16 บิต ใช้สำหรับกำหนดโหมดการทำงานของไอซี มีรายละเอียดดังนี้

[bit 15] OS สั่งเริ่มการทำงานในโหมด single conversion โดยส่งค่าบิต 1 ไปเพื่อเริ่มการทำงาน

ค่ากลับคืนที่ได้ 0 แสดงว่าชิพกำลังคำนวณหาค่าคำตอบจากสัญญาณขาเข้า

1 ชิพหาค่าเสร็จเรียบร้อย สามารถอ่านค่าคำตอบได้

[bit 14-12] MUX กำหนดสัญญาณขาอ้างอิง (เพื่อใช้เปรียบเทียบความแตกต่างสัญญาณระหว่างขาสัญญาณกับขาสัญญาณอ้างอิง มีค่าต่างๆ ได้ดังนี้

000 AIN0 = ขาสัญญาณ AIN1 = ขาอ้างอิง

001 AIN0 = ขาสัญญาณ AIN3 = ขาอ้างอิง

010 AIN1 = ขาสัญญาณ AIN3 = ขาอ้างอิง

011 AIN2 = ขาสัญญาณ AIN3 = ขาอ้างอิง

100 AIN0 = ขาสัญญาณ GND = ขาอ้างอิง

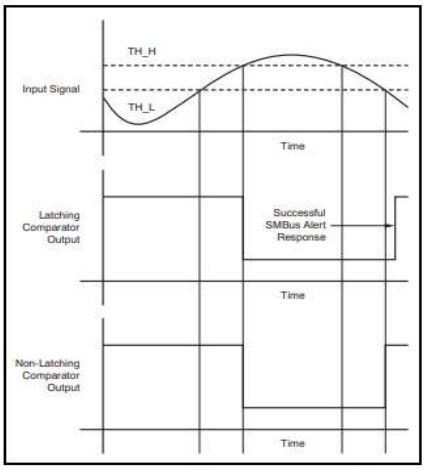
101 AIN1 = ขาสัญญาณ GND = ขาอ้างอิง

110 AIN2 = ขาสัญญาณ GND = ขาอ้างอิง

111 AIN3 = ขาสัญญาณ GND = ขาอ้างอิง

[bit 11-9] PGA ค่าอัตราขยายสัญญาณ (ใช้กำหนดย่านของการวัดของสัญญาณขาเข้า

000	±6.144v	100	±0.512v
001	±4.096v	101	±0.256v
010	±2.048v	110	±0.256v
011	±1.024v	111	±0.256v



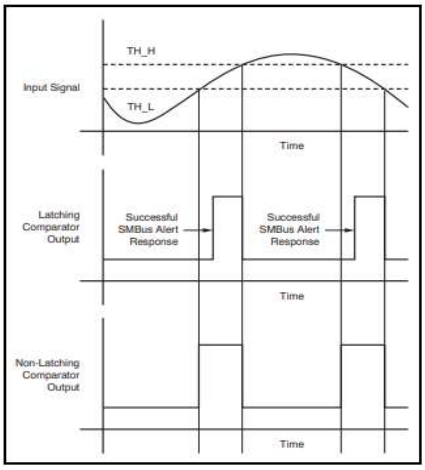
Alert Pin Timing Diagram When Configured as a Traditional Comparator

[bit 8] MODE 0 โหมดการวัดสัญญาณตลอดเวลา

1 โหมดการวัดสัญญาณแบบสั่งการครั้งเดียว (single conversion)

[bit 7-5] data rate กำหนดความเร็วในการทำงานของวงจรวัดสัญญาณ (ยิ่งค่าสูง จะยิ่งมีความคลาดเคลื่อนมากขึ้น)

000	8SPS	100	128SPS
001	16SPS	101	250SPS
010	32SPS	110	475SPS
011	64SPS	111	860SPS



Alert Pin Timing Diagram When Configured as a Window Comparator

[bit 4] COMP_MODE โหมดการทำงานของวงจรแปลงค่าเป็นค่าดิจิทัล

0 โหมดปกติ

1 window comparator

ใช้ร่วมกับการกำหนดย่านสัญญาณ (threashold) เพื่อตรวจสอบหากค่าสัญญาณอะแนล็อกอยู่นอกย่านที่กำหนด โดยจะส่งสัญญาณให้กับขา ALRT (Alert/Ready) เพื่อบอกว่าสัญญาณมีลักษณะอย่างไร

ในโหมดปกติ เมื่อสัญญาณมีค่าเปลี่ยนไปจนขึ้นเกินกว่าค่าสูงสุดที่กำหนด ขา ALRT จะเปลี่ยนสถานะเป็น 0 และเมื่อสัญญาณตกคืนกลับมาต่ำกว่าค่าต่ำสุดทีกำหนด ขา ALRT จะเปลี่ยนสถานะเป็น 1

ในโหมด window comparator สัญญาณ ALRT จะมีสถานะเป็น 1 หากค่าที่วัดอยู่ในช่วงที่กำหนด และเป็น 0 หากอยู่นอกย่าน

[bit 3] COMP_POL โพลาริตี้ของขา ALRT

0 ปกติเป็นศูนย์ (ค่าเริ่มต้น)

1 กลับสถานะ ของขา ALRT

[bit 2] COMP_LAT การกำหนดให้คงค่าสัญญาณของ ALRT

0 ปกติเป็นศูนย์ (ค่าเริ่มต้น) ไม่มีการคงค่า

1 คงสถานะของสัญญาณจนกว่าจะมีการสั่งเคลียรโดยการอ่านค่าจาก conversion register

[bit 1-0] COMP_QUEUE นับจำนวนครั้งของการเปลี่ยนแปลงออกนอกย่านที่กำหนด โดยค่า 0 ถึง 2 แทนการนับตั้งแต่ 1 2 หรือ 4 ครั้ง ส่วนค่า 3 เป็นการยกเลิกการทำงาน comparator

REGISTER	Lo_thresh (Read/Write)							
BIT	15	14	13	12	11	10	9	8
NAME	Lo_thresh15	Lo_thresh14	Lo_thresh13	Lo_thresh12	Lo_thresh11	Lo_thresh10	Lo_thresh9	Lo_thresh8
BIT	7	6	5	4	3	2	1	0
NAME	Lo_thresh7	Lo_thresh6	Lo_thresh5	Lo_thresh4	Lo_thresh3	Lo_thresh2	Lo_thresh1	Lo_thresh0

REGISTER	Hi_thresh (Read/Write)							
BIT	15	14	13	12	11	10	9	8
NAME	Hi_thresh15	Hi_thresh14	Hi_thresh13	Hi_thresh12	Hi_thresh11	Hi_thresh10	Hi_thresh9	Hi_thresh8
BIT	7	6	5	4	3	2	1	0
NAME	Hi_thresh7	Hi_thresh6	Hi_thresh5	Hi_thresh4	Hi_thresh3	Hi_thresh2	Hi_thresh1	Hi_thresh0

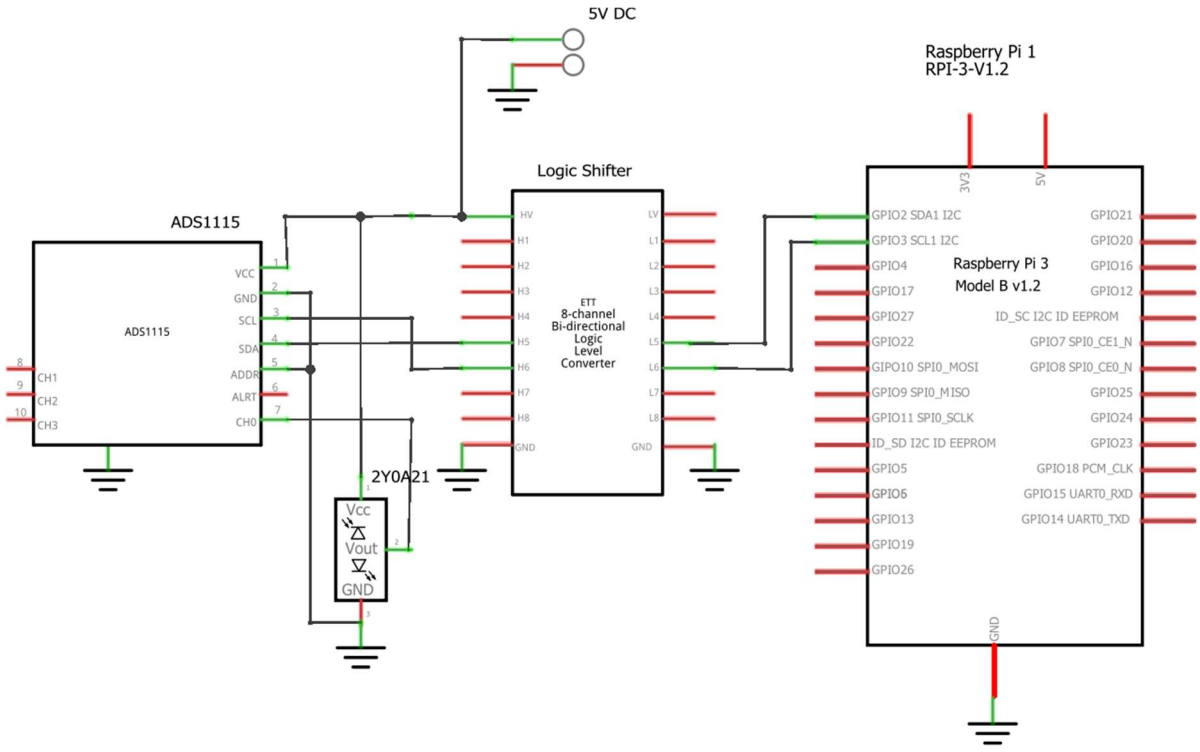
Lo_thresh default = 8000h.

Hi_thresh default = 7FFFh.

Threashhold registers ทั้งสองตัวมีขนาด 16 บิต และรับค่าแบบมีเครื่องหมาย เพื่อกำหนดขอบเขตล่างและขอบเขตบนของสัญญาณที่จะใช้ในการทำงานในโหมด comparator (ในปฏิบัติการนี้เราจะไม่ได้ใช้เรจิสเตอร์ทั้งสองตัวนี้)

<u>อุปกรณ์ที่ต้องการ</u>

- บอร์ด Raspberry Pi พร้อมอแดปเตอร์
- บอร์ด ADS1115
- บอร์ด Logic Shifter
- ตัววัดระยะด้วยอินฟราเรด 2Y0A21
- โปรโตบอร์ด และสายจัมพ์อีกตามต้องการ
- วงจรไฟเลี้ยง 5 V



ตัวอย่างแรกนี้ เป็นการทดลองการใช้ฟังก์ชันของ pigpio ในการอ่านค่าจาก ADS1115 ให้สังเกตว่า ในการรับส่งข้อมูลขนาด 16 บิตไปมาระหว่าง raspberry pi กับ ADS1115 นั้นจะส่งค่าไบต์สูงไปก่อนไบต์ต่ำ (ซึ่งเป็นการส่งแบบ endianness ตรงกันข้ามกับค่าที่เราต้องการ ดังนั้น ค่าที่เรารับและส่งภายในโปรแกรมตัวอย่าง จะมีการสลับไบต์ต่ำกับไบต์สูงกันในทุกจุด)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <pigpio.h>

int running=true;
void gpio_stop(int sig);

int main(){
    int fd;
       uint16_t data;

    if(gpioInitialise() < 0) exit(-1);
    if((fd= i2cOpen(1,0x48,0)) < 0) exit(-1);
    signal(SIGINT,gpio_stop);

    printf("I2C  ADS1115 4ch ADC testing...\n");

    while(running){
       i2cWriteWordData(fd,1,0x03c3);
       while(((data=i2cReadWordData(fd,1))&0x80)==0){
        usleep(100000);
       }
        data = i2cReadWordData(fd,0);
        //reverse LO/HI byte
        printf("Vout = %.4X\n",((data>>8)&0xff)|((data<<8)&0xff00));
        fflush(stdout);
        usleep(250000);
    }
    i2cClose(fd);
    gpioTerminate();
    return 0;
}

void gpio_stop(int sig){
    printf("Exiting..., please wait\n");
    running = false;
}
```

คราวนี้เป็นการใช้กลไกมาตรฐานของลินุกซ์ สังเกตว่าเนื่องจากกลไกของลินุกซ์ไม่มีการแยกแยะไบต์คำสั่ง เราจึงต้องเพิ่มไบต์คำสั่งไปเป็นไบต์แรกของชุดข้อมูลที่จะจัดส่ง

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>
#include <fcntl.h>

int i2cInit(int busID);
void i2cError(int rw,int size);
void i2cWrite(int fd,const void *data,int size);
void i2cRead(int fd,void *data,int size);

int main(){
    int fd;
    unsigned char data[4];

    if((fd=i2cInit(0x48))==-1){
        return -1;
```

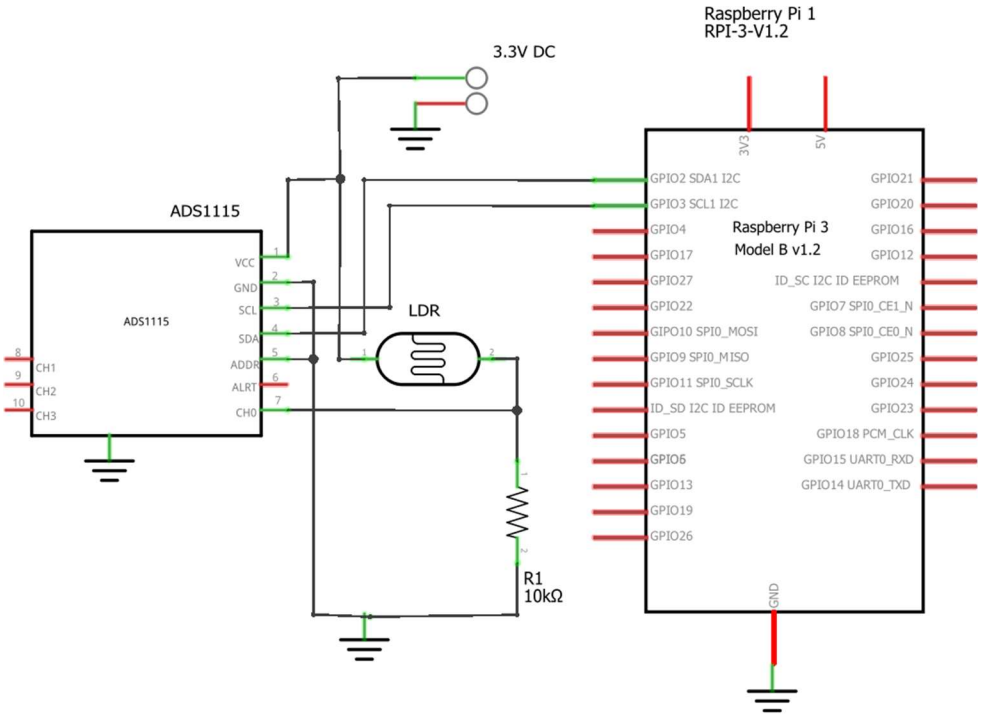
```
    }
    printf("I2C ADS1115 4-ch ADCtesting (Using Linux I/O)...\n");

    while(1){
                    //Start single conversion
                    data[0]=1;
                    data[1]=0xc3;
                    data[2]=0x03;
            i2cWrite(fd,data,3); //Write config register
                    data[0] = 0;
                    while((data[0] & 0x80) == 0){
                           i2cRead(fd,data,2); //Check OS(ready) bit
//        printf("CTRL = %.4X\n",data);
              usleep(100000);
             }
                    //Read result
                    data[0] = 0; // Read from conversion register
                    i2cWrite(fd,data,1);
                    i2cRead(fd,data,2); // Read two bytes
            printf("Vout = %.2X%.2X\r",data[0],data[1]);
            fflush(stdout);
            usleep(250000);
    }

    return 0;
}

int i2cInit(int busID){
    char devname[] = "/dev/i2c-1"; //Raspberry pi 2/3
    int fd;

    if((fd=open(devname,O_RDWR))<0){
        fprintf(stderr,"Fail to connect to the i2c device.\n");
        exit(-1);
    }
    if(ioctl(fd, I2C_SLAVE, busID)<0){
        fprintf(stderr,"Unable to communicate to the i2c device.\n");
        exit(-1);
    }
    return fd;
}

void i2cError(int rw,int size){
    if(rw){
        fprintf(stderr,"Error reading %d byte%s from i2c.\n",size,(size>1)?"s":"");
        exit(-1);
    }
    fprintf(stderr,"Error writing %d byte%s to i2c.\n",size,(size>1)?"s":"");
    exit(-1);
}

void i2cWrite(int fd,const void *data,int size){
    if((write(fd,data,size))!=size)
        i2cError(0,size);
}

void i2cRead(int fd,void *data,int size){
    if((read(fd,data,size))!=size)
        i2cError(1,size);
}
```

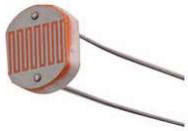
จากโปรแกรมตัวอย่างทั้งสอง ค่าที่อ่านได้มีลักษณะ Endianness ตรงกันข้ามกับที่เราต้องการ ดังนั้นจึงต้องมีการสลับไบต์สูง-ต่ำก่อนการรับส่งข้อมูล โดยโปรแกรมตัวอย่างที่ใช้ฟังก์ชันของ pigpio เราสลับไบต์บนและไบต์ล่างซึ่งกันและกันในระหว่างการแสดงผล ส่วนโปรแกรมตัวอย่างที่ใช้ฟังก์ชันของลินุกซ์ เนื่องจากค่าที่อ่านได้มานั้นถูกนำมาเก็บไว้ในอะเรย์แบบชนิด unsigned char เราจึงแสดงผลโดยใช้ข้อมูลจากตำแหน่งอินเด็กซ์ 0 นำหน้าตำแหน่ง 1

ปฏิบัติการ: การติดต่อกับไอซี ADS1115 4-CH ADC convertor และ LDR

เนื่องจากตัวไอซี ADS1115 เป็นไอซีแปลงสัญญาณจากอะแนล็อกเป็นดิจิทัล ดังนั้นเราสามารถนำไปประยุกต์ใช้งานกับอุปกรณ์ใดๆ ที่ให้สัญญาณเอาต์พุตออกมาในย่านระหว่างช่วง VCC - GND ของไอซี (ถ้าสัญญาณไม่ได้ในย่านนี้ เราก็สามารถใช้วงจรคูณสัญญาณ หรือวงจร voltage divider เพื่อให้ได้สัญญาณในย่านดังกล่าว

ตัวอย่างต่อไปนี้เป็นการใช้ LDR หรือที่เรียกว่า photoresistor ซึ่งค่าความต้านทานเปลี่ยนไปตามแสงที่ตกกระทบ ทั้งนี้ LDR แต่ละรุ่นจะมีค่าความต้านทานที่เปลี่ยนไปเมื่อแสงตกกระทบที่แตกต่างกันไป แต่โดยทั่วไปแล้ว เมื่อไม่มีแสงตกกระทบ จะมีค่าความต้านทานในระดับเมกะโอห์ม แต่เมื่อมีแสงตกกระทบจะอยู่ในช่วงหลักกิโลโอห์ม (ตัวที่นักศึกษาจะได้รับในการทำปฏิบัติการจะอยู่ในช่วงประมาณ 10กิโลโอห์ม) วงจรที่เราจะนำมาใช้นี้ เราต่อ LDR กับ R ขนาด 10kohm เพื่อสร้างเป็นวงจร voltage divider และต่อสายออกมาเพื่อป้อนให้กับ ADS1115 เพื่ออ่านค่าต่อไป





อุปกรณ์ที่ต้องการ

- บอร์ด Raspberry Pi พร้อมอแดปเตอร์
- บอร์ด ADS1115
- LDR
- R ขนาด 10kohm
- โปรโตบอร์ด และสายจัมพ์อีกตามต้องการ
- วงจรไฟเลี้ยง 5 V

สำหรับโปรแกรมตัวอย่างนั้น ให้นักศึกษาใช้โปรแกรมตัวอย่างจากปฏิบัติการก่อนหน้าได้โดยไม่ต้องแก้ไขใดๆ ให้สังเกตว่า ในขณะที่โปรแกรมกำลังทำงานอยู่นั้น หากมีแสงสว่างตกที่ LDR จะมีค่าที่อ่านได้อยู่ค่าประมาณหนึ่ง แต่ถ้าเรานำแสงออกไป (หรือใช้มือบัง LDR) ก็จะพบว่าค่าที่อ่านได้นั้นลดลง ทั้งนี้เนื่องจาก LDR จะมีความต้านทานมากขึ้นเมื่อแสงตกกระทบลดน้อยลง ทำให้อัตราส่วน R1/(R1+R2) ของวงจร divider ที่เราสร้างขึ้นด้วย LDR กับ R ขนาด 10 kohm นั้นมีค่าที่เพิ่มขึ้น (ตัวหารมีค่ามากขึ้น)

ปฏิบัติการ: การทำสแกนคีย์บอร์ด และรับค่าระยะทางจากตัววัดอินฟราเรด หรือจาก LDR

อุปกรณ์ที่ต้องการ

- บอร์ด Raspberry Pi พร้อมอแดปเตอร์
- สายจัมพ์จากขา GPIO ของบอร์ด
- โปรโตบอร์ด และสายจัมพ์อีกตามต้องการ
- วงจร Logic Shifter ขนาด 4 channel หรือเทียบเท่า
- บอร์ด ADS7828
- บอร์ด Logic Shifter
- ตัววัดระยะด้วยอินฟราเรด 2Y0A21 หรือ LDR และ R ขนาด 10kohm
- บอร์ด PCA9535
- วงจรไฟเลี้ยง 5v DC
- คีย์เมตริกซ์ขนาด 4x3

ให้นักศึกษาปรับโปรแกรมที่ได้จากปฏิบัติการเดิม ที่มีการใช้พอร์ต GPIO ต่อกับคีย์บอร์ดเมตริกซ์และทำการแสกนคีย์บอร์ด และที่มีการใช้อัลตราโซนิกในการวัดระยะทาง ให้หันมาต่อคีย์บอร์ดผ่านพอร์ต GPIO ของ PCF8575 และหันมาใช้วงจรวัดระยะทางที่ประกอบไปด้วย ADS1115 และ 2Y0A21 หรือใช้ ADS1115 ร่วมกับ LDR และดูผลว่าสามารถทำงานได้ตามที่คาดหวังหรือไม่