หุ่นยนต์เคลื่อนที่

นายกุลศักดิ์ ช่วยชู

นายวัลลภ อินทรสังขนาวิน

โครงงานพิเศษนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิทยาศาตรบัณฑิต

ภาควิชาฟิสิกส์ประยุกต์

คณะวิทยาศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2542

Mobile Robot

Mr.Kunlasuk Chuaichoo
Mr.Wanlop Intarasangkanawin

A Special Project Submitted in Partial Fulfillment of the
Requirement for the Degree of Bachelor of Science
Department of Applied Physics
Faculty of Science
King Mongkut 's Institute of Technology Ladkrabang
1999

หัวข้อโครงงานพิเศษ	หุ่นยนต์เคลื่อนที่
โดย	นายกุลศักดิ์ ช่วยชู
	นายวัลลภ อินทรสังขนาวิน
ภาควิชา	ฟิสิกส์ประยุกต์
อาจารย์ที่ปรึกษา	ผู้ช่วยศาสตราจารย์วิชิต ศิริโชติ
	รองศาสตราจารย์สุรพล รักวิจัย

ภาควิชาฟิสิกส์ประยุกต์ คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง อนุมัติให้นับโครงงานพิเศษฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิทยาศาสตรบัณฑิต

..... หัวหน้าภาควิชาฟิสิกส์ประยุกต์
(รองศาสตราจารย์สุรพล รักวิจัย)

คณะกรรมการโครงงานพิเศษ

..... ประธานกรรมการ
(ผู้ช่วยศาสตราจารย์วิชิต ศิริโชติ)

..... กรรมการ
(รองศาสตราจารย์สุรพล รักวิจัย)

..... กรรมการ
(ผู้ช่วยศาสตราจารย์ศิริศักดิ์ เตชะทวีกุล)

..... กรรมการ
(อาจารย์ภูมินทร์ จินดาจิธาวัฒน์)

หัวข้อโครงงานพิเศษ	หุ่นยนต์เคลื่อนที่	
โดย	นายกุลศักดิ์	ช่วยชู
	นายวัลลภ	อินทรสังขนาวิน
อาจารย์ที่ปรึกษา	ผู้ช่วยศาสตราจารย์วิชิต	ศิริโชติ
	รองศาสตราจารย์สุรพล	รักวิจัย
ภาควิชา	ฟิสิกส์ประยุกต์	
ปีการศึกษา	2542	

บทคัดย่อ

โครงงานพิเศษนี้เป็นการสร้างหุ่นยนต์แบบง่าย เพื่อใช้ศึกษาการเขียนโปรแกรมควบคุมและใช้ไมโครคอนโทรลเลอร์ MC68HC11 โครงสร้างและรูปแบบของหุ่นยนต์เป็นแบบทรงกลมทำจากแผ่นพลาสติกและแผ่นอะลูมิเนียม บอร์ดควบคุมจะประกอบด้วย MC68HC11 และแรม 32 KByte NVSRAM ส่วนของเซนเซอร์ประกอบด้วย โฟโต้รีซิสเตอร์ใช้ในการวิ่งตามเส้น เซนเซอร์อินฟราเรดตรวจจับสิ่งกีดขวาง คอนเดนเซอร์ไมโครโฟนตรวจจับเสียง เซนเซอร์กันชนใช้ไมโครสวิตช์เพื่อหลบวัตถุที่ชน การเขียนโปรแกรมควบคุมหุ่นยนต์เราใช้ Interactive C (IC) ทำการเขียนโปรแกรมแบบมัลติทาส์คกิ้งได้ง่าย ได้นำเสนอผลการทดลองให้หุ่นยนต์วิ่งค้นหาแถบเส้นสีดำ หุ่นยนต์ตรวจจับเสียงและตรวจจับแสงอินฟราเรด

Special Project Title	Mobile Robot	
Name	Mr.Kunlasak	Chuaichoo
	Mr.Wanlop	Intarasangkanawin
Special Project Adviser	Assistant Professor Wichit	Sirichote
	Associate Professor Surapol	Rakvichai
Department	Applied Physics	
Academic	1999	

Abstract

A simple mobile robot has been built for studying robot programming and the use of microcontroller MC68HC11. The robot platform is circular body made with plastic and aluminum sheet. The control board has a Motorola MC68HC11 8-bit CMOS microcontroller with 32kB NVSRAM. The sensors of the robot are photocell for back-tape tracking, an IR module, and condenser microphone. Three bumping sensors using micro switches provide escape function. The programming language is Interactive C, IC. The IC provides easier multitasking programming. Experimenting of finding, tracking back tape, detect Infrared and responding of sound pressure were presented.

กิตติกรรมประกาศ

โครงงานนี้สำเร็จลุล่วงได้ดี เนื่องจากการอนุเคราะห์ของบุคคลหลายฝ่ายดังนี้

ขอขอบพระคุณ

บิดาและมารดา	ผู้ให้ชีวิตและทุกสิ่งทุกอย่าง
ผศ.วิชิต ศิริโชติ	ผู้ให้คำปรึกษาและคำแนะนำเกี่ยวกับโครงงานพิเศษ
อ.ธวัชชัย ขาวประเสริฐ	สำหรับห้อง Shop เพื่อใช้ในการทำโครงงาน
อาจารย์ภูมินทร์	สำหรับคำปรึกษาและอุปกรณ์
พี่เจน	สำหรับคำปรึกษาและอุปกรณ์
คุณวิน, คุณตี๋ และ คุณเดย์	สำหรับความช่วยเหลือทางด้านโปรแกรมและเซนเซอร์
เพื่อนๆทุกคน	ที่ช่วยเป็นกำลังใจและคอยช่วยเหลือตลอดมา

กุลศักดิ์ ช่วยชู

วัลลภา อินทรสังขนาวิน

สารบัญ

สารบัญรูป

หน้า

สารบัญตาราง

บทที่ 1
บทนำ

ในโลกปัจจุบันนี้ได้มีความก้าวหน้าทางเทคโนโลยีอย่างไม่หยุดยั้ง ตลอดจนการนำหุ่นยนต์หรือเครื่องจักรเข้ามาช่วยในการอำนวยความสะดวกสบาย หรือช่วยในการลดการเสี่ยงอันตรายของมนุษย์ เช่น การสร้างหุ่นยนต์เพื่อใช้ในการตรวจสอบและสำรวจดาวเคราะห์ หรือใช้ในการตรวจวัดกัมมันตภาพรังสี ซึ่งพื้นฐานโดยทั่วไปก็คือการควบคุมการเคลื่อนที่ของหุ่นยนต์เพื่อจะได้นำไปใช้ประโยชน์ตามที่ต้องการ จากแนวความคิดนี้จึงได้มีการศึกษาเกี่ยวกับหุ่นยนต์ที่มีการเคลื่อนที่ โดยควบคุมผ่านทางไมโครคอนโทรลเลอร์ ซึ่งในปัจจุบันได้มีความนิยมเพิ่มขึ้นเรื่อยๆในเรื่องของการใช้ไมโครคอมพิวเตอร์ชิปเดี่ยว (Single-chip microcomputer) เพราะวงจรรวมมีขนาดเล็กและมีราคาไม่แพงมากจนเกินไป

1.1 วัตถุประสงค์

1. เพื่อศึกษาการทำงานของไมโครคอนโทรลเลอร์
2. เพื่อศึกษาถึงวงจรควบคุมการทำงานและเซนเซอร์ต่างๆ
3. เพื่อศึกษาเกี่ยวกับโปรแกรมควบคุมการทำงานของหุ่นยนต์
4. สามารถนำที่หุ่นยนต์ที่สร้างขึ้นมาใช้ประโยชน์ได้

1.2 วิธีดำเนินการ

การดำเนินการในการสร้างหุ่นยนต์ที่มีการเคลื่อนที่มีขั้นตอนดังนี้

1. ศึกษาและค้นคว้าข้อมูลเกี่ยวกับวงจร รูปแบบหุ่นยนต์และโปรแกรม
2. ทำการศึกษาเกี่ยวกับเซนเซอร์ต่างๆที่จะนำมาใช้
3. จัดหาอุปกรณ์ในส่วนของการสร้างรูปร่างและวงจรในการควบคุมรวมไปถึงเซนเซอร์
4. ทำการทดลองการทำงานของเซนเซอร์แต่ละอย่าง
5. นำเซนเซอร์ทุกอย่างมาประกอบรวมกันและทำการทดลอง

1.3 ประโยชน์ที่ได้รับ

1. สามารถเข้าใจถึงหลักการทำงานของไมโครคอนโทรลเลอร์
2. มีความรู้เพิ่มขึ้นในส่วนของการศึกษาและการนำเซนเซอร์ต่างๆมาประยุกต์ใช้
3. มีความเข้าใจในส่วนของการเขียนโปรแกรมควบคุมไมโครคอนโทรลเลอร์มากขึ้น

บทที่ 2
ทฤษฎีที่เกี่ยวข้อง

ในการเคลื่อนที่ของหุ่นยนต์มีอยู่ด้วยกันหลายวิธี ซึ่งการเคลื่อนที่ด้วยวิธีล้อหมุนก็เป็นการเคลื่อนที่วิธีหนึ่ง โดยสามารถทำได้ด้วยการเชื่อมต่อเซนเซอร์เข้ากับไมโครคอนโทรลเลอร์และควบคุมการทำงานผ่านทางการประมวลผลคำสั่งของไมโครคอนโทลเลอร์ ซึ่งในการใช้ไมโครคอนโทรลเลอร์จะมีข้อดีในเรื่องของความสามารถรอบตัวในการประยุกต์,ชนิดของพลังงานที่ใช้, ขนาด, และความสะดวกในการใช้ โดยส่วนสำคัญของไมโครคอนโทรลเลอร์ที่เป็นส่วนที่น่าสนใจที่สามารถใช้ในการควบคุม การตัดสินใจของหุ่นยนต์ได้นั้น คือส่วนของเซนเซอร์และโปรแกรม ซึ่งโดยทั่วไปในการจะเปลี่ยนพฤติกรรมของหุ่นยนต์จะใช้วิธีการต่อวงจรเซนเซอร์ตรวจจับทางกายภาพใหม่และเพิ่มอุปกรณ์บางส่วนเข้าไป ซึ่งในส่วนของฮาร์ดแวร์ของหุ่นยนต์ตัวนี้ เลือกใช้ไมโครคอนโทรลเลอร์ MC68HC11 จาก Motorola โดยการเชื่อมต่ออินพุทเอาท์พุทขาของชิปจะเชื่อมต่อกับส่วนของเซนเซอร์และไอซีขับมอเตอร์เพื่อตรวจจับสัญญาณและควบคุมการเคลื่อนที่ ดังนั้นพื้นฐานทางฟิสิกส์ของอุปกรณ์ที่ใช้จำเป็นอย่างยิ่ง ซึ่งมีดังนี้

2.1 มอเตอร์

มอเตอร์ไฟฟ้าจะเปลี่ยนพลังงานไฟฟ้าไปเป็นพลังงานกล โดยจะมีอยู่หลายรูปแบบและหลายขนาด ซึ่งมีทั้งมอเตอร์แม่เหล็กไฟฟ้าแบบกระแสตรง (DC) และแบบกระแสสลับ (AC) โดยมอเตอร์แบบ AC จะใช้สำหรับเครื่องจักรขนาดใหญ่ เช่น เครื่องซักผ้า เครื่องปั่นแห้ง ซึ่งมอเตอร์แบบ AC นี้ไม่ค่อยได้ใช้เกี่ยวกับการเคลื่อนที่ของหุ่นยนต์ เพราะว่าในการเคลื่อนที่ของหุ่นยนต์จะใช้แบตเตอรี่แบบ DC เป็นตัวจ่ายกำลังไฟฟ้าเข้าสู่มอเตอร์

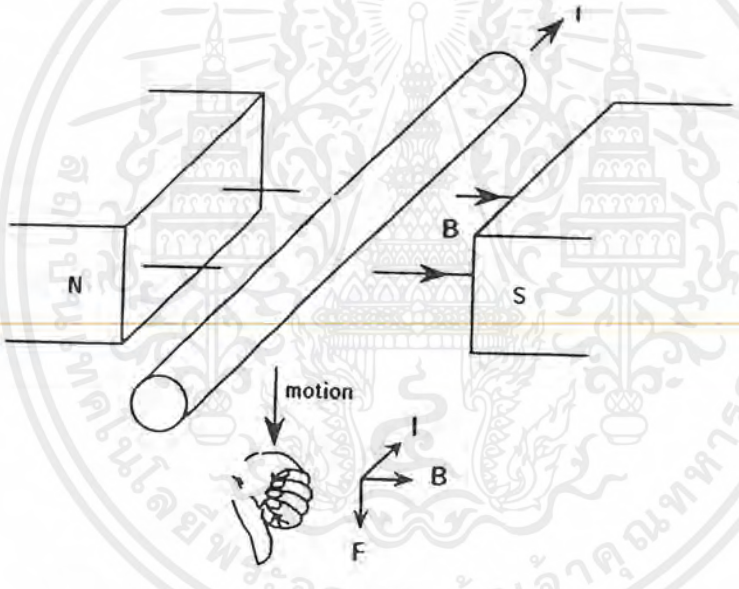
มอเตอร์แบบ DC โดยปกติจะใช้สำหรับงานที่มีขนาดเล็ก ซึ่งจะมีความหลากหลายมากทั้งขนาดและรูปร่าง โดยปกติแล้วมอเตอร์แบบ DC จะมีความเร็วในการทำงานสูงมากและมีค่ากำลังบิด (torque) ต่ำมาก แต่ในการจะเปลี่ยนแปลงคุณลักษณะเหล่านี้ในมอเตอร์แบบ DC สามารถทำได้ด้วยการนำเพลามาต่อเข้ากับที่เฟืองของมอเตอร์ ซึ่งจะทำให้มอเตอร์มีการหมุนช้าลงและมีค่ากำลังบิดมากขึ้น โดยในการต่อเพลาสามารถต่อเข้าไปได้โดยตรงหรืออาจจะใช้มอเตอร์แบบที่มีเพลาต่อเอาไว้ภายในเรียบร้อยแล้วก็ได้

ที่ส่วนปลายของมอเตอร์แบบ DC ส่วนใหญ่จะมีขั้วไฟฟ้า 2 ขั้ว โดยสามารถใส่แรงดันไฟฟ้าเข้าไปตกคร่อมที่ขั้วไฟฟ้า ซึ่งจะทำให้มอเตอร์เกิดการหมุนไปในทิศทางหนึ่ง แต่ถ้ามีการต่อแรงดันแบบสลับขั้วเข้าไปจะทำให้มอเตอร์มีการหมุนไปในทิศทางตรงข้าม โดยในการสลับขั้วของมอเตอร์จะใช้ในการพิจารณาทิศทางของการหมุน ในขณะที่แอมพลิจูดของแรงดันจะพิจารณาจากความเร็วของมอเตอร์

อย่างไรก็ตาม ในมอเตอร์กระแสตรงบางชนิด เช่น มอเตอร์แบบสเตร็บ (Stepper motor) จะมีขั้วไฟฟ้ามากกว่า 2 ขั้ว โดยปกติแล้วจะมี 6 ขั้ว หรือ 8 ขั้ว ซึ่งสัญญาณที่ถูกใส่เข้าไปในเส้นลวดเหล่านี้จะมีพลังงานที่แตกต่างกันออกไปตามขดลวดภายในมอเตอร์และมอเตอร์อีกแบบที่มีขั้วมากกว่า 2 ขั้ว คือ มอเตอร์แบบเซอร์โว (Servo motor)

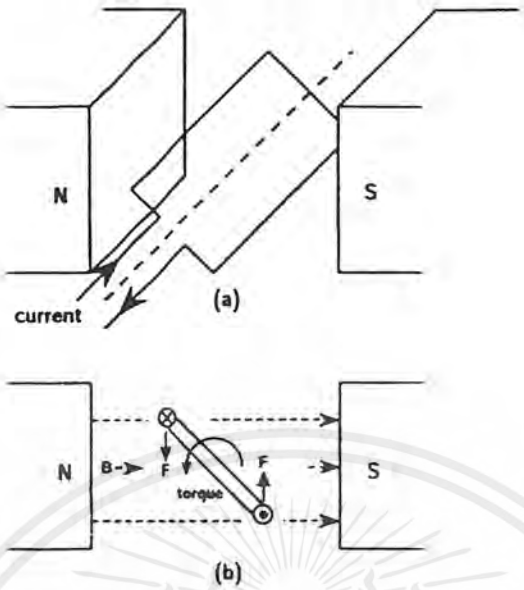
2.1.1 การทำงานของมอเตอร์แบบ DC

ในโครงงานนี้ เราจะสนใจในการทำงานของมอเตอร์ซึ่งหัวเฟืองเป็นแม่เหล็กถาวรแบบ DC (permanent magnet DC gearhead motors) โดยถ้าเข้าใจโครงสร้างกลไกและค่าแรงบิดก็จะช่วยได้มากโดยการอ่านเอกสารประกอบรายละเอียดของมอเตอร์ เพื่อใช้ในการเลือกขนาดของมอเตอร์ที่ถูกต้อง และยังช่วยในการออกแบบการควบคุมมอเตอร์จากไมโครคอนโทรลเลอร์



รูปที่ 2.1 เป็นสนามเส้นแรงแม่เหล็ก B ที่เกิดขึ้นโดยแม่เหล็กถาวรในทิศทางจากขั้วเหนือไปยังขั้วใต้

แรงแม่เหล็กไฟฟ้าในมอเตอร์แบบ DC จะเกิดขึ้นเมื่อมีกระแสไปเหนี่ยวนำที่สนามแม่เหล็ก ดังในรูปที่ 2.1 ซึ่งสนามแม่เหล็กจะเกิดขึ้นโดยแม่เหล็กถาวร ซึ่งจะเกิดเส้นแรงการดูด (flux line) ของสนามแม่เหล็กจากขั้วทางเหนือไปยังขั้วทางใต้ จากกฎแรงของ Lorentz บอกว่ากระแสจะถูกตัวเหนี่ยวนำนั้นเหนี่ยวนำไปในสนามแม่เหล็กเพื่อสร้างแรง ซึ่งแรง F จะตั้งฉากกับทั้งทิศทางของกระแส I และทิศทางของสนามแรงดูด B โดยแรง F จะพิจารณาจากกฎมือขวา ซึ่งนิ้วมือจะงอจากทิศทางของกระแสตรงไปยังทางทิศทางของสนามการดูด และนิ้วหัวแม่มือจะชี้ไปในทิศทางของแรงผลลัพธ์ที่ถูกสร้าง ซึ่งในกรณีของรูปที่ 2.1 จะเกิดแรงในทิศทางลงมาข้างล่าง



รูปที่ 2.2 รูป a เป็นลูปของวงจรที่กระแสวิ่งเข้าไปทางด้านซ้ายและออกมาทางด้านขวา รูป b เป็นผลลัพธ์ของแรงในทิศทางที่ตรงกันข้ามกับระยะทางจากกึ่งกลางของการหมุน ซึ่งเกิดจากการหมุนจนกระทั่งมีทิศลงมาทางแนวดิ่ง

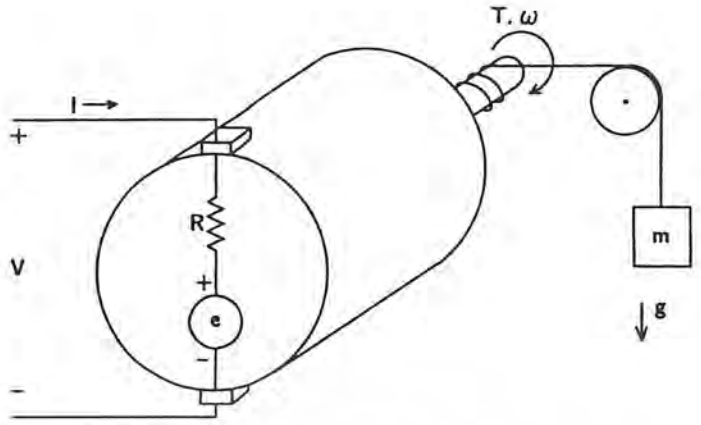
การเคลื่อนที่แบบหมุนรอบต้องการลูปของขดลวด โดยรูปที่ 2.2 (a) แสดงลูปของวงจรบนแกนของการหมุนและตำแหน่งในการติดตั้งสนามแรงดูดด้วยแม่เหล็กถาวร และรูปที่ 2.2 (b) เป็นผลลัพธ์ที่ได้ของแรง เพราะว่าแรงจะถูกสร้างในทิศทางตั้งฉากกับทั้งทิศทางของกระแสและสนามแม่เหล็ก โดยกระแสจะตรงเข้าไปในตามลูปจนถึงจุดยอด ซึ่งจากกฎมือขวาจะทำให้แรงมีทิศทางลงมาข้างล่าง

2.1.2 แรงบิด, ความเร็ว, กำลังงาน, และพลังงาน

แรงบิด (Torque) เป็นแรงเชิงมุมที่มอเตอร์สามารถส่งที่ระยะทางที่แน่นอนจากเพลา ตัวอย่างเช่น แรงบิดมีค่าเท่ากับ 5 ออนซ์-นิ้ว มีความหมายว่า ที่ระยะทาง 1 นิ้วจากเพลาของมอเตอร์ ซึ่งมอเตอร์จะมีแรงพอที่จะสามารถดึงน้ำหนักขนาด 5 ออนซ์หมุนผ่านลอก ดังรูปที่ 2.3 ซึ่งในหน่วยเมตริก แรงบิดของมอเตอร์จะเป็นนิวตัน-เมตร (N-m) โดยในหน่วยเมตริกของแรงบิดจะพบในเทอมของ กรัม-แรง-เซ็นติเมตร (g-f-cm) ที่ซึ่ง แรง-กรัม หมายถึงแรงที่เกิดจากแรงโน้มถ่วงกระทำต่อมวล 1 กรัม โดยในการแปลงหน่วยทำได้ตามสมการนี้

$$1N = 1\frac{kg-m}{\sec^2} = 0.225lb$$

โดย 1 กิโลกรัม = 2.21 ปอนด์ และ 1 นิ้ว = 2.54 เซ็นติเมตร



รูปที่ 2.3 เป็นวงจรเสมือนของมอเตอร์แบบ DC ซึ่งการพันของมอเตอร์เสมือนกับมีค่าความต้านทาน R และตัวสร้างแรงดันซึ่งทำให้กระแสเคลื่อนที่ไปตามตัวนำ e โดยกำลังไฟฟ้าขาเข้าหาได้จาก $P_e = VI$ และกำลังงานกลที่ออกมาเกิดจากแรงบิดและความเร็วการหมุน $P_m = T\omega$

เมื่อพูดเกี่ยวกับการเปลี่ยนกำลังไฟฟ้าเป็นกำลังงานกลในมอเตอร์ ถ้ารู้ความสัมพันธ์เกี่ยวกับกำลังงาน (วัตต์) และพลังงาน (จูล) ก็จะมีประโยชน์ โดยกำลังงานเป็นอัตราส่วนของการใช้พลังงานจนหมด ซึ่งความสัมพันธ์ของกำลังงานและพลังงานเป็นดังนี้

$$1Watt = 1\frac{Joule}{\sec}$$

รูปที่ 2.3 แสดงการเปลี่ยนกำลังไฟฟ้าเป็นกำลังงานกลในมอเตอร์แบบ DC โดยแหล่งจ่ายกำลังไฟฟ้าไปยังมอเตอร์ P_e เท่ากับ แรงดันตกคร่อมมอเตอร์ V คูณกับกระแส I โดยกระแสจะวัดในหน่วยของแอมแปร์ ซึ่งเป็นจำนวนของประจุในหน่วยคูลอมป์ผ่านไปยังหน้าตัดของตัวเหนี่ยวนำต่อวินาที

$$P_e = VI$$

$$1Ampere = 1\frac{Coulomb}{\sec}$$

$$1Watt = 1Volt.Ampere = 1Volt.\frac{Coulomb}{\sec}$$

กำลังงานกล P_m มีค่าเท่ากับแรงบิดที่ออกมาโดยเพลา T คูณกับความเร็วเชิงมุมของตัวมันเอง ω โดยค่าแรงบิดมีหน่วยเป็นนิวตัน-เมตร และความเร็วเชิงมุมจะวัดในหน่วยของเรเดียนต่อวินาที

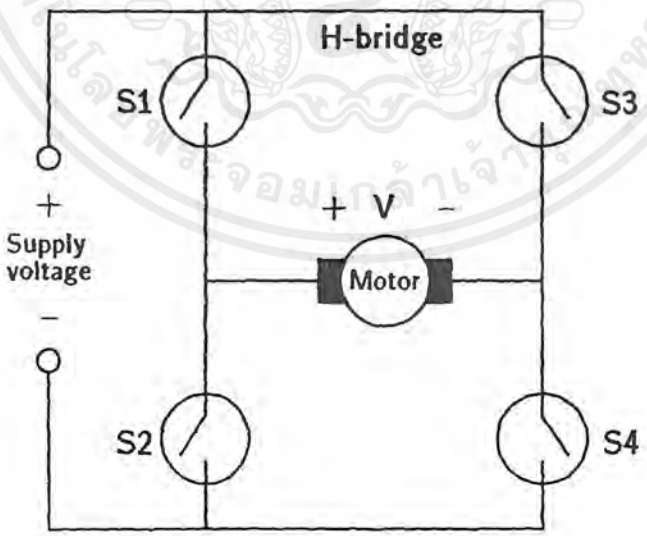
$$P_m = T\omega$$
$$\frac{2\pi rad}{\sec} = 1\frac{rev}{\sec}$$
$$1Watt = 1\frac{Nm}{\sec}$$

กำลังงานเป็นพลังงานต่อ 1 หน่วยเวลา ซึ่งเป็นการบอกว่าพลังงาน 1 จูล สามารถคิดได้เป็น 2 แบบคือ เป็น 1 นิวตัน-เมตร หรือ 1 คูลอมป์-โวลต์

$$1J = 1Nm \quad \text{และ} \quad 1J = 1CV$$

2.1.3 การเชื่อมต่อมอเตอร์

ไมโครคอนโทรลเลอร์ไม่สามารถที่จะขับมอเตอร์ได้โดยตรง เพราะไม่สามารถที่จะให้กระแสได้เพียงพอ จึงต้องมีการมีการต่อวงจรขับ ซึ่งกำลังของมอเตอร์จะถูกจ่ายจากแหล่งกำลังอื่นและมีการควบคุมสัญญาณจากไมโครคอนโทรลเลอร์ ซึ่งวงจรในการเชื่อมต่อไฟฟ้าเป็นอุปกรณ์ที่มีความหลากหลายทางด้านเทคโนโลยี เช่น รีเลย์ (Relay), ทรานซิสเตอร์แบบไบโพลาร์ (bipolar transistor), มอสเฟต (MOSFET) และวงจรตัวขับมอเตอร์ ซึ่งโครงสร้างพื้นฐานของวงจรจะคล้ายๆ กัน โดยวงจร H-bridge จะประกอบด้วยสวิตซ์ 4 ตัวที่มีการต่อกันเป็นโครงสร้างรูป H ดังรูปที่ 2.4



รูปที่ 2.4 เป็นโครงสร้างของวงจร H-bridge ที่ใช้ในการควบคุมมอเตอร์ โดยสวิตซ์ 4 ตัวจะถูกควบคุมผ่านทางไมโครคอนโทรลเลอร์ และจะพิจารณาทิศทางโดยกระแสที่ใส่เข้าไปในมอเตอร์ ซึ่งในการเปลี่ยนทิศทางของกระแสจะทำให้เกิดการเปลี่ยนทิศทางของการหมุน

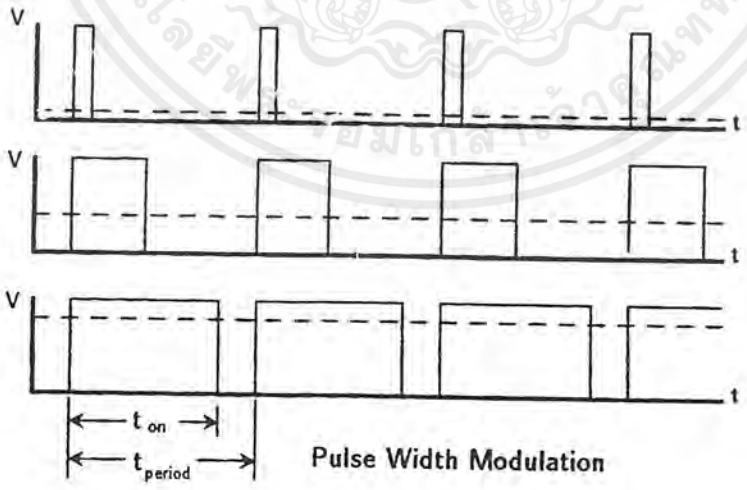
H-Bridges

ใน H-bridge สวิตซ์จะถูกเปิดและปิดโดยการใส่แรงดันตรงขั้วในการกำหนดทิศทางการหมุน หรือใส่แรงดันสลับขั้วเพื่อให้หมุนในทิศทางตรงข้าม ตัวอย่าง ถ้าสวิตซ์ S1 และ S4 ในรูปที่ 2.4 ถูกปิด ในขณะที่สวิตซ์ S2 และ S3 ถูกเปิด กระแสจะวิ่งจากซ้ายไปขวาในมอเตอร์ และเมื่อสวิตซ์ S2 และ S3 ถูกปิดและสวิตซ์ S1 และ S4 ถูกเปิด กระแสจะวิ่งจากขวาไปซ้ายเป็นเหตุให้มอเตอร์วิ่งกลับทาง

ในการควบคุมความเร็วของมอเตอร์ สวิตซ์จะถูกเปิดและปิดที่อัตราความเร็วต่างกันในการใส่ค่าเฉลี่ยแรงดันไปตกคร่อมมอเตอร์ ซึ่งเทคนิคนี้เรียกว่า Pulse-width modulation ดังในรูปที่ 2.5 ที่ซึ่ง V เป็นแรงดันที่ตกคร่อมมอเตอร์ และ t เป็นเวลา โดยตัวอย่างถ้าสวิตซ์ S1 และ S4 ใช้ pulse-width modulation ในขณะที่สวิตซ์ S2 และ S3 ถูกปล่อยให้เปิด (ดูในรูปที่ 2.4) จะทำให้เมื่อ S1 และ S4 ถูกปิด แรงดันที่ตกคร่อมมอเตอร์จะมีค่าเท่ากันและมีสภาพขั้วเหมือนกับแรงดันที่จ่าย และจะมีค่าเท่ากับ 0 V เมื่อสวิตซ์ทั้งคู่ถูกเปิด โดยการปรับความเร็วของมอเตอร์แบบ DC สามารถทำได้โดยการเปลี่ยน Pulse-Width Ratio

$$\text{Pulse-Width Ratio} = t_{on} / t_{period}$$

รีเลย์สามารถใช้ในการเปิดปิดมอเตอร์และกลับทิศทางการหมุนได้ แต่ว่ารีเลย์ไม่ค่อยได้ใช้เป็นตัวควบคุมความเร็วแบบ Pulse-width modulation เพราะว่ารีเลย์ไม่สามารถปิดเปิดได้เร็วพอ ซึ่งสวิตซ์แบบโซลิดสเตรท เช่น พวกไบโพลาร์ทรานซิสเตอร์และพวกมอสเฟต เป็นทางเลือกที่สะดวกกว่าสำหรับ pulse-width modulation



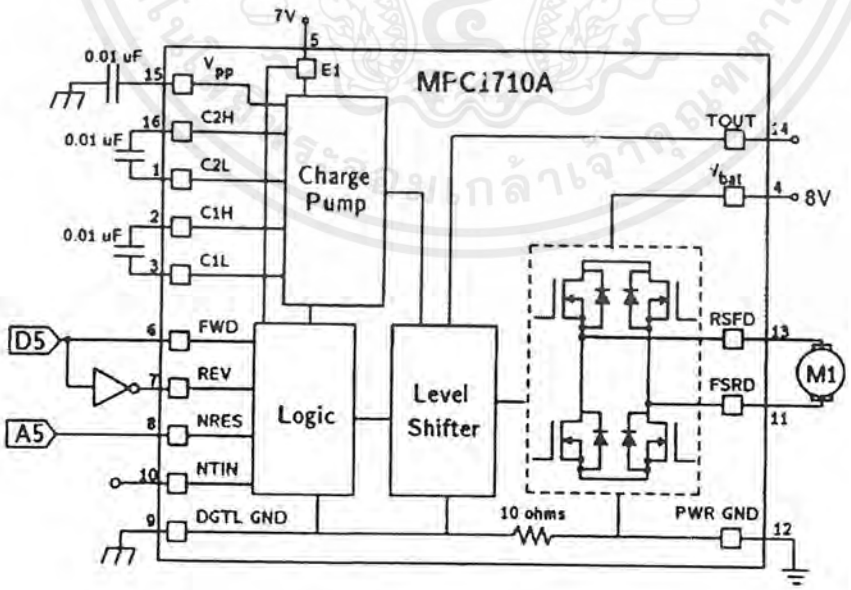
รูปที่ 2.5 เป็น Pulse-width modulation ของแรงดัน โดยการหมุนสวิตซ์ให้ H-bridge เปิดและปิด สำหรับความยาวหลายๆช่วงเวลาในการสร้างแรงดันที่แตกต่างกันคร่อมมอเตอร์ ซึ่งเส้นสีดำแทนแรงดันที่ใส่เข้าไปเมื่อสวิตซ์ถูกปิด และเส้นประ แทนผลลัพธ์ของแรงดันเฉลี่ยที่ใส่เข้าไปคร่อมมอเตอร์

ไอซี ที่ใช้ในการขับกำลังมอเตอร์

ไอซี (Integrated circuits) ที่ใช้ในการขับกำลังมอเตอร์ถูกสร้างขึ้นมาเพื่อทำให้เกิดความสะดวกในการติดต่อระหว่างมอเตอร์และไมโครคอนโทรลเลอร์ โดยวงจรในไอซี จะมีวงจรจำกัดกระแส และการป้องกันแรงดันมากเกิน ซึ่งชิปในการขับมอเตอร์ก็คือ MPC1710A จาก Motorola ดังแสดงในรูปที่ 2.6 โดยใช้ H-bridge ที่ประกอบด้วย MOSFET แบบ n-channel 4 ตัว ซึ่งมีวงจร Level Shifter และ Charge Pump รวมอยู่ในชิปเพื่อใช้ในการขับสวิตช์

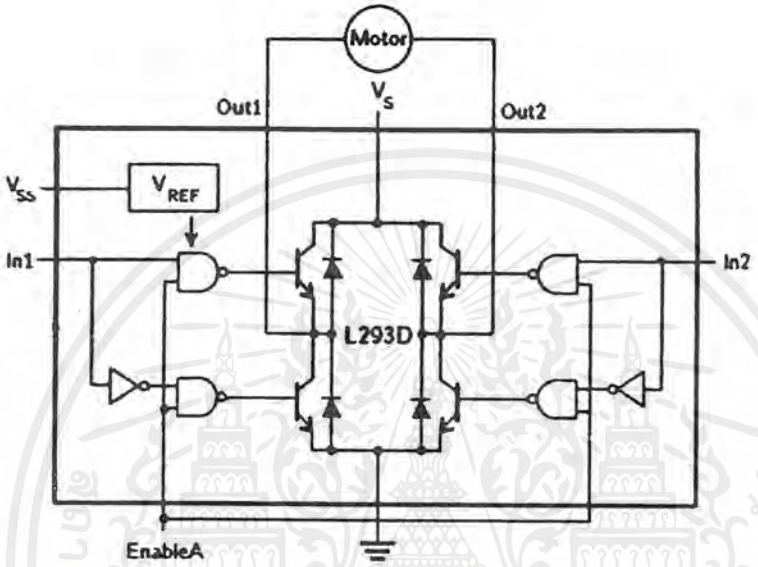
ส่วนประกอบภายนอกที่ใช้ในการเชื่อมต่อ MPC1710A กับ MC68HC11A0 คือ ตัวเก็บประจุ 3 ตัว และอินเวอร์ตเตอร์ 1 ตัวเท่านั้น โดยจะใช้พอร์ท D ขาที่ PD5 ในการกำหนดทิศทางการเคลื่อนที่ว่าเคลื่อนที่ไปข้างหน้าหรือย้อนหลังของมอเตอร์ และพอร์ท A ขาที่ A5 ในการปรับช่วงความยาวการสั่น (pulse-width modulation) สำหรับการควบคุมความเร็ว ซึ่ง MPC1017A ของ Motorola ซึ่งสามารถส่งกระแสขนาด 1 แอมป์ บนความความต้านทาน 0.4 โอห์มเมื่อเป็นกระแส source และบนความต้านทาน 0.2 โอห์ม เมื่อเป็นกระแส sink

สำหรับมอเตอร์ 2 ตัว จะต้องใช้ MPC1710A เป็นจำนวน 2 ตัว โดยมอเตอร์ตัวที่ 1 จะควบคุมโดยขา PD5 และ PA5 ซึ่งมอเตอร์อีกตัวควบคุมโดยขา PD4 และ PA6 และจากที่ MPC1710A มีรูปร่างเป็นแบบ surface-mount ซึ่งมีขนาดเล็ก 16 ขา ซึ่งเป็นการยากที่จะใช้เทคโนโลยีในการเชื่อมต่อที่มีความเร็วสูงและมีการต่อที่คดเคี้ยว ด้วยเหตุผลนี้ เราจึงใช้ชิปที่มีความสามารถมากกว่า คือ L293D จากบริษัท SGS Thompson



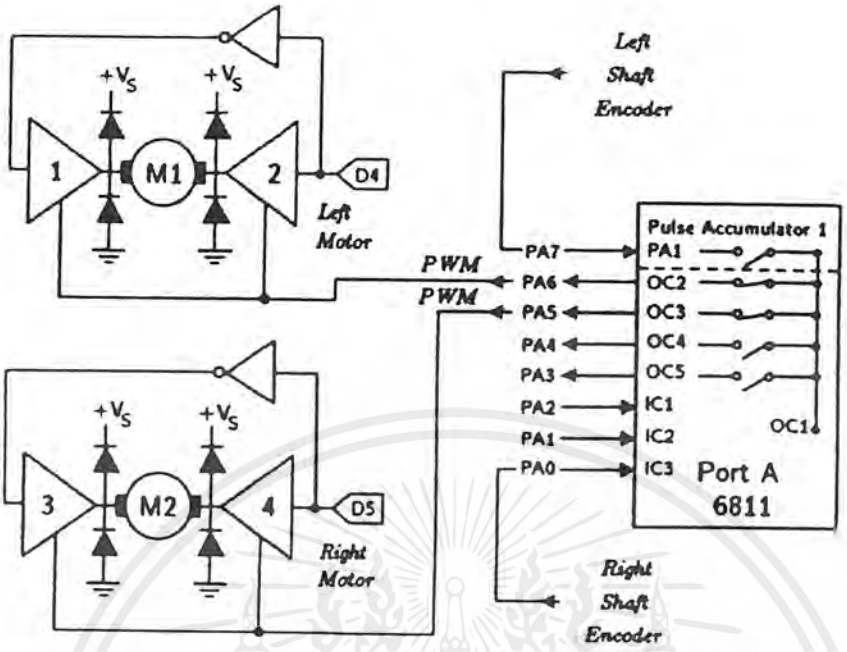
รูปที่ 2.6 เป็นชิป MPC1710A ของ Motorola ซึ่งใช้ในการขับมอเตอร์ โดยชิปตัวขับมอเตอร์ใช้ H-bridge ที่ทำจาก MOSFET แบบ n-channel

การที่เลือกใช้ L293D เพราะมีขนาดปกติ 16 ขาแบบ dual-inline-package (DIP) ซึ่งแสดงดังรูปที่ 2.7 โดยใช้ H-bridge แบบ bipolar ซึ่งสวิตซ์ทั้งหมดทำจากอุปกรณ์ชนิด n และวงจรที่เพิ่มขึ้นมาในการขับสวิตซ์ โดย L293D สามารถส่งกระแสขนาด 600 มิลลิแอมป์ไปยังมอเตอร์กับแรงดันซึมซาบ (saturation voltage) ที่ตกคร่อมขนาด 1.4 โวลต์ เมื่อเป็นกระแส source และขนาด 1.4 โวลต์ เมื่อเป็นกระแส sink



รูปที่ 2.7 เป็นไอซีตัวขับกำลังมอเตอร์ L293D ของบริษัท SGS Thompson ซึ่งการขับมอเตอร์ใช้ H-bridge ที่ทำจากทรานซิสเตอร์แบบไบโพลาร์ (bipolar transistor)

รูปที่ 2.8 เป็นการเชื่อมต่อ L293D กับหุ่นยนต์ โดย H-bridge โดย L293D มีค่าลอจิกบนชิป (On-chip logic) ซึ่งจะให้สัญญาณอินาเบิล (Enable signal) โดยอินพุทที่ไปยัง H-bridge สามารถใช้ในการกำหนดทิศทางการหมุนของมอเตอร์ และสัญญาณอินาเบิลสามารถใช้สำหรับ pulse-width modulation ซึ่งจะใช้พอร์ท D ขาที่ PD5 ในการกำหนดทิศทางของมอเตอร์ด้านขวา โดยตัวกลับสัญญาณ (inverter) ใช้ในการกำหนดด้านหนึ่งของ H-bridge ให้มีสภาพขั้วแรงดันของสัญญาณเกทให้มีค่าตรงข้ามกับสัญญาณอีกด้านหนึ่ง ซึ่งตัวอย่างสมมุติว่าถ้าสวิตซ์ S1 และ S4 ติด ในทำนองเดียวกันจะทำให้สวิตซ์ S2 และ S3 ดับ โดย H-bridge เป็น pulse-width modulation โดยการต่อขาของตัวขับของมอเตอร์ที่อยู่ด้านขวาไปยังพอร์ท A ที่ขา PA5 ซึ่งความได้เปรียบอย่างหนึ่งของ L293D มี Full H-bridge 2 ตัวบรรจุอยู่บนชิป ซึ่งหมายความว่าจะใช้ L293D เพียงตัวเดียวในการขับล้อของมอเตอร์ทั้ง 2 ล้อ โดยพอร์ท D ขาที่ PD4 ใช้ในการกำหนดทิศทางของล้อที่อยู่ทางซ้าย และพอร์ท A ขา PA6 ต่อไปยังสัญญาณอินาเบิลสำหรับเป็น pulse-width modulation



รูปที่ 2.8 เป็นชิป L293D ซึ่งใช้ในการขับมอเตอร์ของหุ่นยนต์ โดยพอร์ท D ที่ขา PD4 และ PD5 ใช้ในการเลือกทิศทางไปข้างหน้าหรือย้อนหลังของมอเตอร์ตัวซ้ายและขวาตามลำดับ ในขณะที่พอร์ท A ที่ขา PA6 และ PA5 เป็นขาที่ให้ pulse-width modulation แก่มอเตอร์ตัวซ้ายและตัวขวาตามลำดับ สังเกตว่า OC1 ใช้ในการควบคุม OC2 และ OC3

2.2 เซนเซอร์

เซนเซอร์มีอยู่ด้วยกันหลายประเภท ซึ่งสามารถเชื่อมต่อไปยังไมโครคอนโทรลเลอร์เพื่อทำการควบคุม ทำให้มีระบบเกี่ยวกับการสัมผัสรู้ต่างๆ ซึ่งจะนำไปใช้ประโยชน์ต่างๆตามรูปแบบของแต่ละเซนเซอร์

2.2.1 เซนเซอร์ทางแสง

เซนเซอร์ของแสงที่มองเห็นและเซนเซอร์ของแสงอินฟราเรดมีช่วงห่างของสเปกตรัมกว้าง ซึ่งโฟโต้เซลล์ เป็นอุปกรณ์ตรวจจับทางแสงที่ง่ายที่สุดที่ใช้ในการติดต่อกับไมโครคอนโทรลเลอร์

2.2.1.1โฟโต้รีซิสเตอร์หรือตัวต้านทานไวแสง

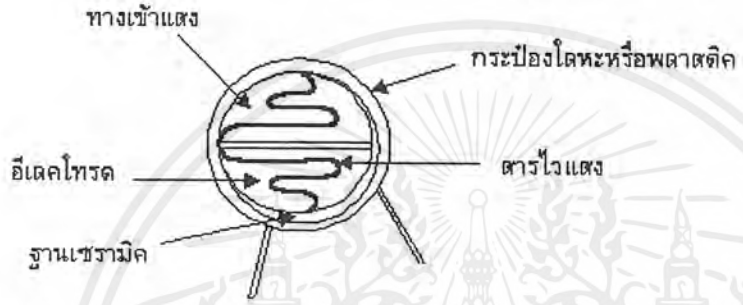
ตัวต้านทานไวแสง (Light Independent Resistor) หรือเรียกสั้น ๆ ว่า LDR ทำมาจากสารแคดเมียมซัลไฟล์ (Cds) หรือแคดเมียมซีลิไนด์ (Cdse) ซึ่งเป็นสารประกอบชนิดกึ่งตัวนำมาฉาบบนแผ่นเซรามิคที่ใช้เป็นฐานรอง แล้วต่อขาจากสารที่ฉาบเอาไว้ออกมาดังโครงสร้างในรูปที่ 2.9

คุณสมบัติทางแสง

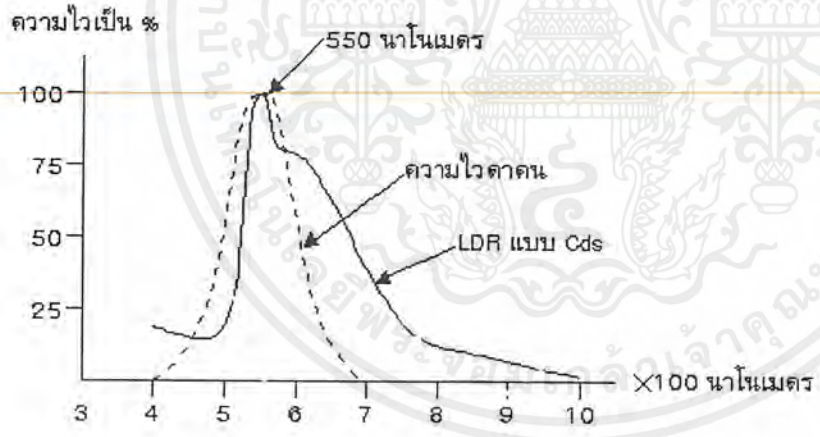
LDR ไวต่อแสงช่วงคลื่น 400-1000 นาโนเมตร ซึ่งครอบคลุมช่วงคลื่นที่ไวต่อตามนุษย์ (400-700 นาโนเมตร) นั่นคือ LDR ไวต่อแสงอาทิตย์และแสงจากหลอดไส้หรือหลอดเรืองแสงและยังไวต่อแสงอินฟาเรดที่ตามองไม่เห็นอีกด้วย (ช่วงคลื่นตั้งแต่ 700 นาโนเมตรขึ้นไป)

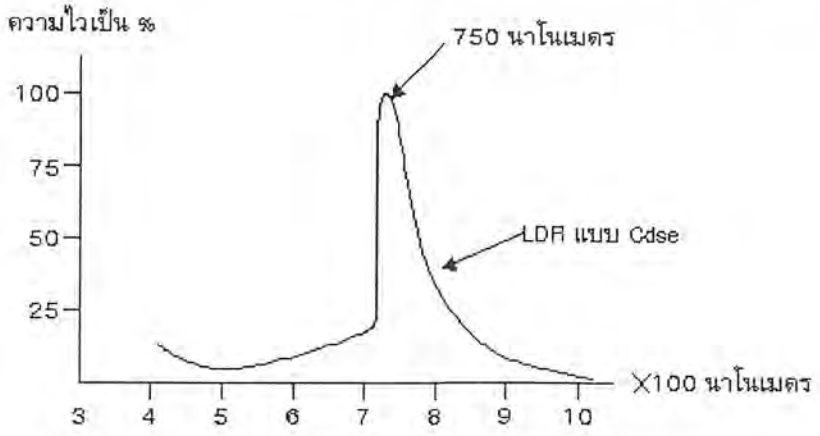
คุณสมบัติทางไฟฟ้า

อัตราส่วนของความต้านทาน LDR ขณะที่ไม่มีแสงกับในขณะที่มีแสง อาจมีค่าต่างกัน 100, 1,000, 10,000 เท่า แล้วแต่แบบหรือรุ่น

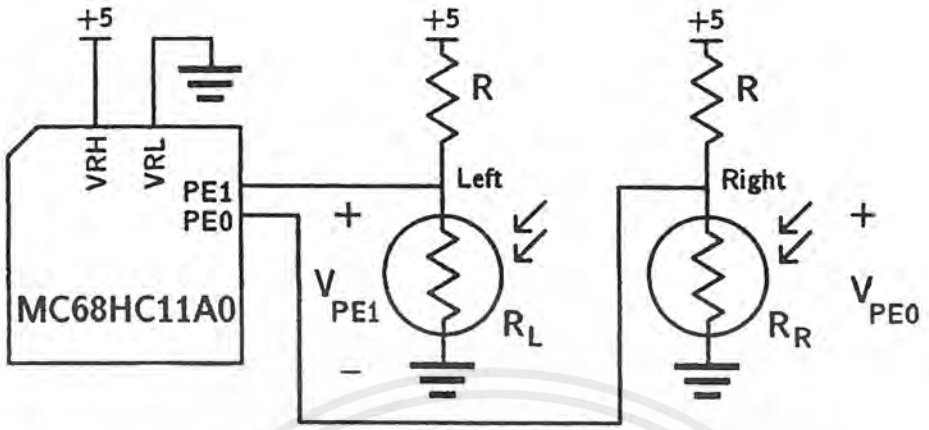


รูปที่ 2.9 ส่วนประกอบของโฟโต้รีซิสเตอร์





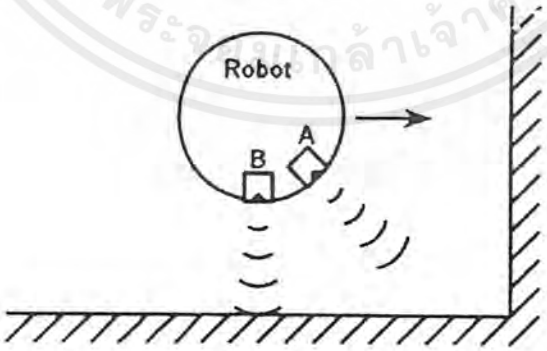
รูปที่ 2.10 แสดงความไวที่รับแสงในย่านความยาวคลื่นต่างๆ



รูปที่ 2.11 เป็นโฟโต้รีซิสเตอร์ซึ่งต่อแบบโวลต์เตจดีไวเดอร์ต่อเข้ากับพอร์ท E โดยพอร์ท E เป็นพอร์ท A/D

2.2.1.2 ตัวเซนเซอร์รังสีอินฟราเรดระยะใกล้

ตัวเซนเซอร์ประเภทนี้จะมีช่วงกว้างของลำแสงที่แคบ มากกว่าเครื่องวัดระยะแบบการใช้คลื่นเสียง (Sonar range finder) โดยการเคลื่อนที่ไปตามกำแพงจะใช้ตัวตรวจจับ 2 ตัว (โดยตัวหนึ่งชี้ไปที่กำแพง และอีกตัวชี้ทำมุม 45 องศามากกว่าที่อยู่ส่วนหน้า) ดังในรูปที่ 2.12 ซึ่งหุ่นยนต์จะสามารถเคลื่อนที่ไปตามกำแพงโดยใช้ตัวตรวจจับเพียง 1 ตัว โดยการเปลี่ยนวิธีคล้ายๆกับหางเรือ ซึ่งในกรณีนี้จะเคลื่อนที่จากกำแพงเป็นมุมโค้งจากกำแพงเมื่อเซนเซอร์ตรวจจับบางอย่างได้ และเคลื่อนที่เป็นมุมไปข้างหน้าเมื่อตรวจจับสัญญาณไม่ได้

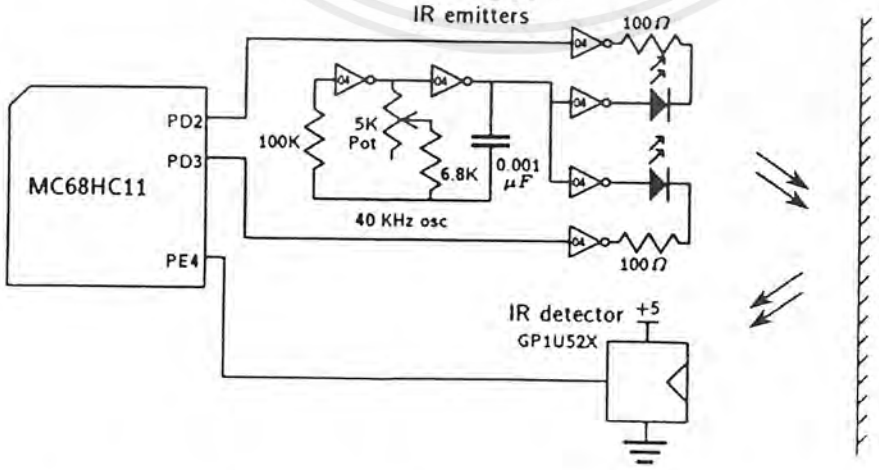


รูปที่ 2.12 เป็นรูปหุ่นยนต์ที่เคลื่อนที่ไปตามกำแพงโดยใช้ตัวตรวจจับ 2 ตัว คือ A และ B โดยถ้าไม่มีตัวตรวจจับตัวใดเจอสิ่งกีดขวาง จะทำให้หุ่นยนต์เคลื่อนที่เป็นมุมไปทางขวาเพื่อค้นหากำแพง แต่เมื่อเซนเซอร์ B เพียงตัวเดียวเจอสิ่งกีดขวาง หุ่นยนต์จะเคลื่อนที่ไปข้างหน้า แต่ถ้าเซนเซอร์ A พบสิ่งกีดขวางเพียงตัวเดียวหรือพร้อมกับ B จะทำให้หุ่นยนต์เลี้ยวซ้าย

ตัวเปล่งแสงอินฟราเรดและตัวตรวจจับดังรูปที่ 2.13 โดยตัวเปล่งแสงอยู่ข้างบนเป็น LED ซึ่งผลิตจากแกลเลียมอาร์เซไนด์ (GaAs) ซึ่งแผ่พลังงานอินฟราเรดระยะใกล้ที่ 880 นาโนเมตร โดยตัวแผ่และตัวตรวจจับ (โฟโต้ไดโอดและโฟโต้ทรานซิสเตอร์) สามารถซื้อได้จากบริษัทที่ผลิตเซมิคอนดักเตอร์คือของ Siemens, Motorola, Hewlett-Packard และอื่นๆ และจากรูปที่ 2.13 ตัวข้างล่างเป็นตัวตรวจจับที่บริษัท Sharp เป็นผู้ผลิต คือ GP1U52X ซึ่งประกอบด้วย ตัวขยายสัญญาณที่สมบูรณ์, ตัวกรองสัญญาณ, และตัวจำกัดค่า

รูปที่ 2.13 เป็นเซนเซอร์อินฟราเรดระยะใกล้ โดยตัวล่างเป็นตัวตรวจจับของ Sharp และตัวที่อยู่ข้างบนเป็น LED อินฟราเรดระยะใกล้

ตัวตรวจจับของ Sharp จะตอบสนองต่อพาหะโมดูเลตที่ปล่อยออกมาโดย LED อินฟราเรดระยะใกล้ ซึ่งหมายความว่า นักเขียนโปรแกรมจะต้องตระหนักในการกระพริบของ LED คล้ายกับที่ตัวตรวจจับทำ ซึ่งการโมดูเลตนี้จะทำให้อัตราสัญญาณต่อสัญญาณรบกวนเพิ่มมากขึ้น (signal-to-noise) โดยวงจรอย่างน้อยที่สุดต้องมี ไอซี 74HC04 ซึ่งเป็นอินเวอร์เตอร์มาต่อเพื่อให้เป็นเซนเซอร์ที่ใกล้เคียงไปยัง MC68HC11 ดังแสดงในรูปที่ 2.14



รูปที่ 2.14 เป็นตัวตรวจจับแสงอินฟราเรดระยะใกล้ G1U52X ของ Sharp ที่ตรวจจับพลังงานที่แพร่จาก LED อินฟราเรดระยะใกล้เหมือนกับ SFH 484 LED ของ Siemens

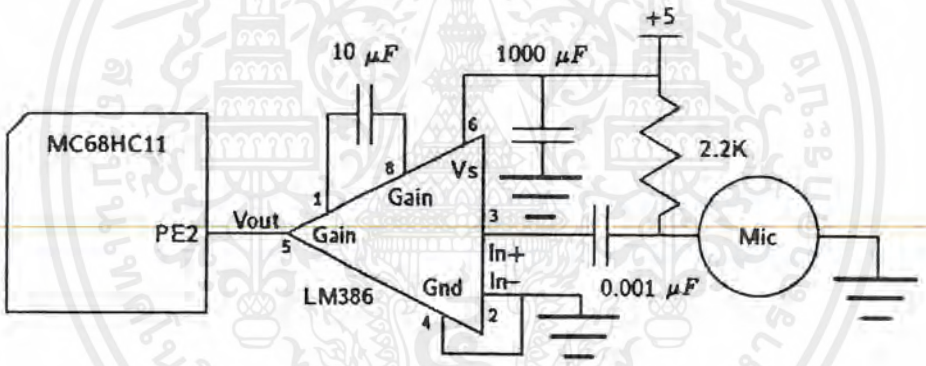
2.2.2 เซนเซอร์ทางเสียง

เซนเซอร์ของเสียงในย่านที่สามารถได้ยินจะทำให้หุ่นยนต์สามารถมีลักษณะพิเศษขึ้นในการทำงานของมันได้ ซึ่งจะช่วยหุ่นยนต์ในการตรวจจับและหลบหลีกสิ่งกีดขวางข้างหน้าได้ เซนเซอร์ที่ใช้ในโครงงานนี้เราจะใช้ไมโครโฟน

ไมโครโฟน

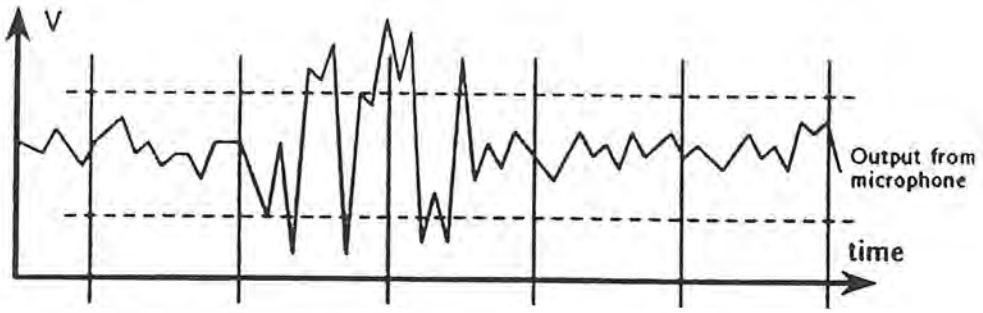
ไมโครโฟนสามารถเชื่อมต่อไปยังไมโครคอนโทรลเลอร์ได้อย่างง่ายดาย โดยจะมีผลต่อพฤติกรรมของหุ่นยนต์คือ การเคลื่อนที่ไปข้างหน้าหรือออกห่างจากเสียง, การฟังรูปแบบเฉพาะของเสียง, การจำกัดให้อยู่ในขอบเขตของตำแหน่งเสียง

สัญญาณจากไมโครโฟนแบบธรรมดาจะมีการถูกขยายก่อนการอ่านสัญญาณโดยไมโครคอนโทรลเลอร์ ดังรูปที่ 2.15 โดยการใช้ LM386 ออม-แอมป์ ซึ่งเอาท์พุทที่ถูกขยายจะต่อไปยังขา A/D ที่ port E



รูปที่ 2.15 เป็นวงจรการขยายของไมโครโฟน โดยใช้ LM336 ออม-แอมป์

ปัญหาสำคัญอย่างหนึ่งของการใช้ไมโครโฟนก็คือ ต้องการสัญญาณตัวอย่างหลายๆ ความถี่ โดยในรูปที่ 2.16 เป็นเอาท์พุทของสัญญาณที่ออกมาจากไมโครโฟน ซึ่งถ้าจะให้หุ่นยนต์พยายามที่จะค้นหาเสียงปรบมือหรือนกหวีด เราอาจจะต้องพยายามทำเสียงสัญญาณหลายๆครั้งเพื่อจะทำให้การค้นหาของหุ่นยนต์ไม่เกิดการผิดพลาด โดยการอ่านค่าของไมโครโฟนจะอ่านเป็นค่าแรงดันซึ่งมีค่าอยู่ระหว่าง 0 โวลต์และ 5 โวลต์ ซึ่งสัญญาณที่เกิดจากการปรบมืออาจจะเกิดแค่เพียงมิลลิวินาที นั่นหมายความว่า โดยไมโครคอนโทรลเลอร์จะต้องทำการตรวจสอบเอาท์พุทของไมโครโฟนบ่อยๆ ทำให้ต้องใช้เวลาในการทำงานของไมโครคอนโทรลเลอร์ทั้งหมดในการมองหาสัญญาณความถี่สูงๆ หรือสัญญาณที่เร็วมากๆ จึงอาจจำเป็นต้องใช้ไมโครคอนโทรลเลอร์หรือส่วนของฮาร์ดแวร์ที่ใช้เฉพาะอย่างเพื่อทำงานอย่างเดียวในการจับสัญญาณของไมโครโฟน



รูปที่ 2.16 หุ่นยนต์จะมีการตอบสนองเมื่อค้นหา (Detect) สัญญาณเสียง ซึ่งเมื่อสัญญาณจากไมโครโฟนอยู่เหนือเส้นประที่อยู่ข้างบนหรือต่ำกว่าเส้นประที่อยู่ข้างล่าง โดยแต่ละเส้นของแนวดิ่งแทนตัวอย่าง ในขณะเมื่อไมโครคอนโทรลเลอร์จะอ่านค่าจากตัวแปลงสัญญาณ A/D ที่ต่อกับไมโครโฟน นอกจากเสียว่าตัวอย่างมีความถี่ภายในเกิดขึ้นหลายความถี่ ซึ่งจะทำให้เสียงที่เราสนใจอาจเกิดการผิดพลาดได้ง่าย

ปัญหาที่สำคัญอีกอันหนึ่ง คือไมโครโฟนที่ติดตั้งบนตัวของหุ่นยนต์ มักจะค้นหา (Detect) เสียงที่เกิดจากมอเตอร์ของตัวเอง จึงจำเป็นที่ต้องหาวิธีในการป้องกันไมโครโฟนจากเสียงรบกวนเหล่านี้ อาทิเช่น เพิ่มวงจร High pass filters

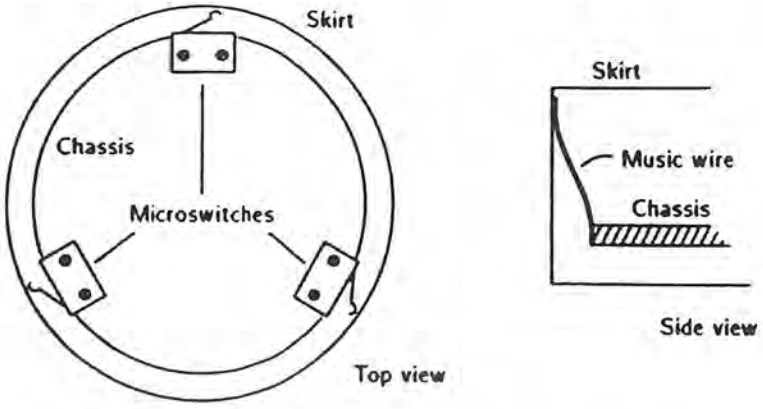
2.2.3 เซนเซอร์ทางแรง

เซนเซอร์ทางแรงเป็นเซนเซอร์ที่ได้รับการยอมรับว่ามีความน่าเชื่อได้มากที่สุด โดยมีคลื่นรบกวนต่ำมาก และสร้างสัญญาณในการเชื่อมต่อได้ง่าย ซึ่งเซนเซอร์ทางแรงจะใช้ในการพิจารณาเมื่อหุ่นยนต์ได้รับการสัมผัสกับวัตถุ โดยจะทำให้หุ่นยนต์เคลื่อนที่ออกจากวัตถุเมื่อมีการชนกัน

<u>ไมโครสวิตช์</u>

ไมโครสวิตช์เป็นอุปกรณ์ที่มีขนาดเล็กและเป็นชนิดโมเมนทารี (Momentary switch) ซึ่งสามารถใช้ในการเชื่อมต่อเป็นสัญญาณกระแทก (bumper) เมื่อหุ่นยนต์เคลื่อนที่ไปกระทบกับสิ่งกีดขวาง

รูปที่ 2.17 เป็นการใช้ไมโครสวิตช์ในการหาการชนระหว่างหุ่นยนต์และสิ่งกีดขวาง โดยเมื่อหุ่นยนต์ชนกับวัตถุทำให้มีสวิตช์ 1 หรือ 2 ตัวจะถูกปิด ซึ่งเป็นการบอกตำแหน่งของหุ่นยนต์กับสิ่งกีดขวางที่กระทบกัน

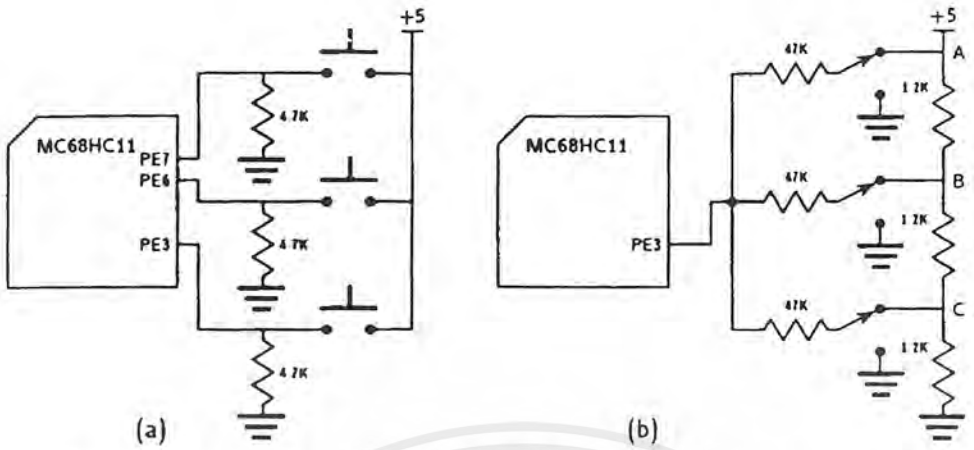


รูปที่ 2.17 เป็นตัวเซนเซอร์ทางแรงแบบกระแทก (bumper) ซึ่งนำมาใช้ในหุ่นยนต์รูปทรงกระบอก โดยใช้ไมโครสวิตช์ 3 ตัว มาเรียงให้สมดุลรอบๆ บริเวณของโครงหุ่นยนต์

รูปที่ 2.18 เป็น 2 วิธีในการเชื่อมต่อสวิตช์ที่ใช้ในการกระแทกไปยังไมโครคอนโทรลเลอร์ โดยในรูปที่ 2.18(a) เป็นการต่อแบบตรงๆ คือ ขาหนึ่งๆของพอร์ท E ต่อไปยังแต่ละตัวของสวิตช์ ซึ่งเมื่อหุ่นยนต์ชนกับสิ่งกีดขวาง สวิตช์ 1 หรือ 2 จะถูกปิดจะเกิดการเปลี่ยนสถานะของบิตที่ตรงกันจาก 0 เป็น 1 โดยวิธีนี้จะสามารถเข้าใจได้ง่ายแต่ว่าจะใช้ขาอินพุทของ MC68HC11 เปลืองถึง 3 ขา

ในรูปที่ 2.18(b) เป็นอีกวิธีหนึ่ง ซึ่งใช้ขาอินพุทเพียง 1 ขาเท่านั้น โดยการเชื่อมต่อของตัวต้านทานจะใช้ในการสร้างแรงดันที่แตกต่างกันออกไปที่ขาอินพุทของ MC68HC11 ซึ่งขึ้นอยู่กับว่าสวิตช์ตัวไหนถูกปิด โดยใช้โหมดของ A/D เข้ามาช่วย ซึ่งตัวขับซอฟแวร์จะไปอ่านที่ขา PE3 ทำการปรับเปลี่ยนและกำหนดค่า 1 ใน 8 บิต โดยค่าบิตที่ได้จะเป็นการบอกว่าสวิตช์หรือกลุ่มของสวิตช์ตัวไหนถูกปิด

ในการวิเคราะห์จะแสดงดังในรูปที่ 2.18(b) โดยกระแสจะวิ่งมาจากแหล่งจ่ายไฟ +5 โวลต์ มายังความต้านทานคือ 1.2 กิโลโอห์มอีก 3 ตัว ตรงไปยังสายดิน (ground) ซึ่งเป็นการเปรียบเทียบกระแสที่ไหลผ่านมายังส่วนต่างๆของวงจร โดยในการประมาณค่าดังตัวอย่างในรูป ซึ่งกำลังงานการแบ่งแรงดัน (voltage divider) จะแบ่งออกเป็น 1/4, 1/2, และ 1 คูณกับค่าแรงดันที่จ่าย ดังนั้นผลรวมของแรงดันจะเป็น 1/3 * (A + B + C) ซึ่งแต่ละจุดของ A, B, และ C แต่ละตัวต่อกันไปยังสายดิน โดย A/D จะใช้ในการแปลงค่าเป็นดิจิตอลระหว่างค่า 0 และ 225 ซึ่งกลุ่มแรงดันที่อ่านได้จะมีค่าเป็น 1/3 ของ 225 คูณกับค่าผลรวมของแรงดันจากสวิตช์ที่ถูกกด ตัวอย่างถ้ามีสวิตช์ A ตัวเดียวที่ถูกกด (ต่อไปยังสายดิน) เอาท์พุทที่ออกมาจาก A/D จะเป็น 1/3 * 225 * (0 + 1/2 + 1/4) ≅ 64 และเมื่อสวิตช์ B และ C ถูกกดพร้อมกัน เอาท์พุทจะมีค่าเป็น 85



รูปที่ 2.18 เป็นการต่อวงจรเซนเซอร์ทางแรง 2 แบบ โดยในรูป (a) สวิตช์แต่ละตัวจะต่อแยกไปยังแต่ละขาของพอร์ท E ส่วนในรูป (b) เป็นการต่อสวิตช์การกระแทกทั้งหมดไปยังขาของพอร์ท E เพียงขาเดียว ซึ่งจะใช้ A/D มาช่วยในการพิจารณาว่าสวิตช์ตัวไหนถูกปิด

2.3 แหล่งจ่ายไฟ

ในการเลือกแหล่งจ่ายไฟ เราจะต้องรู้ส่วนสำคัญและศึกษาคุณสมบัติ โดยแบ่งออกเป็น

ความสามารถในการประจุ (Rechargeability) แหล่งจ่ายไฟที่ไม่สามารถประจุไฟฟ้าใหม่ได้จะเป็นแบบ primary cell ในขณะที่อีกแบบสามารถประจุไฟฟ้าเข้าไปใหม่ได้จะเป็นแบบ secondary หรือ storage battery

ความหนาแน่นของพลังงาน (Energy density) จำนวนของพลังงานสูงสุดต่อหน่วยมวลของแต่ละแหล่งจ่ายไฟที่สามารถเก็บได้ ซึ่งปกติจะวัดในหน่วย วัตต์-ชั่วโมง/กิโลกรัม (Wh-kg) หรือวัดในหน่วยของ พลังงานต่อหน่วยปริมาตร

ความจุ (Capacity) ความจุของแหล่งจ่ายไฟ คือพลังงานที่เก็บอยู่ในเซลล์ ซึ่งโดยปกติหน่วยที่ใช้กัน คือ แอมป์-ชั่วโมง หรือ มิลลิแอมป์-ชั่วโมง โดยความจุเกิดจากความหนาแน่นของพลังงานและมวลของแหล่งจ่ายไฟ

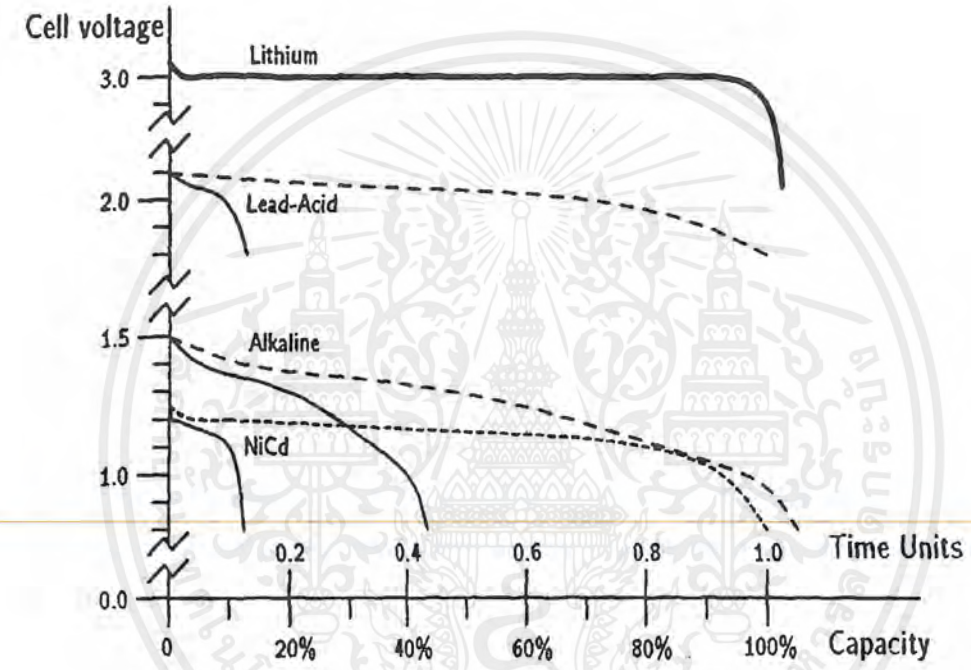
แรงดัน (Voltage) แรงดันที่เกิดโดย primary cell เป็นคุณลักษณะของปฏิกริยาเคมีเฉพาะที่เกิดขึ้นในแหล่งจ่ายไฟ โดยแรงดันจะขึ้นอยู่กับสภาวะของประจุในเซลล์

อัตราการคายประจุ (Discharge rate) เป็นอัตราที่แหล่งจ่ายไฟคายประจุในหน่วยของกระแส ซึ่งอัตราการคายตัวของประจุสูงสุดจะกำหนดโดย ค่าความต้านทานภายในของแหล่งจ่ายไฟ

ช่วงชีวิต (Shelf life) แหล่งจ่ายไฟจะคายประจุแม้แต่เมื่อไม่ได้ใส่โหลดภายนอก ซึ่งก็จะเป็นการวัดความเร็วของการเกิดการคายประจุนี้

การขึ้นอยู่กับอุณหภูมิ (Temperature dependence) คุณสมบัติของแหล่งจ่ายไฟส่วนมาก ทั้งในเรื่องของค่าความจุและช่วงชีวิตจะมีผลขึ้นอยู่กับอุณหภูมิ

แหล่งจ่ายไฟในอุดมคติจะต้องมีความหนาแน่นของพลังงานสูง มีแรงดันสม่ำเสมอระหว่างการคายประจุ มีความต้านทานภายในต่ำ และสามารถที่จะคายประจุได้อย่างรวดเร็ว ซึ่งควรจะทนต่ออุณหภูมิได้อย่างสุดๆ มีช่วงชีวิตที่ไม่จำกัด สามารถที่จะประจุไฟเข้าไปใหม่ได้ และขายในราคาต่ำ แต่ในตอนนี้ยังไม่มีเทคโนโลยีอันไหนที่จะทำได้ดังที่กล่าวมาแล้ว จึงจำเป็นต้องศึกษาคุณสมบัติเหล่านี้ให้มีความเหมาะสมกับความต้องการของงานที่ใช้ ซึ่งข้อมูลในตารางที่ 2.1 และรูปที่ 2.19 อาจจะช่วยแนะนำในการเลือกแหล่งจ่ายไฟเพื่อนำไปใช้งาน



รูปที่ 2.19 กราฟเปรียบเทียบ ความจุและความต่างศักย์ของถ่านแต่ละชนิด

ตารางที่ 2.1 ปริมาณและคุณสมบัติของแบตเตอรี่ชนิดต่างๆ

Battery Chemistry	Recharge	Energy Density (Whr/kg)	Cell Voltage	Typical Capacity (mAh)	Internal Resistance (ohms)	Comments
Alkaline	No	130	1.5	AA 1400 C 4500 D 10000	0.1	Most common primary battery
Lead-Acid	Yes	40	2.0	1.2 - 120 Ah	C-size 0.006	Available in a wide variety of sizes
Lithium	No	300	3.0	A 1800 C 5000 D 14000	0.3	Execellent energy density, high unit cost
Mercury	No	120	1.35	Coin 190	10	
NiCd	Yes	38	1.2	AA 500 C 1800 D 4000	0.009	Low internal resistance, available from many sources
NiMH	Yes	57	1.3	AA 1100 4/3A 2300		Better energy density than NiCd, expensive
Silver	No	130	1.6	Coin 180	10	
Zinc-Air	No	310	1.4			High energy density but not widly available, limited range of sizes
Carbon-Zinc	No	75	1.5	D 6000		Inexpensive but obsolete

All numbers listed here are approximate. Precise values depend on the details of the particular battery.
Some values depend on the battery's state of charge, temperature, and discharge history.

2.4 ไมโครคอนโทรลเลอร์

ปัจจุบันไมโครคอนโทรลเลอร์มีมากมายหลายเบอร์ และเป็นที่แพร่หลายอย่างยิ่งในการใช้งานควบคุมอุปกรณ์ทางอิเล็กทรอนิกส์ ไมโครคอนโทรลเลอร์เป็นไมโครโปรเซสเซอร์ประเภทหนึ่งที่ได้รับการออกแบบมาเพื่อใช้งานกับระบบควบคุมที่มีขนาดเล็ก โดยภายในไอซีไมโคคอนโทรลเลอร์หนึ่งตัวจะประกอบด้วยหน่วยการทำงานหลักของระบบคอมพิวเตอร์ครบถ้วน เช่นหน่วยประมวลผลกลางหรือ CPU หน่วยความจำ พอร์ตในการติดต่อหรือควบคุมอุปกรณ์ต่างๆเป็น ต้น ซึ่งหากว่าเป็นการใช้งานไมโครโปรเซสเซอร์ทั่วไปก็จะต้องใช้ไอซีภายนอกมาประกอบเพื่อทำหน้าที่เหล่านี้ ดังนั้นจึงอาจกล่าวได้ว่าไมโครคอนโทรลเลอร์เป็นระบบคอมพิวเตอร์เพื่องานควบคุมที่สมบูรณ์โดยบรรจุภายในตัวไอซีเพียงตัวเท่านั้น ในบางครั้งจึงอาจพบว่ามีการเรียกไมโครคอนโทรลเลอร์ว่าเป็น ระบบไมโครคอมพิวเตอร์ชิปเดี่ยว

บทที่ 3

MC68HC11

3.1 ลักษณะทั่วไปของไมโครคอนโทรลเลอร์ 68HC11

ไมโครคอนโทรลเลอร์ 68HC11 ได้ถูกออกแบบโดยรวมอุปกรณ์ที่จำเป็นในการใช้งานไว้ภายในตัวของมันเอง ไม่ว่าจะเป็นพอร์ทอินพุท-เอาท์พุท วงจรตั้งเวลา ตัวนับระบบการอินเตอร์รัปต์ หน่วยความจำรอม แรม และอีอีพรอม วงจรแปลงสัญญาณอนาลอกเป็นดิจิตอล ระบบป้องกันความผิดพลาดที่เกิดขึ้นจากโปรแกรม และส่วนติดต่อสื่อสารของข้อมูลผ่านทางพอร์ทอนุกรม สามารถสรุปคุณสมบัติทั่วไปของ 68HC11 ได้เป็น 2 ลักษณะคือ ทางฮาร์ดแวร์และซอฟแวร์ดังนี้

3.1.1 คุณสมบัติทางฮาร์ดแวร์

- ซีพียูขนาด 8 บิต
- มีหน่วยความจำรอมภายในขนาด 4, 8 หรือ 12 กิโลไบต์ (บางเบอร์จะไม่มีส่วนนี้)
- มีหน่วยความจำอีอีพรอมภายในขนาด 512 ไบต์หรือ 2 กิโลไบต์
- มีหน่วยความจำแรมภายในขนาด 192, 256 หรือ 512 ไบต์
- มีตัวตั้งเวลาและตัวนับขนาด 8 บิต
- มีวงจรแปลงสัญญาณอนาลอกเป็นดิจิตอล 8 บิต 8 ช่อง
- มีวงจรพัลส์แอคคิวมูเลเตอร์ขนาด 8 บิต
- มีส่วนติดต่อสื่อสารผ่านทางพอร์ทอนุกรม (Serial Communication Interface : SCI)
- มีวงจรเชื่อมต่ออุปกรณ์อนุกรม (Serial Peripheral Interface : SPI)
- มีวงจรรีลไทม์อินเตอร์รัปต์
- มีระบบวอตช์ด็อก
- สามารถขยายตัวตั้งเวลาเป็นแบบ 16 บิตได้ โดยมีฟังก์ชันการทำงานเพิ่มเติมคือ วงจรปรีสเกลเลอร์ที่สามารถโปรแกรมได้ 4 สเกล อินพุทสำหรับตรวจจับสัญญาณ 3 อินพุท และเอาท์พุทสำหรับวงจรสัญญาณเปรียบเทียบ 5 เอาท์พุท
- บรรจุในตัวถังหลายแบบคือ DIP ขนาด 48 ขา, PLCC ขนาด 52 ขาหรือ QFC ขนาด 64 ขา
- มีระบบอินเตอร์รัปต์ 2 ระดับ จาก 21 แหล่ง
- สามารถติดต่อหน่วยความจำภายนอกได้ 64 กิโลไบต์
- สามารถติดต่อเชื่อมกันทำงานเป็นแบบมัลติโปรเซสเซอร์ได้

3.1.2 **คุณสมบัติทางซอฟแวร์**

- มีชุดคำสั่งเพิ่มเติมมากกว่าไมโครโปรเซสเซอร์เบอร์ 6800 และ 6801 จึงสามารถใช้ชุดคำสั่งเดียวกับ 6800 และ 6801 ได้
- สามารถทำการหารเลข 16 บิตโดยได้ผลลัพธ์เป็นตัวเลข 16 บิตและเศษขนาด 16 บิต
- สามารถประมวลผลข้อมูลละเอียดถึงระดับบิต
- มีโหมดการทำงาน WAIT และโหมด STOP เพื่อประหยัดพลังงาน

MC68HC11 เป็นชิปไมโครคอนโทรลเลอร์ ที่ได้รับการผลิตโดยเทคโนโลยี HCMOS (High-density CMOS) จึงทำให้มีความเร็วในการทำงานสูง ในขณะที่ใช้กำลังงานไฟฟ้าต่ำ

3.1.3 **ส่วนประกอบหลักของ** MC68HC11

ไมโครคอนโทรลเลอร์ตระกูล 68HC11 ที่นำมาอ้างอิงเพื่อการนำเสนอคือ MC68HC11A8 ซึ่งจะประกอบด้วยส่วนหลักๆ 6 ส่วน

1. **ซีพียู** ในส่วนนี้มีส่วนประกอบย่อยออกเป็น 3 ส่วนคือ ส่วนควบคุมโหมดการทำงานของชิป (MODE), ส่วนกำเนิดสัญญาณนาฬิกา (CLK) และส่วนลอจิกอินเตอร์รัปต์ (INT) โดยทั้งสามส่วนนี้จะได้รับการควบคุมการทำงานจากแก่นซีพียู
2. **หน่วยความจำ** ชิป MC68HC11A8 มีครบทั้ง 3 แบบคือ รอม แรม และอีอีพรอม ซึ่งการต่อและเรียกใช้หน่วยความจำเบอร์นี้จะขึ้นอยู่กับส่วนซีพียูเป็นหลัก
3. **แอกคิวมูเลเตอร์-วอตช์ด็อก-ตัวตั้งเวลาหลัก** ส่วนนี้จะมีการติดต่อกับพอร์ท A โดยใช้พอร์ท A เป็นทางผ่านของข้อมูล พัลส์แอกคิวมูเลเตอร์จะใช้พอร์ท PA7 ในขณะที่ส่วนตัวตั้งเวลาหลักจะใช้ PA3-PA6 โดยในชิปยังมีวงจรวอตช์ด็อกเพื่อช่วยให้ชิปสามารถทำงานได้อย่างต่อเนื่อง แม้ว่าจะมีรีเซตระบบอยู่บ่อยๆก็ตาม
4. **พอร์ทอินพุทและเอาต์พุท** ใน MC68HC11A8 จะมีพอร์ทอินพุท เอาต์พุทด้วยกัน 5 พอร์ทดังนี้
   - พอร์ท A เป็นทั้งพอร์ทอินพุทและเอาต์พุท โดยมีการทำงานแยกกันคือ PA7 เป็นพอร์ทที่สามารถส่งผ่านข้อมูลได้สองทิศทาง ในขณะที่ PA3-PA6 เป็นพอร์ทเอาต์พุท ของตัวตั้งเวลาหลัก และ PA0-PA2 เป็นพอร์ทอินพุท
   - พอร์ท B เป็นพอร์ทเอาต์พุท
   - พอร์ท C เป็นพอร์ทอินพุทเอาต์พุทสามารถส่งข้อมูลได้ 2 ทิศทาง โดยที่พอร์ทนี้จะทำหน้าที่เป็นบัสแอดเดรสไบต์ต่ำและบัสข้อมูลด้วย โดยได้รับการควบคุมการทำงานแบบมัลติเพล็กซ์ จึงไม่เกิดความสับสน

- พอร์ท D เป็นพอร์ทที่ใช้ในการถ่ายทอดสัญญาณจากส่วนเชื่อมต่ออุปกรณ์อนุกรมและส่วนสื่อสารข้อมูลอนุกรม จึงมีลักษณะเป็นพอร์ทอินพุทเอาต์พุทที่สามารถส่งผ่านข้อมูลได้ 2 ทิศทาง
- พอร์ท E เป็นพอร์ทอินพุทสำหรับวงจรแปลงสัญญาณอนาลอกเป็นดิจิตอล (Analog Digital Converter)

5. **วงจรแปลงสัญญาณอนาลอกเป็นดิจิตอล** (ADC) เป็นวงจรที่ช่วยเสริมให้ชิปไมโครคอนโทรลเลอร์มีประสิทธิภาพมากขึ้น โดยปกติแล้ว 68HC11 จะมีวงจร ADC 8 ช่อง แต่ในเบอร์ MC68HC11A8 ซึ่งเป็นรูปแบบ 48 ขานั้นจะมีวงจร ADC นี้เพียง 4 ช่อง แต่ถ้าเป็นรูปแบบ 52 ขาคือเบอร์ MC68HC11A8CFN1 ก็จะมีวงจร ADC ครบ 8 ช่อง การเรียกใช้งานวงจรส่วนนี้จะได้รับการควบคุมจากซีพียู
6. **สโตรบ/แฮนด์เชก** จะทำงานร่วมกับส่วนขยายบัส ทั้งนี้เพื่อให้ซีพียูสามารถทำงานได้กับแอดเดรสถึง 16 บิต ที่ส่วนนี้จะมีสัญญาณที่สำคัญๆอยู่ 2 สัญญาณคือ STRB/R/$\overline{W}$ และ STRA/AS โดยทั้งสองสัญญาณคือ สัญญาณสโตรปเพื่อให้ชิปสามารถทำการอ่านเขียนข้อมูลได้ ส่วนนี้มีวงจรแฮนด์เชกเพื่อตรวจสอบความพร้อมในการรับส่งข้อมูลของชิปกับอุปกรณ์ภายนอก

3.2 โหมดการทำงานของ 68HC11

68HC11 มีโหมดการทำงาน 4 โหมด ดังนี้

1. **โหมดซิงเกิลชิป** (Single-chip Operatig Mode)

โหมดนี้ 68HC11 จะทำงานด้วยลำโพงตัวเดียว ภายใน 68HC11 จะมีหน่วยความจำและพอร์ท ดังนั้นเมื่อทำงานในโหมดซิงเกิลชิป ตัวซีพียูก็จะทำการเรียกข้อมูลที่อยู่ในหน่วยความจำภายในออกมาใช้งาน และใช้พอร์ทที่มีอยู่รับหรือส่งค่าออกไป

ดังนั้นในโหมดนี้จะไม่ต้องการหน่วยความจำภายนอก หรือชิปพอร์ทอินพุทเอาต์พุทภายนอก เมื่อใช้งานในโหมดนี้จะช่วยลดค่าใช้จ่ายลงได้มาก เพราะไม่ต้องมีชิปอื่นๆภายนอกมาสนับสนุนเลย และนอกจากนั้นยังมีความเชื่อถือได้สูง เพราะแถบจะไม่มีจุดต่อสัญญาณภายนอกเลยทำให้โอกาสที่จะเกิดความผิดพลาดมีลดลง แต่ในโหมดนี้จะไม่สามารถเพิ่มหน่วยความจำหรือพอร์ทเพื่อขยายระบบได้

2. **โหมดมัลติเพล็กซ์ขยาย** (Expanded Multiplexed Operating Mode)

การทำงานในโหมดนี้จะแตกต่างกับโหมดแรกโดยโปรแกรมควบคุมการทำงานของซีพียูจะอยู่ในหน่วยความจำภายนอก โดยที่พอร์ท B และ C ของ 68HC11 ทำหน้าที่เป็นบัสแอดเดรสและ

ข้อมูล (พอร์ท B ทำหน้าที่เป็นบัสแอดเดรสไบต์สูง ส่วนพอร์ท C ทำหน้าที่เป็นบัสแอดเดรสไบต์ต่ำ และบัสข้อมูล)

การที่พอร์ท C จะสามารถเป็นทั้งบัสแอดเดรสและบัสข้อมูลได้นั้นต้องใช้กระบวนการที่เรียกว่า การมัลติเพล็กซ์ (Multiplexed) การแยกสัญญาณแอดเดรสและข้อมูลมาใช้งานจะต้องต่อวงจรเพื่อทำการดีมัลติเพล็กซ์ภายนอก โดยสัญญาณควบคุมที่นำมาใช้ดีมัลติเพล็กซ์สัญญาณแอดเดรสและข้อมูลได้แก่ สัญญาณ STROBE B/R/$\overline{W}$ (STRB/R/$\overline{W}$) และสัญญาณ STROBE A/ADDRESS STROBE (STRA/AS) ในโหมดนี้ชิปจะสามารถติดต่อกับหน่วยความจำภายนอกเพิ่มได้สูงสุดถึง 64 กิโลไบต์ แต่ก็มีข้อเสียคือต้องใช้อุปกรณ์ต่อเพิ่มมากไม่ว่าจะเป็นหน่วยความจำ ชิปพอร์ทอินพุทเอาต์พุททำให้ค่าใช้จ่ายของระบบสูงขึ้น แต่อย่างไรก็ตามโหมดนี้เป็นโหมดที่นิยมใช้งานมากที่สุด เพราะสามารถต่อขยายหน่วยความจำและพอร์ทได้มากมาย

3. **โหมดบูตสแตร็ปพิเศษ** (Special Bootstrap Operating Mode)

คล้ายกับโหมดซิงเกิลชิปคือตัวไมโครคอนโทรลเลอร์สามารถทำงานได้โดยลำพัง แต่จะมีข้อพิเศษกว่าคือสามารถเขียนโปรแกรมแล้วโหลดเข้าไปในแรมภายในชิปได้เลย โปรแกรมที่ใช้ในการบูตจะถูกบรรจุ ไว้ในบูตสแตร็ปรอมจำนวน 192 ไบต์ รอมนี้จะสามารถเรียกข้อมูลออกมาใช้งานได้ก็ต่อเมื่อ 68HC11 ทำงานในโหมดบูตสแตร็ปนี้เท่านั้น โดยโปรแกรมบูตนี้จะเก็บไว้ที่แอดเดรส $BF40-$BFFF

เมื่อเรียกใช้งานโปรแกรมบูต ส่วน SCI (Serial Communication Interface) จะส่งข้อมูลโปรแกรมไปยังแรมภายในชิปที่แอดเดรส $0000-$00FF หลังจากที่ส่งข้อมูลที่แอดเดรส $00FF เรียบร้อยแล้ว ซีพียูจะเริ่มมาทำงานที่แอดเดรส $0000 ต่อไป ในโหมดนี้จะส่งผ่านข้อมูลผ่านพอร์ท SCI หลังจากที่ทำการรีเซตซีพียูแล้ว ส่วน SCI จะทำงานที่สัญญาณนาฬิกา E/16 (หรือมีอัตราบอดเท่ากับ 7, 812 สำหับสัญญาณนาฬิกาที่มีความถี่ 2 เมกะเฮิรตซ์) นอกจากนี้ยังมีลักษณะพิเศษประการหนึ่งคือ การป้องกันการคัดลอก โดยจะมีบิตป้องกัน (Security bit) เข้ามาเกี่ยวข้อง ถ้าหากบิตป้องกันนี้ถูกเซตเป็น " 1 " ที่เอาต์พุทของส่วน SCI จะส่งค่า $FF ออกไปทำให้ข้อมูลในอีอีพรอมถูกลบ และจะทำเช่นนี้จนกว่าข้อมูลในอีอีพรอมภายในชิปหมด ในทางตรงข้ามถ้าไม่มีการเซตบิตป้องกันข้อมูลที่เป็นการหยุดการทำงานวงจรชิปจะถูกส่งออกไปภายนอก โดยผ่านส่วน SCI

อินเตอร์รัปต์เวกเตอร์ของ 68HC11 เมื่อถูกใช้งานในโหมดนี้สามารถเข้าถึงหน่วยความจำแรมได้เร็วขึ้น การใช้อินเตอร์รัปต์จะทำให้ผู้ใช้งานสามารถกระโดดไปทำงานในส่วนอื่นได้ อินเตอร์รัปต์เวกเตอร์ที่มีใน 68HC11 สรุปได้ดังตาราง

ตารางที่ 3.1 อินเตอร์รัปต์เวกเตอร์ในโหมดการทำงานแบบบูตสแตร็ป

แอดเดรส	เวกเตอร์
$00C4	SCI
$00C7	SPI
$00CA	Pluse Accumulator Input Edge
$00CD	Pluse Accumulator Overflow
$00D0	Timer Overflow
$00D3	Timer Output Compare 5
$00D6	Timer Output Compare 4
$00D9	Timer Output Compare 3
$00DC	Timer Output Compare 2
$00DF	Timer Output Compare 1
$00E2	Timer Input Compare 3
$00E5	Timer Input Compare 2
$00E8	Timer Input Compare 1
$00EB	Real Time Interupt
$00EE	IRQ
$00F1	XIRQ
$00F4	SWI
$00F7	Illegal Opcode
$00FA	COP Fail
แอดเดรส	เวกเตอร์
$00FD	Clock Monitor
$BF40	Reset

ถ้าหากต้องการใช้งานอินเตอร์รัปต์ SWI ต้องระบุคำสั่งกระโดดเพื่อให้ซีพียูสามารถเรียกใช้งานได้ในแรมที่ตำแหน่ง $00F4, $00F5 และ $00F6 เมื่อเกิดอินเตอร์รัปต์ SWI ขึ้น ค่าเวกเตอร์นี้จะเป็นตัวบอกให้ซีพียูทำการอ่านข้อมูล เพื่อประมวลผลที่ $00F4 ในแรม และที่นั่นก็จะมีคำสั่ง JUMP เพื่อให้ซีพียูสามารถตอบสนองการอินเตอร์รัปต์ได้

อีกลักษณะพิเศษหนึ่งคือ สามารถที่จะเข้าไปทำงานที่รอมแอดเดรส $0000 ได้ โดยไม่ต้องทำการดาวน์โหลดก่อน วิธีการนี้จะใช้ได้กับเฉพาะเมื่อใช้สัญญาณนาฬิกา E/16 เริ่มด้วยการส่ง $55 ไปแทนค่า $FF ด้วยคำสั่งดังกล่าวนี้จะทำให้ซีพียูกระโดดไปทำงานที่แอดเดรส $0000 โดยไม่ต้องผ่านการดาวน์โหลดได้

4. **โหมดทดสอบพิเศษ** (Special test Operating Mode)

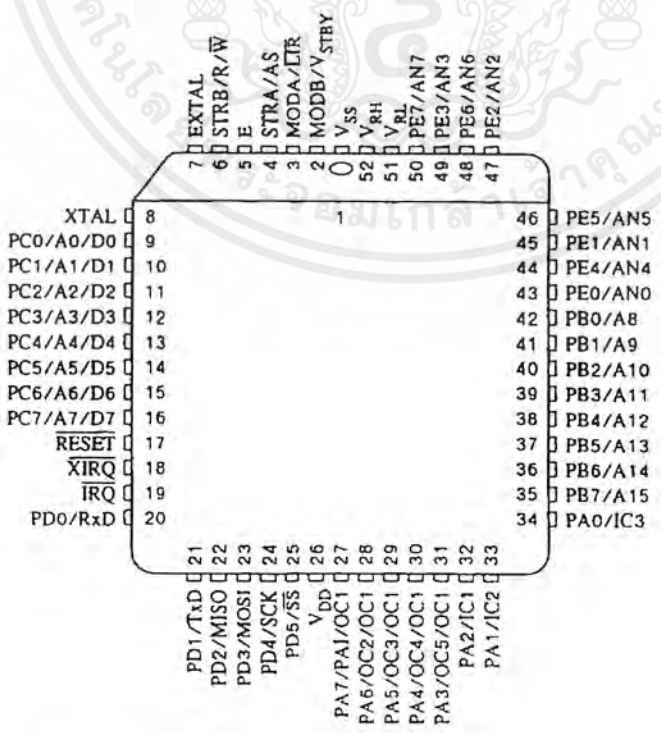
เป็นโหมดการทำงานที่โรงงานผู้ผลิตเป็นผู้กำหนด การทำงานในโหมดนี้จะคล้ายคลึงกับโหมดมัลติเพล็กซ์ขยาย การรีเซตและอินเตอร์รัปต์เวกเตอร์จะได้รับการเฟตช์จากหน่วยความจำภายนอกที่แอดเดรส $BFC0-$BFFF มากกว่าที่แอดเดรส $FFC0-$FFFF รีจิสเตอร์ TEST 1 จะได้รับการอินาเบิลเพื่อเป็นการบ่งบอกให้ทราบว่าขณะนี้ซีพียูพร้อมรับการตรวจสอบแล้ว โหมดการทำงานนี้เป็นโหมดที่ผู้ใช้งานไม่ควรใช้เพราะทำให้ระบบป้องกันข้อมูลภายในชิปด้อยลงไป

MODB	MODA	โหมดการทำงาน
1	0	ซิงเกิลชิป (Single chip)
1	1	มัลติเพล็กซ์ขยาย (Extended Multiplexed)
0	0	บูตสแตร็ปพิเศษ (special bootstrap)
0	1	ทดสอบพิเศษ (Special test)

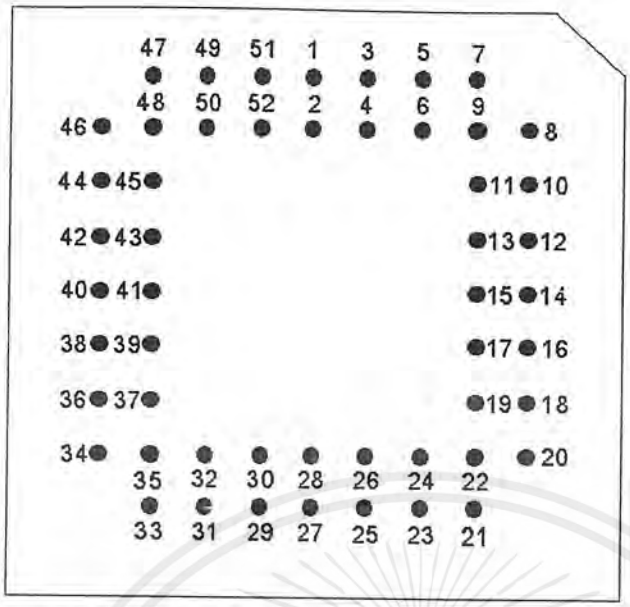
ตารางที่ 3.2 การเลือกโหมดการทำงานของ MC68HC11

3.3 รายละเอียดสัญญาณและการจัดขาของ 68HC11

ในรูปที่ 3.1 แสดงการจัดขาของ 68HC11 ซึ่งเป็นตัวถังแบบ PLCC ขนาด 52 ขา



รูปที่ 3.1 แสดงการจัดขาของ 68HC11 แบบ PLCC 52 ขา



Bottom View of 52-pin PLCC socket

รูปที่ 3.2 ภาพมองจากด้านล่างของซ็อกเก็ตแบบ PLCC 52 ขา

รายละเอียดของสัญญาณขาทั้งหมดมีดังนี้

ขาไฟเลี้ยงและกราวด์ (V_{DD}, V_{SS})

เป็นขาที่ใช้สำหรับจ่ายแรงดันเพื่อให้ 68HC11 ทำงานโดยขา V_{DD} ต้องป้อนแรงดัน +5 โวลต์ ในขณะที่ขา V_{SS} เป็นกราวด์ และเนื่องจาก 68HC11 มีโครงสร้างมาจากอุปกรณ์จำพวกซีมอส ดังนั้นจะต้องระมัดระวังอย่างมากในการจ่ายแรงดันไฟเลี้ยง แหล่งจ่ายไฟต้องมีคุณภาพค่อนข้างดี แรงดันคงที่และต้องต่อตัวเก็บประจุแบบเซรามิกค่า 0.1 ไมโครฟารัดคร่อมระหว่างขา V_{DD} และ V_{SS} และต้องติดตั้งตัวเก็บประจุนี้ให้ใกล้ขาใดขาหนึ่งมากที่สุด

ขา $\overline{RESET}$

เป็นขาอินพุทรับสัญญาณเพื่อให้ 68HC11 เริ่มต้นทำงานใหม่อันอาจเกิดจากสภาวะที่ซีพียูเพิ่งได้รับไฟเลี้ยงให้เริ่มทำงาน หรือเกิดจากการทำงานภายในวงจรเกิดความผิดพลาดแล้วถูกตรวจจับได้โดยวงจรวอตช์ด็อก จากนั้นวงจรวอตช์ด็อกก็จะป้อนสัญญาณมาที่ขานี้เพื่อกระตุ้นให้ซีพียูเริ่มทำงานใหม่หมดโดยสัญญาณที่ป้อนเข้าขานี้ต้องมีสภาวะลอจิก " 0 "

ขา XTAL, EXTAL

ทั้ง 2 ขานี้ถูกจัดไว้ให้ต่อกับคริสตอลหรือวงจรกำเนิดสัญญาณนาฬิกา เพื่อควบคุมวงจรกำเนิดสัญญาณนาฬิกาในชิป ความถี่ที่ปรากฏอยู่ที่ขานี้มีค่าสูงกว่าที่ขา E ที่อยู่ประมาณ 4 เท่า ถ้าหากใช้วงจรกำเนิดสัญญาณนาฬิกาภายนอก ขา XTAL ต้องปล่อยลอยไว้หรือตัวต้านทานค่าตั้งแต่

10 กิโลโอห์ม-100 กิโลโอห์มลงกราวด์แล้วป้อนสัญญาณนาฬิกาเข้าที่ขา EXTAL โดยความถี่ป้อนนี้ต้องสูงกว่าที่ขา E ประมาณ 4 เท่า ในการต่อวงจรเพื่อกำเนิดสัญญาณนาฬิกานั้นควรต่อบัฟเฟอร์ที่มีอิมพีแดนซ์สูง เช่น ใช้ไอซีเบอร์ 74HC04 ทั้งนี้ก็เผื่อไว้สำหรับในกรณีที่ต้องการต่อวงจรกำเนิดสัญญาณนาฬิกานี้จ่ายให้แก่ไมโครคอนโทรลเลอร์ตัวอื่นด้วย อย่างไรก็ดีถ้าหากต้องการต่อวงจรกำเนิดสัญญาณนาฬิกาเข้ากับไมโครคอนโทรลเลอร์เพียง 2 ตัว สามารถทำได้โดยต่อสัญญาณจากขา XTAL ผ่านตัวต้านทาน 220 โอห์ม เข้าที่ขา EXTAL ของไมโครคอนโทรลเลอร์อีกตัวหนึ่งก็ได้

ขา E

เป็นขาเอาต์พุทของวงจรกำเนิดสัญญาณนาฬิกาภายในชิป สามารถใช้เป็นฐานเวลาอ้างอิงได้โดยความถี่ที่ออกจากขานี้จะมีค่า 1/4 เท่าของความถี่อินพุทที่ขา XTAL และ EXTAL เมื่อใดที่ขา E นี้มีสภาวะลอจิกเป็น " 0 " หมายความว่า ขณะนี้ซีพียูกำลังทำการประมวลผลภายในอยู่ ถ้าหากเป็น " 1 " หมายถึง ขณะนี้ข้อมูลได้รับการเรียกใช้จากซีพียู ถ้าหากไมโครคอนโทรลเลอร์อยู่ในโหมด STOP สัญญาณที่ขา E จะไม่มีออกมาหรือเกิดสภาวะ High Impedance นั่นเอง

ขา $\overline{IRQ}$ (Interrupt Request)

เป็นขาอินพุทสำหรับเรียกการอินเตอร์รัปต์แบบอะซิงโครนัสของชิป 68HC11 โดยการรับสัญญาณเพื่ออินเตอร์รัปต์นี้สามารถจะเลือกได้ว่าจะรับสัญญาณขอบขาลง (Negative Edge) หรือจะรับการทริกเป็นแบบระดับสัญญาณ โดยปกติจะกำหนดให้รับการอินเตอร์รัปต์ด้วยการทริกแบบระบบสัญญาณ นั่นคือ ทำงานที่ระดับลอจิก " 0 " เมื่อต่อใช้งานปกติต้องต่อตัวต้านทานค่า 4.7 กิโลโอห์มพูลอัปเข้ากับไฟเลี้ยง

ขา $\overline{XIRQ}$ (Non-Maskable Interrupt)

เป็นขาอินพุทสำหรับตอบรับการอินเตอร์รัปต์แบบนอน-มาสเคเบิล หลังจากที่มีการรีเซตซีพียูสามารถรับการอินเตอร์รัปต์ด้วยการทริกแบบระดับสัญญาณ โดยทำงานที่ลอจิก " 0 " และต้องต่อตัวต้านทานพูลอัปกับไฟเลี้ยงเข้าที่ขา $\overline{XIRQ}$ นี้ด้วย ในขณะที่ไม่มีการส่งสัญญาณอินเตอร์รัปต์

ขา MODA/$\overline{LIR}$ และ MODB/V_{STBY} (Mode A/lode intruction register และ Mode B/Standby voltage)

หลังจากที่มีการรีเซตไมโครคอนโทรลเลอร์ขานี้ (ทั้ง MODA และ MODB) จะถูกใช้ให้เป็นขาสำหรับเลือกโหมดการทำงานของ 68HC11 ซึ่งเลือกได้เพียง 1 โหมดจาก 4 โหมด ที่ได้อธิบายไปแล้ว หลังจากเลือกโหมดการทำงานแล้ว ขา MODA จะเปลี่ยนลักษณะการทำงาน เป็นขา $\overline{LIR}$

แทนซึ่งจะมีลักษณะเป็นขาเอาต์พุทใช้ในการแสดงสภาวะว่าขณะนี้ได้เริ่มจัดการกับคำสั่งแล้ว โดยจะแอกทีฟเป็นลอจิก " 0 " ในทันทีที่ขา E เริ่มส่งสัญญาณนาฬิกาพัลส์แรกของแต่ละคำสั่งออกมา ซึ่งก็คือ เมื่อซีพียูเริ่มเฟตช์ข้อมูลคำสั่ง ขา $\overline{LIR}$ นี้จะแอกทีฟทันที สัญญาณที่ขา $\overline{LIR}$ จะมีไว้ช่วยในการแก้ไขโปรแกรม (Program debugging) เช่นเดียวกับขา MODA เมื่อขา MODB ถูกใช้ในการเลือกโหมดการทำงานไปแล้ว ที่ขา MODB นี้จะเปลี่ยนหน้าที่เป็นขาอินพุท V_{STBY} แทน โดยอินพุท V_{STBY} นี้เป็นอินพุท สำหรับจ่ายแรงดันเพื่อสแตนด์บายให้แก่แรมภายในชิป ถ้าหากแรงดันที่ขานี้มากกว่าแรงดันไฟเลี้ยงชิปประมาณ 0.7 โวลต์แล้ว หน่วยความจำแรมภายในชิปทั้ง 256 ไบต์ และส่วนลอจิกรีเซตจะรับแรงดันจากขานี้แทนแรงดันที่จ่ายเข้าที่ขา V_{DD} นั่นคือใช้เป็นขาที่จ่ายแรงดันเพื่อเลี้ยงหน่วยความจำแรม เพื่อป้องกันข้อมูลภายในชิปสูญหาย เมื่อไม่มีการจ่ายแรงดันไฟเลี้ยงปกติ สำหรับส่วนลอจิกรีเซตจะต้องกำหนดให้มีสภาวะ " 0 " ก่อนที่จะปลดแรงดันออกจากขา V_{DD} และต้องรักษาสภาวะลอจิก " 0 " นี้ไปจนกว่าที่ขา V_{DD} จะได้รับแรงดันไฟเลี้ยงในระดับปกติ

ขา V_{RL} และ V_{RH} (A/D Converter Reference Voltages)

เป็นขาที่ใช้สำหรับต่อแรงดันอ้างอิงสำหรับวงจรแปลงสัญญาณแอนาล็อกเป็นดิจิตอลภายในชิป

ขา STRB/R/$\overline{W}$ (Strobe B และ Read/Write)

เป็นขาที่สามารถทำงานได้หลายหน้าที่ขึ้นอยู่กับโหมดการทำงานของไมโครคอนโทรลเลอร์ ถ้าหาก 68HC11 อยู่ในโหมดซิงเกิลชิปขานี้จะเป็นขาเอาต์พุท STRB หรือเรียกว่า ขาสโตรปสำหรับการแฮนด์เชกกับอุปกรณ์ภายนอกที่ต่อเข้าที่ขาอินพุทเอาต์พุท แต่ถ้าอยู่ในโหมดมัลติเพล็กขยายขานี้จะแสดงตนเป็นขา R/$\overline{W}$ ใช้ในการควบคุมทิศทางของการถ่ายเทข้อมูลกับบัสข้อมูลภายนอกชิป ถ้าขานี้เป็นลอจิก " 0 " จะหมายความว่า ขณะนี้ซีพียูกำลังทำการเขียนข้อมูลลงไปในบัสข้อมูลภายนอก แต่ถ้าเป็นลอจิก " 1 " จะเป็นการแสดงว่า ขณะนี้กำลังอ่านข้อมูลเข้ามาเพื่อทำการประมวลผล ถ้าหากมีการเขียนข้อมูลอย่างต่อเนื่อง ขา R/$\overline{W}$ ก็จะเป็นลอจิก " 0 " ตลอดเวลา จนกว่าจะทำการเขียนข้อมูลเสร็จ นอกจากนี้เมื่อใช้งานขา R/$\overline{W}$ ร่วมกับขา E จะสามารถใช้เป็นสัญญาณอีนาเบิลสำหรับหน่วยความจำสแตติกแรมภายนอก

ขา STRA/AS (Strobe A และ Address Strobe)

เป็นขาอินพุทโดยรับสัญญาณเป็นแบบขอบขาของสัญญาณ ลักษณะการใช้งานขานี้จะขึ้นอยู่กับการกำหนดโหมดการทำงานของ 68HC11 สำหรับโหมดซิงเกิลชิปขานี้จะเป็นขา STRA คือเป็นขาสโตรปสำหรับการแฮนค์เชกกับอุปกรณ์ภายนอกที่ต่อเข้ากับพอร์ทขนานอินพุทเอาต์พุท ใน

โหมดมัลติเพล็กซ์ขยายขานี้จะเปลี่ยนลักษณะการทำงานเป็นขาเอาต์พุท AS โดยใช้ในวงจรดีมัลติเพล็กซ์สัญญาณข้อมูลกับสัญญาณแอดเดรสที่พอร์ท C

ขาสัญญาณของพอร์ท

พอร์ท A, D และ E จะเป็นพอร์ทที่ไม่ขึ้นกับโหมดการทำงานของ 68HC11 นั่นคือ ไม่ว่า 68HC11 จะทำงานในโหมดใดก็ตาม พอร์ททั้ง 3 ยังคงมีลักษณะสัญญาณและการทำงานเช่นเดิม ในขณะที่พอร์ท B จะถูกกำหนดให้เป็นพอร์ทเอาต์พุท หาก 68HC11 ทำงานในโหมดซิงเกิลชิป แต่ถ้าเป็นโหมดมัลติเพล็กซ์ขยายจะกลายเป็นขาแอดเดรสไบต์สูง ส่วนพอร์ท C จะเป็นพอร์ทอินพุทเอาต์พุท ถ้า 68HC11 ทำงานในโหมดซิงเกิลชิป ถ้าหากทำงานในโหมดมัลติเพล็กซ์ขยาย พอร์ท C นี้จะใช้เป็นบัสมัลติเพล็กซ์ระหว่างแอดเดรส 8 บิตต่ำกับบัสข้อมูล

ตารางที่ 3.3 สรุปลักษณะหน้าที่ของขาสัญญาณของ 68HC11

พอร์ท-บิต	เมื่อทำงานในโหมดซิงเกิลชิปและบูตสแตร็ป	เมื่อทำงานในโหมดมัลติเพล็กซ์ขยายและทดสอบพิเศษ
A-0		PA0/IC3
A-1		PA1/IC2
A-2		PA2/IC1
A-3		PA3/OC5/และ-หรือ OC1
A-4		PA4/OC4/และ-หรือ OC1
A-5		PA5/OC3/และ-หรือ OC1
A-6		PA6/OC2/และ-หรือ OC1
A-7		PA7/PAI/และ-หรือ OC1
พอร์ท-บิต	เมื่อทำงานในโหมดซิงเกิลชิปและบูตสแตร็ป	เมื่อทำงานในโหมดมัลติเพล็กซ์ขยายและทดสอบพิเศษ
B-0	PB0	A8
B-1	PB1	A9
B-2	PB2	A10
B-3	PB3	A11
B-4	PB4	A12
B-5	PB5	A13
B-6	PB6	A14
B-7	PB7	A15

C-0	PC0	A0/D0
C-1	PC1	A1/D1
C-2	PC2	A2/D2
C-3	PC3	A3/D3
C-4	PC4	A4/D4
C-5	PC5	A5/D5
C-6	PC6	A6/D6
C-7	PC7	A7/D7
D-0	PD0/RxD	PD0/RxD
D-1	PD1/TxD	PD1/TxD
D-2	PD2/MISO	PD2/MISO
D-3	PD3/MOSI	PD3/MOSI
D-4	PD4/SCK	PD4/SCK
D-5	PD5/$\overline{ss}$	PD5/$\overline{ss}$
	STRA	AS
	STRB	R/$\overline{w}$
E-0	PE0/AN0	PE0/AN0
E-1	PE1/AN1	PE1/AN1
E-2	PE3/AN2	PE2AN2
E-3	PE3/AN3	PE3/AN3
E-4	PE4/AN4# #	PE4/AN4##
E-5	PE5/AN5# #	PE5/AN5##
E-6	PE6/AN6# #	PE6/AN6##
E-7	PE7/AN7# #	PE7/AN7##

ไม่มีในตัวถังแบบ 48 ขา

พอร์ท A (PA0-PA7)

สังเกตจากตารางที่ 3.2 จะเห็นว่า พอร์ท A นี้แต่ละสัญญาณสามารถทำงานได้ตั้งแต่ 2-3 ฟังก์ชัน โดยสามารถเป็นได้ทั้งอินพุทแคปเจอร์ (Input Capture) 3 อินพุท คือ IC1, IC2 และ IC3 เป็นเอาต์พุทของฟังก์ชันเปรียบเทียบ (Output Compare) 4 ช่อง คือ OC2, OC3, OC4 และ OC5 หรือจะเป็นเอาต์พุทของฟังก์ชันเปรียบเทียบลำดับที่ 5 (OC1)

พอร์ท B (PB0-PB7)

เมื่อ 68HC11 ทำงานในโหมดซิงเกิลชิป ขาสัญญาณทั้งหมดจะเป็นขาเอาต์พุทของพอร์ทเอาต์พุท และใช้งานร่วมกับสัญญาณสโตรบเอาต์พุท จะมีพัลส์เอาต์พุทมาปรากฏที่ขา STRB ในทุกๆช่วงเวลาที่มีการเขียนข้อมูลออกมาที่พอร์ท B

ถ้าหากทำงานในโหมดมัลติเพล็กซ์ขยาย ขาสัญญาณทั้งหมดของพอร์ท B จะทำหน้าที่เป็นขาแอดเดรส 8 บิตสูง (A8-A15)

พอร์ท C (PC0-PC7)

ถ้า 68HC11 ทำงานในโหมดซิงเกิลชิป ขาของพอร์ท C ทั้งหมดจะเป็นขาพอร์ทอินพุทเอาต์พุท ถ้าเป็นพอร์ทอินพุท พอร์ท C สามารถที่แลตช์สัญญาณอินพุทที่ป้อนเข้ามานี้ได้ โดยการป้อนสัญญาณเข้าที่ขา STRA สัญญาณพอร์ท C มักใช้ในการแฮนด์เชกของการเชื่อมต่อพอร์ทขนาน โดยในขณะเดียวกันขา STRB จะต้องถูกใช้เป็นสายควบคุมการแฮนด์เชก

แต่การทำงานในโหมดมัลติเพล็กซ์ขยาย ขาสัญญาณพอร์ท C จะสามารถเป็นได้ทั้งขาของแอดเดรสและข้อมูลโดยการมัลติเพล็กซ์ ปกติขานี้จะเป็นสัญญาณแอดเดรส A0-A7 ในแต่ละไซเคิลการทำงานของไมโครคอนโทรลเลอร์ แต่ถ้าหากขา E แอกทีฟเป็น " 1 " สัญญาณที่ขานี้จะกลับเป็นขาสัญญาณข้อมูล D0-D7 โดยทิศทางการเข้าออกของข้อมูลที่พอร์ท C จะแสดงออกมาที่ขาสัญญาณ R/$\overline{w}$

พอร์ท D (PD0-PD5)

ขาสัญญาณพอร์ท D ทั้ง 6 ขา ปกติจะใช้เป็นขาสัญญาณอินพุทเอาต์พุท และสามารถรองรับในการใช้งานเชื่อมต่อสื่อสารข้อมูลอนุกรม (SCI) และเชื่อมต่ออุปกรณ์อนุกรม (SCI) ด้วย ถ้าหากมีการอีนาเบิลในส่วนนี้ให้ทำงาน

ขา PD0 เป็นขาอินพุทรับข้อมูล (RxD) เมื่อใช้ในการเชื่อมต่อสื่อสารข้อมูลอนุกรม (SCI)

ขา PD1 เป็นขาเอาต์พุทส่งข้อมูล (TxD) เมื่อใช้ในการเชื่อมต่อสื่อสารข้อมูลอนุกรม (SCI)

ขา PD2-PD5 ใช้สำหรับเชื่อมต่ออุปกรณ์อนุกรม (SCI) โดย PD2 เป็นขาสัญญาณมาสเตอร์อินสลาฟเอาต์ (Master In Slave Out: MISO) PD3 เป็นขาสัญญาณมาสเตอร์เอาต์สลาฟอิน (Master Out Slave In: MOSI)

PD4 เป็นขาสัญญาณนาฬิกาอนุกรม (SCK) และขา PD5 เป็นขาอินพุทเลือกสลาฟ ($\overline{ss}$)

พอร์ท E (PE0-PE7)

พอร์ท E เป็นพอร์ทอินพุทและเป็นขาอินพุทสำหรับวงจรแปลงสัญญาณแอนาล็อกเป็นดิจิตอล ใน 68HC11 แบบ 48 ขา พอร์ท E จะมีเพียง 4 ขา แต่ถ้าเป็นแบบ 52 ขา จะมีครบ 8 ขา

สัญญาณอินพุทที่จะป้อนเข้ามาที่พอร์ท E นี้ ต้องมีความแน่นอนของสัญญาณสูงพอสมควร เพราะความแม่นยำของการแปลงสัญญาณแอนาล็อกเป็นดิจิตอลจะขึ้นอยู่กับความเที่ยงตรงของสัญญาณอินพุทเป็นหลัก

รีจิสเตอร์ของซีพียู

ซีพียูของไมโครคอนโทรลเลอร์ 68HC11 จะมีรีจิสเตอร์ใช้งานอยู่ 7 ตัว ได้แก่

1. แอกคิวมูเลเตอร์ A และ B

   เป็นรีจิสเตอร์ขนาด 8 บิต ที่ใช้ในการเก็บค่าโอเปอแรนด์และผลของการคำนวณทางคณิตศาสตร์หรือผลจากการจัดการข้อมูลโดยตัวซีพียู ในการประมวลผลทางคณิตศาสตร์หรือลอจิกจะต้องนำข้อมูลเหล่านั้นมาเก็บในรีจิสเตอร์ทั้งสองตัวนี้ จึงจะสามารถประมวลผลได้

2. แอกคิวมูเลเตอร์ D

   เป็นรีจิสเตอร์ขนาด 16 บิต ใช้ในการประมวลผลและเก็บค่าจากการคำนวณทางคณิตศาสตร์และลอจิก แอกคิวมูเลเตอร์ D ก็เกิดมาจากการรวมกันของแอกคิวมูเลเตอร์ A และ B จึงทำให้มีขนาด 16 บิต

3. รีจิสเตอร์อินเด็กซ์ IX

   เป็นรีจิสเตอร์ขนาด 16บิต ใช้ในการชี้ตำแหน่งแอดเดรส เพื่อเข้าไปจัดการประมวลผลกับข้อมูลในแอดเดรสนั้นๆ นอกจากนั้น IX สามารถใช้เป็นตัวนับหรือรีจิสเตอร์เก็บข้อมูลชั่วคราวได้

4. รีจิสเตอร์อินเด็กซ์ IY

   เป็นรีจิสเตอร์ขนาด 16บิตมีหน้าที่เหมือนกับ IX แต่แตกต่างกันที่ในทุกคำสั่งที่ต้องเกี่ยวข้องกับ IY จะต้องมีข้อมูลไบต์พิเศษของรหัสแมชชีน และมีไซเกิลพิเศษของช่วงเวลาในการเอ็กซีคิวต์คำสั่งด้วย จึงทำให้คำสั่งที่มี IY ไปเกี่ยวข้องต้องมีขนาดเพิ่มขึ้นอย่างน้อย 2 ไบต์ขึ้นไปหรือที่เรียกว่า พรีไบต์ (Prebyte)

5. รีจิสเตอร์ตัวชี้สแต็ก (Stack Pointer : SP)

   เป็นรีจิสเตอร์ขนาด 16บิต ใช้เก็บแอดเดรสบนสแต็ก โดยสแต็กใน 68HC11 จะมีลักษณะการเก็บข้อมูลเข้าและนำข้อมูลออกมาเป็นแบบ LIFO (Last-In-First-Out) หรือข้อมูลที่เข้าไปเก็บในสแต็กหลังสุด เมื่อจะเรียกออกมาจะถูกเรียกออกมาก่อน สแต็กใช้เก็บข้อมูลของรีจิสเตอร์ เมื่อต้องมีการนำรีจิสเตอร์ตัวนั้นไปทำงานในโปรแกรมย่อยอื่น หรือต้องไปใช้ในการตอบสนองการอินเตอร์รับต์ เพื่อเป็นการป้องกันข้อมูลเดิมสูญหายจึงต้องเก็บข้อมูลนั้นไว้ในหน่วยความจำสำรองซึ่งก็คือสแต็กนั่นเอง

6. รีจิสเตอร์โปรแกรมเคาน์เตอร์ (Program Pounter : PC)

เป็นรีจิสเตอร์ขนาด 16บิต ใช้เก็บค่าของแอดเดรสของคำสั่งถัดไปที่ซีพียูจะไปทำการเอ็กซีคิวต์

7. รีจิสเตอร์รหัสเงื่อนไข (Condition Code Register : CCR)

เป็นรีจิสเตอร์ขนาด 8 บิต ในแต่ละบิตจะแสดงความหมายของผลจากการกระทำคำสั่งที่เพิ่งจะเอ็กซีคิวต์ไปของซีพียู แต่ละบิตของ CCR เป็นอิสระต่อกันจึงสามารถตรวจสอบสถานะได้โดยใช้โปรแกรม และยังสามารถนำผลการตรวจสอบนั้นไปดำเนินการต่อได้ด้วย

3.4 วงจรแปลงสัญญาณอานาลอกเป็นดิจิตอล

ไมโครคอนโทรลเลอร์ 68HC11 มีวงจรแปลงสัญญาณอนาลอกเป็นดิจิตอล (ADC) อยู่ภายในชิปขนาด 8 บิตจำนวน 8 ช่อง โดยใช้ขาสัญญาณพอร์ท E (PE1-PE7) เป็นขาอินพุทของสัญญาณแอนาล็อก วงจร ADC ใน 68HC11 นั้นเป็น ADC แบบซักเซสซีฟแอปพร็อกซิเมชัน (Successive Approximation) ซึ่งมีความสามารถในการทำงานสูง โดยสามารถสุ่มค่าและเก็บรักษาค่าไว้ได้ ถึงแม้ว่าแรงดันอินพุทจะมีการเปลี่ยนแปลงอย่างรวดเร็วก็ตาม มีค่าความผิดพลาดต่ำ

วงจร ADC ใน 68HC11 ต้องทำงานร่วมกับขา V_{RH} และ V_{RL} โดยขาทั้งสองจะเป็นขาที่ใช้ป้อนแรงดันอ้างอิง เพื่อใช้ในวงจร ADC แรงดันที่ป้อนให้แก่ขา V_{RH} และ V_{RL} สามารถจะใช้แหล่งจ่ายไฟหลักร่วมกับ 68HC11 ได้ หรือจะแยกใช้แหล่งจ่ายอิสระก็จะยิ่งดีเพราะจะทำให้ความแม่นยำในการแปลงสัญญาณเพิ่มขึ้นมาก

ค่าความผิดพลาดของวงจร ADC ใน 68HC11 มีค่าประมาณ $\pm$ 1 LSB อันประกอบด้วยค่าความผิดพลาดจากการควอนไตซิง (Quantizing) $\pm$ 0.5 LSB และความผิดพลาดจากย่านของแรงดันอ้างอิงคือตั้งแต่ระดับ V_{RH} ถึง V_{RL}

ย่านของแรงดันอ้างอิงสามารถกำหนดปรับเปลี่ยนได้โดยกำหนดแรงดันที่ขา V_{RH} เป็นแรงดันอ้างอิงสูงสุด และแรงดันที่ขา V_{RL} เป็นแรงดันอ้างอิงต่ำสุด สามารถกำหนดได้สูงสุดคือ V_{RH} เท่ากับ 5 โวลต์ $\pm$ 10 เปอร์เซ็นต์ และ V_{RL} เป็น 0 โวลต์อย่างไรก็ตามจากการทดสอบพบว่าที่ประสิทธิภาพการทำงานได้ดีที่สุดของวงจรย่านแรงดันอ้างอิงไม่ควรต่ำกว่า 2.5-3 โวลต์ ($V_{RH} - V_{RL} \geq$ 2.5-3.0 โวลต์) นั่นคือ แรงดัน V_{RH} สามารถจะต่ำกว่าแรงดัน V_{DD} (แรงดันไฟเลี้ยงของไมโครคอนโทรลเลอร์) และแรงดัน V_{RL} สามารถสูงกว่า 0 โวลต์ได้

ในการแปลงสัญญาณแต่ละช่องจะใช้เวลา 32 ไซเคิลของสัญญาณนาฬิกา E ดังนั้นความถี่ของสัญญาณนาฬิกา E จะต้องสูงกว่า 750 กิโลเฮิรตซ์ แต่ถ้าหากระบบใดมีความถี่ของสัญญาณนาฬิกาต่ำกว่า 750 กิโลเฮิรตซ์ วงจรออสซิลเลเตอร์ R-C ภายในชิปจะเป็นผู้ส่งสัญญาณ

นาฬิกาให้แก่วงจร ADC แทน ซึ่งวงจรออสซิลเลเตอร์สามารถอินาเบิลให้ทำงานได้โดยเซตบิต CSEL ในรีจิสเตอร์ OPTION

3.5 อินเตอร์เลคทีฟ ซี (IC)

อินเตอร์เลคทีฟ ซี (IC) เป็นภาษา C ซึ่งประกอบไปด้วยตัว compile ที่สามารถใช้กับไมโครคอนโทรลเลอร์ MC68HC11 ได้เป็นอย่างดีสามารถหา Download ได้ที่ website ftp://cherupakha.media.mit.edu/pub/projects/interactive-c/ อินเตอร์เลคทีฟซีนี้มีโครงสร้างในการเขียนโปรแกรมเหมือนภาษาซีทั่วๆไป (for, while, if, else), ตัวแปรแบบ local global, arrays, pointer, structures และ integer แบบ 16-bit และ 32-bit, และจำนวนแบบ 32-bit floating point ซึ่ง IC จะทำงานโดยการ compile ไปเป็นรูปโค้ดแฝง (pseudo-code หรือ p-code) และจะถูกตีความหมายด้วยโปรแกรมภาษา run-time machine การใช้งานอินเตอร์เลคทีฟซี สามารถอ่านได้จากภาคผนวก ข.

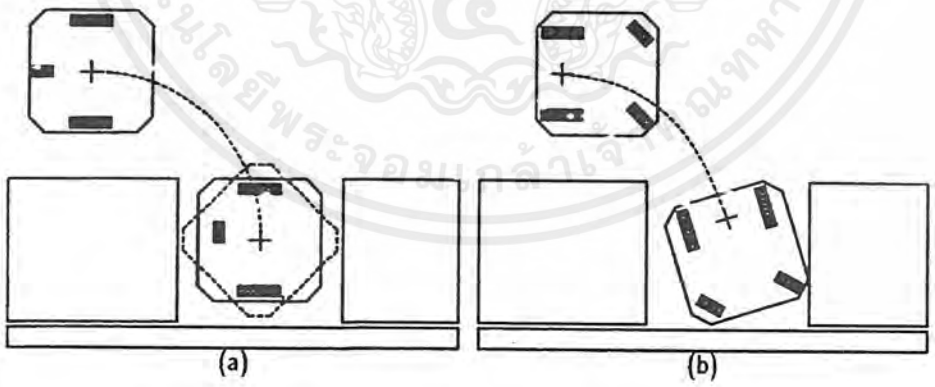
บทที่ 4
โครงสร้างของหุ่นยนต์

4.1 โครงสร้างของหุ่นยนต์

- ลักษณะรูปแบบของหุ่นยนต์

ลักษณะของหุ่นยนต์ที่ทำการสร้างขึ้นนี้ รูปทรงจะมีส่วนสำคัญในการเคลื่อนที่ได้อย่างสะดวกกว่าแบบรูปทรงสี่เหลี่ยม โดยเมื่อพิจารณาการเคลื่อนที่ดังรูปที่ 4.1 (ข) จะสังเกตเห็นได้ว่าการเคลื่อนของหุ่นยนต์รูปทรงสี่เหลี่ยมจะเกิดการชนขณะทำการเลี้ยว หากเป็นรูปที่ 4.1 (ก) จะง่ายต่อการเลี้ยวไม่เกิดการชนทำให้สามารถเคลื่อนที่ไปได้

รูปที่ 4.1 รูปทรงของหุ่นยนต์มีผลต่อการเคลื่อนที่



รูปที่ 4.2 รูปแบบการขับเคลื่อนล้อ

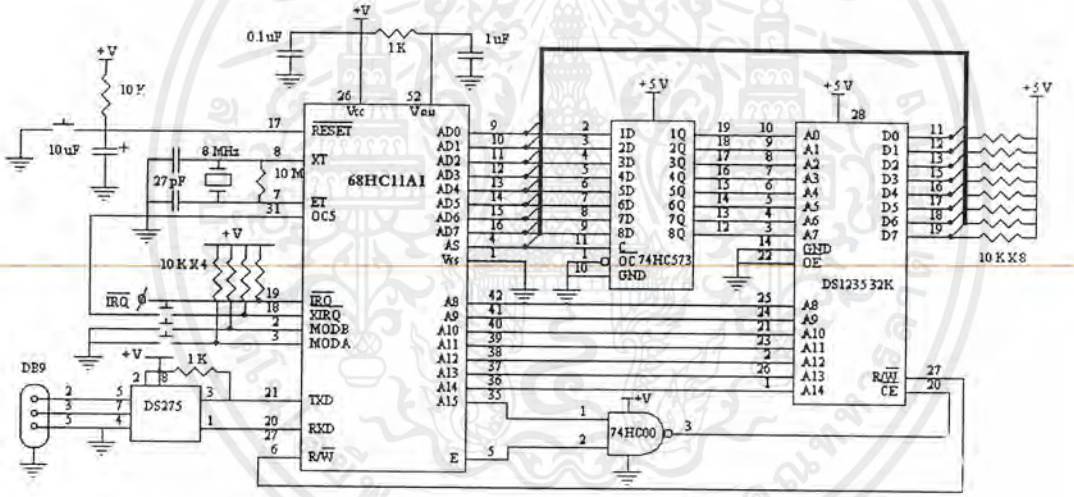
จากรูปที่ 4.2 รูปแบบการขับเคลื่อนล้อจะมีผลต่อการเคลื่อนในที่แคบมีสิ่งกีดขวางเป็นอุปสรรค รูปที่ 4.2(a) รูปแบบจะใช้ 2 ล้อคู่มีความเร็วไม่เท่ากันซึ่งสามารถเลี้ยวได้เร็ว อีก 1 ล้อจะเป็นล้อฟรีทำการพยุงหุ่นยนต์ ซึ่งเป็นแบบสามารถเคลื่อนที่สะดวกในที่แคบมีสิ่งกีดขวาง

ส่วนรูปที่ 4.2 (b) เมื่อจะใช้การขับเคลื่อน 2 ล้อที่มีความเร็วทำกัน และมี Servo motor เป็นตัวควบคุมการเลี้ยวซึ่งทำให้การเลี้ยวช้าและเมื่อเข้าสู่ที่แคบมีสิ่งกีดขวางจะไม่สามารถเคลื่อนที่ได้

4.2 ส่วนประกอบของบอร์ดควบคุม

จากรูปที่ 4.2 เป็นวงจรแสดงวงจรบอดร์ดควบคุมหลักของหุ่นยนต์ ซึ่งประกอบด้วยส่วนต่างๆดังนี้

1. ซีพียู MC68HC11
2. NVRAM 32 KBYTE DS1235K
3. ไอซี 74HC373
4. ไอซี 74HC00
5. DS275

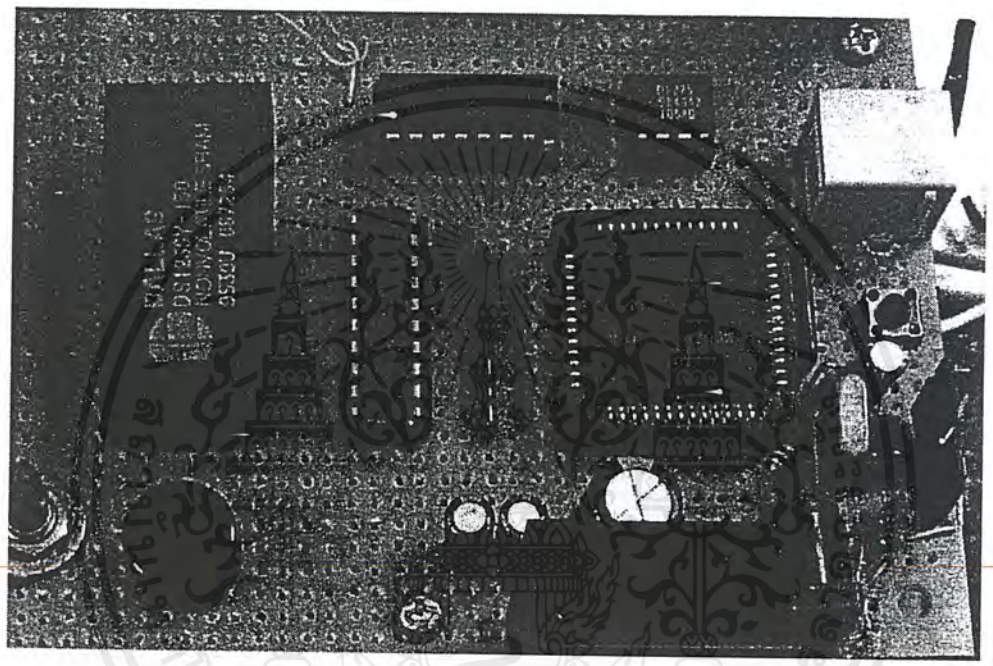


รูปที่ 4.3 วงจรบอร์ดควบคุมหลัก

จากรูปที่ 4.3 หน่วยความจำ ภายในชิปที่มีไม่เพียงพอ ดังนั้นเราจึงทำการต่อหน่วยความจำภายนอกเพิ่มเติม จากคุณสมบัติของ MC68HC11 เราจะทำหน่วยความจำภายนอกได้โดยใช้โหมดมัลติเพล็กซ์ขยาย โดยให้ขา MODA และ MODB ต่อเข้า Jumper โดยให้ขาทั้งสองต่อลงกราวด์ ขาของพอร์ท B จะกำหนดเป็นแอดเดรสไบต์สูงคือเชื่อมต่อเข้ากับ DS1235 ที่ขา A8-A15 ในขณะที่ขาพอร์ต C ถูกกำหนดให้เป็นแอดเดรสไบต์ต่ำ A0-A7 โดยมี 74HC373 เป็นไอซีทำหน้าที่ดีมัลติเพล็กซ์สัญญาณแอดเดรสไบต์ต่ำและสัญญาณข้อมูล โดยทำงานร่วมกับสัญญาณ AS จาก

MC68HC11 ถ้าหากเป็นการส่งค่าแอดเดรส สัญญาณที่ขา AS จะแอกทีฟทำให้อีนาเบิล 74HC373 ข้อมูลจากพอร์ต C ก็จะถูกส่งเข้าไปที่ขาข้อมูล D0-D7 ของแรม สำหรับ NAND GATE ที่ต่อเข้ากับขา E และขา A15 จะทำการกำหนดให้ DS1235 อีนาเบิลทำให้แรมอ่านและเขียนข้อมูลลงบนแรม

ในส่วนของการเชื่อมต่อการสื่อสารข้อมูลทางพอร์ตอนุกรมกับคอมพิวเตอร์ จะใช้ DS275 Line-Power RS232 Transceiver Chip ของ Dallas โดยต่อวงจรดังรูปที่ 4.2

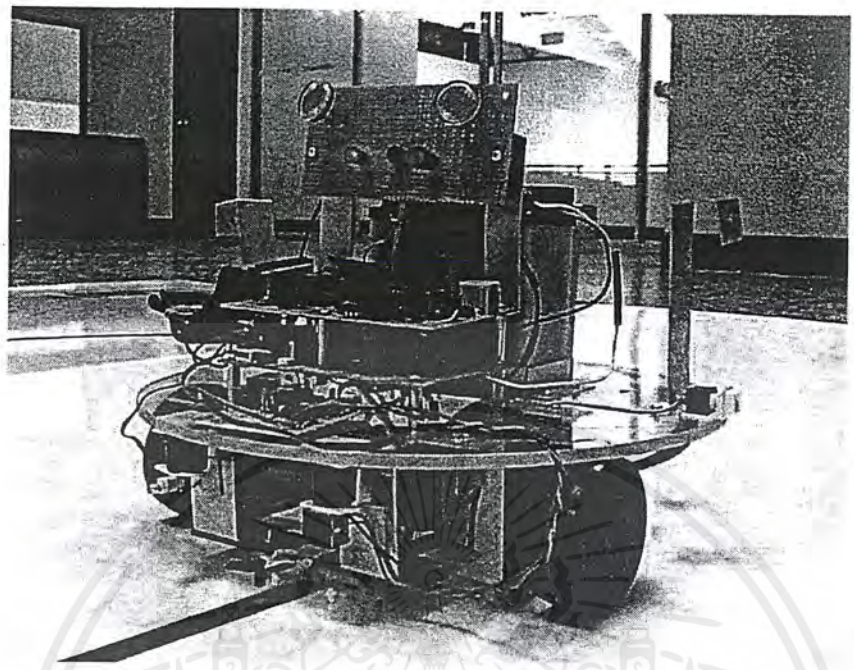


รูป 4.4 แสดงวงจรบอร์ดควบคุม

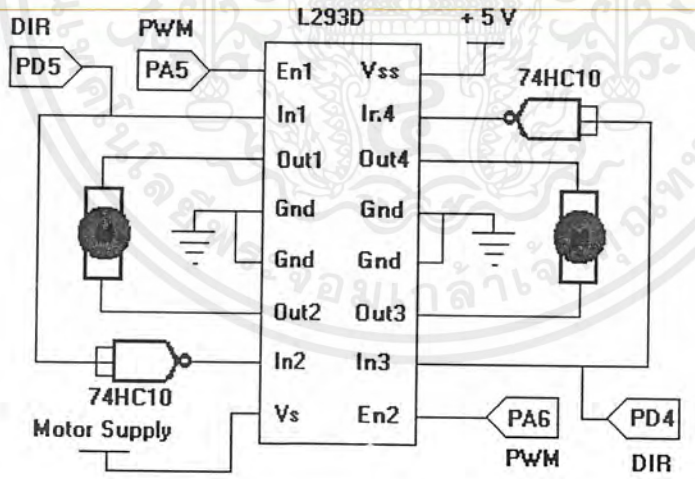
4.3 ส่วนประกอบของบอร์ดเซนเซอร์

ส่วนประกอบของบอร์ดเซนเซอร์จะประกอบไปด้วย

- เซนเซอร์กระแทก
- เซนเซอร์อินฟราเรด
- เซนเซอร์ไมโครโฟน
- เซนเซอร์แสง LDR
- วงจรขับมอเตอร์กระแสตรง



รูปที่ 4.5 รูปแสดงบอร์ดเซนเซอร์



รูปที่ 4.6 วงจรขับมอเตอร์

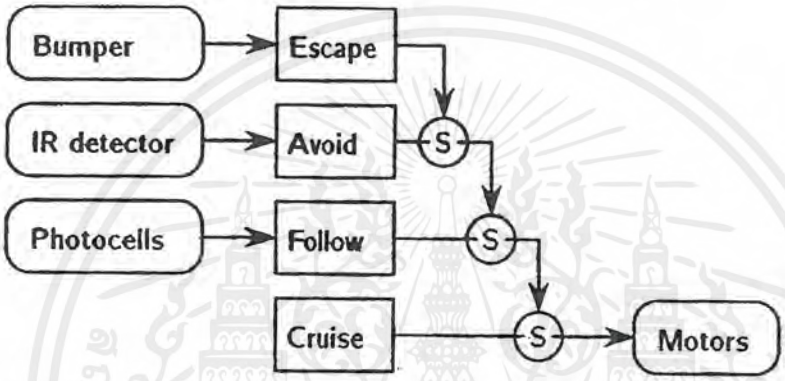
4.4 การทำงานของหุ่นยนต์

ในการทำงานของหุ่นยนต์จะทำงานแบบ Multi Tasking กล่าวคือสามารถทำงานทำงานแต่ละอย่างในระยะเวลาอันสั้นทำให้การทำงานหลายๆอย่างในเวลาเดียวกันมีความต่อเนื่องเช่นในขณะให้ LED ทำการกระพริบทุกๆ 1 วินาที เราสามารถทำงานอย่างอื่นพร้อมกันไปด้วย เช่น ขับ

เคลื่อนมอเตอร์หรือตรวจวัดแสงเป็นต้นในส่วนของเซนเซอร์ตรวจจับซึ่งหุ่นยนต์สามารถทำงานได้จะประกอบด้วย

- ตรวจจับสิ่งกีดขวาง (IR)
- ตรวจจับการชน (Bumper)
- ตรวจจับแสงซึ่งใช้โฟโต้รีซิสเตอร์
- ตรวจจับสัญญาณเสียง ๆ(Microphone)

ลักษณะการทำงานดังรูปข้างล่าง



รูปที่ 4.7 การทำงานของหุ่นยนต์

จากรูปที่ 4.7 การทำงานของหุ่นยนต์จะเริ่มทำ Function Cruise ทุกครั้งกล่าวคือหุ่นยนต์จะวิ่งตรงไปเรื่อยในขณะไม่มีเหตุการณ์อื่นมาอินเตอร์รัปต์ เมื่อมีการวัตถุมากระแทกชนกับ Bumper จะเกิดการอินเตอร์รัปต์ขึ้นให้มอเตอร์หมุนหนีวัตถุ หาก IR ตรวจจับวัตถุสิ่งกีดขวางได้ก็จะส่งสัญญาณควบคุมมอเตอร์ให้หลีกหนีวัตถุ Photocells ก็เช่นเดียวกัน เมื่อให้ค้นหาเจอแสง ก็จะทำการควบคุมมอเตอร์ให้เดินหน้า โดยเราจะให้ความสำคัญของแต่ละเหตุการณ์แตกต่างกันทำให้การทำงานที่เกิดขึ้นพร้อมกันจะทำเหตุการณ์ที่สำคัญก่อน

บทที่ 5
การทดลอง

การทดลองการใช้งานโดยทดสอบโปรแกรมควบคุมเซนเซอร์แต่ละอย่างหลังจากนั้นเราจะทำการรวมโปรแกรมทั้งหมดเข้าด้วยกันโดยทำการเขียนโปรแกรมให้สอดคล้องกัน ดังนั้นการทดลองเราจะทำการทดลองทั้งสิ้น 4 การทดลองดังนี้

การทดลองที่ 1 วิ่งตามแถบเส้นสีดำ

อุปกรณ์

1.LED Super Byte สีแดง 2 ตัว

2. โฟโต้รีซิสเตอร์ 2 ตัว

3. ตัวต้านทาน 220 โอมห์ 2 ตัว

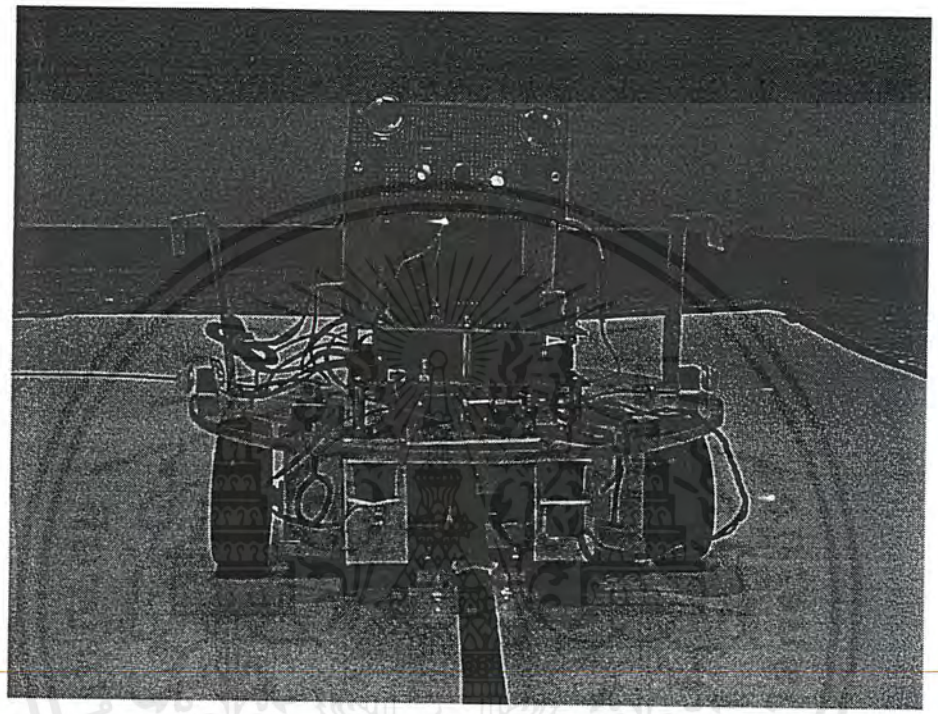
หลักการทำงาน

สีดำกับสีขาวจะมีค่าการสะท้อนของแสงต่างกันเมื่อกระทบพื้น ดังนั้นเมื่อแสงจาก LED Super Byte ตกกระทบพื้นจะมีการสะท้อน เราจะทำการตรวจสอบค่าแรงดันที่วัดได้ผ่านทางพอร์ต ADC ของไมโครคอนโทรลเลอร์ แสงตกกระทบสีดำจะมีการสะท้อนที่มากทำให้ค่าแรงดันที่อ่านได้มีค่ามากในขณะที่แสงกระทบพื้นสีขาวมีการสะท้อนที่ต่ำทำให้ค่าแรงดันที่อ่านได้มีค่าน้อย ซึ่งค่าที่อ่านได้มีค่าแตกต่างกันประมาณ 1-2 โวลต์ การติดตั้ง LDR และ LED Super Byte จะต้องไม่สูงจนเกินไปจนแสงตกกระทบน้อย ไม่ต่ำเกินไปจะชิดพื้น และอยู่ใกล้ติดกัน ดังรูปที่ 5.1 และการปรับระยะห่างระหว่าง LDR1 และ LDR2 จะมีผมต่อการเลี้ยวของหุ่นยนต์ด้วย

รูปที่ 5.1 แสดงอุปกรณ์ตรวจจับเทปดำโดยใช้โฟโต้รีซิสเตอร์และ LED Super byte

ผลการทดลอง

จากรูปที่ 5.2 ข้างล่างจะสังเกตเห็นไฟจาก LED Super Byte ตกกระทบพื้นสีขาวสะท้อนมายังโฟโต้รีซิสเตอร์แล้วทำการอ่านค่าแรงดันเพื่อแยกแยะแถบสีดำกับสีขาวผลการทดลองสามารถแยกแยะได้ดี



รูปที่ 5.2 แสดงเซนเซอร์ที่ทำการติดตั้งแล้วและทำการทดลองวิ่งตามเส้น

สามารถดูไฟล์วีดีโอการทำงานของหุ่นยนต์เคลื่อนที่ตามแถบเส้นสีดำได้ที่ Website http://www.apl1.sci.kmitl.ac.th

การทดลองที่ 2 วิ่งหาเส้นสีดำแล้วเข้าเส้นเมื่อหลุดออกจากเส้น

อุปกรณ์

1.LED Super Byte สีแดง 2 ตัว

2. โฟโต้รีซิสเตอร์ 3 ตัว

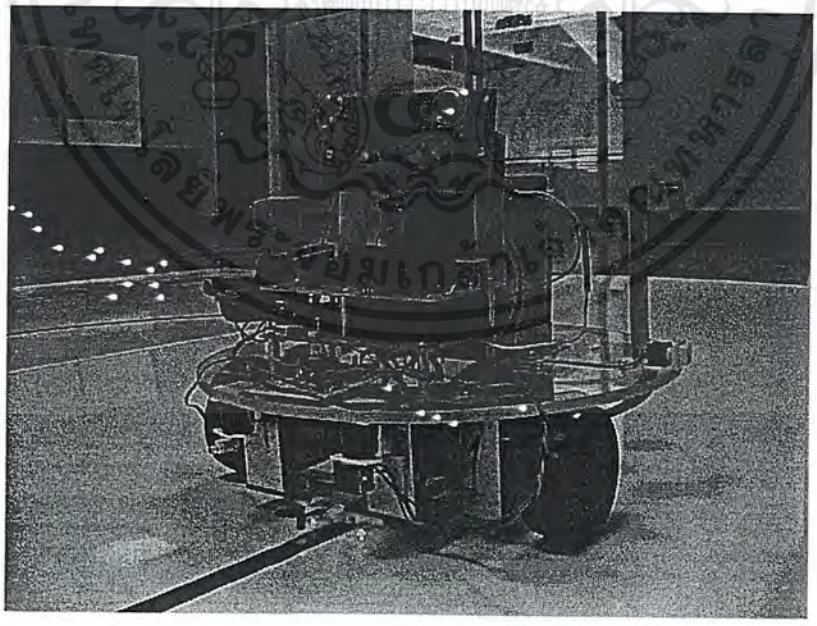
3. ตัวต้านทาน 220 โอมห์ 3 ตัว

หลักการทำงาน

จะมีโฟโต้รีซิสเตอร์และ LED Super Byte เพิ่มขึ้นมาอีกอย่างละ 1 ตัวเพื่อทำการตรวจสอบพื้นที่วิ่งเจอว่ามีสีอะไร หากหุ่นยนต์วิ่งออกจากแถบเส้นสีดำ โฟโต้รีซิสเตอร์ทั้ง 3 ตัวจะรับรู้ว่าเป็นพื้นสีขาวและ วิ่งไปจนกระทั่งโฟโต้รีซิสเตอร์ด้านหน้าตรวจจับพื้นว่ามีแถบสีอะไร ดังรูปที่ 5.3

รูปที่ 5.3 แสดงการวางตำแหน่งอุปกรณ์ตรวจจับหาเส้นสีดำเมื่อวิ่งหลุดออกจากเส้น

ผลการทดลอง



รูปที่ 5.4 แสดงเซนเซอร์ที่ทำการติดตั้งแล้วและใช้เซนเซอร์ตัวที่ 3 ทำการตรวจจับว่ามีการเข้าเส้นหรือเปล่า

สามารถดูไฟล์วีดีโอหุ่นยนต์เคลื่อนที่หาเส้นแถบดำเมื่อหุ่นยนต์วิ่งหลุดจากแถบดำได้ที่

การทดลองที่ 3 ทดสอบเซนเซอร์เสียง

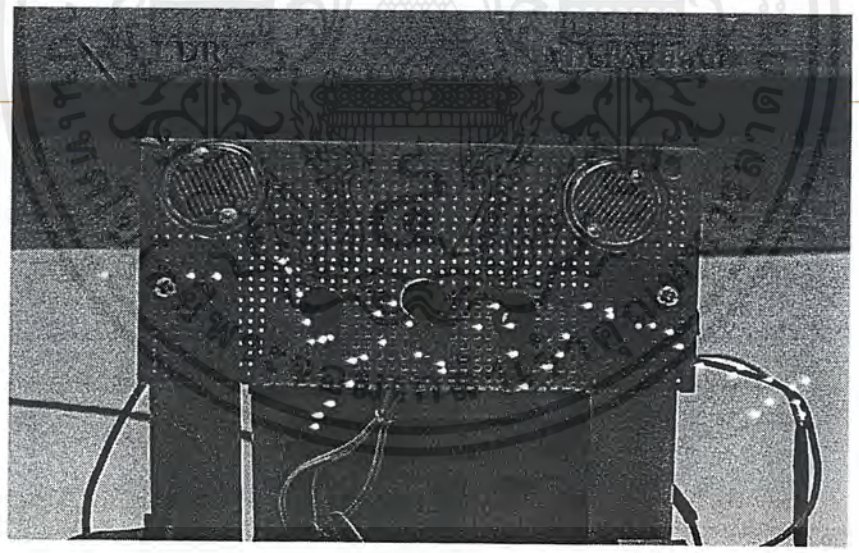
อุปกรณ์

1. ไมโครโฟน
2. LM368
3. ตัวเก็บประจุ 0.001, 10 และ 1000 ไมโครฟารัด
4. ตัวต้านทาน 2.2 กิโลโอห์ม

หลักการ

สัญญาณจากไมโครโฟนแบบธรรมดาจะมีการถูกขยายก่อนการอ่านสัญญาณโดยไมโครคอนโทรลเลอร์ ดังรูป 2.15 โดยการใช้ LM386 ออม-แอมป์ เป็นตัวขยายสัญญาณ ซึ่งเอาท์พุทที่ถูกขยายจะต่อไปยังขา A/D ที่ port E จากนั้นทำการวัดระดับแรงดัน โดยทำการตั้งแรงดันไว้ค่าหนึ่งจากโปรแกรม จากนั้นหากแรงดันสูงกว่าที่โปรแกรมไว้จะทำการวิ่งหนีหรือหยุด

ผลการทดลอง



รูปที่ 5.5 แสดงตำแหน่งของไมโครโฟนและ LDR

สามารถดูไฟล์วีดีโอหุ่นยนต์ตรวจจับเสียงได้ที่ Website http://www.apl1.sci.kmitl.ac.th

การทดลองที่ 4 ทดสอบเซนเซอร์แสงอินฟราเรด

อุปกรณ์

1.LED Infrared

2. IR detector

3. 74HC04

4. ตัวต้านทาน 100 โอห์ม 2 ตัว 5,6.8 และ 100 กิโลโอห์มอย่างละ 1 ตัว

5. ตัวเก็บประจุ 0.001 ไมโครฟารัด 1 ตัว

หลักการ

ทำการเจนเนอเรตความถี่ โดยใช้วงจร Oscillate ให้ LED Infrared กระพริบด้วยความถี่สามารถปรับความต้านทาน 5 กิโลโอห์มให้ได้ความถี่ตรงกับความถี่ที่ IR detector ตรวจจับได้ ตรวจจับสัญญาณที่ได้จาก IR detector หากตรวจจับมีวัตถุกีดขวาง แรงดันเอาต์พุทที่ได้จาก IR detector จะมีค่าสูงขึ้นหากมีค่าสูงกว่าที่เรากำหนดไว้โดยใช้โปรแกรม เราจะให้หุ่นยนต์เคลื่อนที่ถอยหลังเป็นระยะทางระยะทางหนึ่ง หลังจากนั้นจะทำการตรวจจับแสงอินฟราเรดอีกครั้งแล้วเคลื่อนที่

ผลการทดลอง

สามารถดูไฟล์วีดีโอหุ่นยนต์ตรวจจับแสงอินฟราเรดได้ที่

Website http://www.apl1.sci.kmitl.ac.th

ข้อบกพร่องที่พบ

1. แหล่งจ่ายไฟสำหรับหุ่นยนต์มักมีปัญหาทางด้านกำลังขับมอเตอร์ ซึ่ง IC ที่ใช้ขับเคลื่อนมอเตอร์คือ L293D มีเอาต์พุทสามารถจ่ายกระแสได้ข้างละ 600 มิลลิแอมป์ มีกำลังเพียง 5 วัตต์
2. เซนเซอร์ LDR ที่ใช้ในการเดินตามเส้นมักมีปัญหาในด้านการตั้งค่า ซึ่งขาเซนเซอร์ที่ทำการติดตั้งไม่มั่นคงสามารถขยับเขยื้อนได้ แสงที่รับได้จึงมักมีการคลาดเคลื่อน ต้องทำการตั้งค่าอยู่บ่อยๆ
3. เซนเซอร์ทางเสียง มักมีปัญหาในการตั้งค่าย่านแรงดันเริ่มต้นได้ยินเสียงในโปรแกรม ดังนั้นสมควรที่จะต่อวงจร High pass filter เพื่อกรองสัญญาณเสียงที่มีความถี่ต่ำ

ภาคผนวก ก. Library

```
/* lib_rw11.c */
/*    VERSION HISTORY FOR lib_rw11.c  V2.81          Jun 11 1994
Jlj Created library for Rug Warrior from an       earlier library by jsr and fgm*/
persistent int test_number;

/*******************/
/* TIME PRIMITIVES */
/*******************/
/***********************************/
/* location of various time stuff: */
/* 0x14: time in milliseconds      */
/***********************************/
void reset_system_time()
{
  pokeword(0x14, 0);
  pokeword(0x12, 0);
}

/*  returns valid time from 0 to 32,767 minutes (approx)  */
float seconds()
{
  return ((float) mseconds()) / 1000.;
}

void sleep(float seconds)
{
    msleep((long)(seconds * 1000.));
}

void msleep(long msec)
```

```
{
    long end_time= mseconds() + msec;
    while (1) {
        /* if the following test doesn't execute at least once a second,
          msleep may not halt */
        long done= mseconds()-end_time;
        if (done >= 0L && done <= 1000L) break;
    }
}

void beep()
{  tone(500., .3);
 }

/*  1/2 cycle delay in .5us goes in 0x26 and 0x27  */
void tone(float frequency, float length)
{
    pokeword(0x26, (int)(1E6 / frequency));
    bit_set(0x1020, 0b00000001);
    sleep(length);
    bit_clear(0x1020, 0b00000001);

    /*  following is important to reduce # of interrupts
        when tone is off  */
    pokeword(0x26, 0);

    bit_clear(0x1000, 8);
}

void beeper_on()
{
```

```
    bit_set(0x1020, 0b00000001);
    bit_set(0x1022, 0b00001000);
}

void beeper_off()
{
    bit_clear(0x1022, 0b00001000);
    bit_clear(0x1020, 0b00000001);
    pokeword(0x26, 0);
    bit_clear(0x1000, 0b00001000);      /* turn power to spkr off */
}

void set_beeper_pitch(float frequency)
{
    pokeword(0x26, (int)(1E6 / frequency));
}

long timer_create_mseconds(long timeout)
{
    return mseconds() + timeout;
}

long timer_create_seconds(float timeout)
{
    return mseconds() + (long) (timeout * 1000.);
}

int timer_done(long timer)
{
    return timer < mseconds();
}
```

```
/* Rug Warrior's analog ports 6 and 7 are unassigned */

int analog(int port)
{  poke(0x1039, 0b10000000);
   poke(0x1030, port);
   return peek(0x1031);
}

/*****************************/
/*** Multi-Tasking Support ***/
/*****************************/

/*  gives process that calls it 256 ticks (over 1/4 sec)
   more to run before being swapped out

   call repeatedly to hog processor indefinitely  */
void hog_processor()
{
   poke(0x0a, 0);
}

/* Debugging utilities */

/* Dump 8 bytes to the LCD screen, starting at addr */
void dump(int addr)
{ printf("%x: %x %x ", addr, peekword(addr), peekword(addr + 2));
  printf("     %x %x\n",  peekword(addr + 4), peekword(addr + 6));
}

/*****************************************************************************/
```

```
/* Rug Warrior specific functions and constants                    */
/*********************************************************************/

/* Digital ports 1 and 2 are unassigned */

int digital(int port)              /* Return 1 bit from PA1 or PA2 (or PA0) */
{ return 1 & (peek(0x1000) >> (port & 3));
}

/* Indices for accessing sensors connected to the A/D converter.
   e.g. to read value of right photo cells use analog(photo_right) */

int photo_right = 0;
int photo_left = 1;
int microphone = 2;
int pyro = 5;

/*********************************************************************/
/*
/* Motor Control Primitives
/*
/*   init_motors()              - Must be called to enable motors
/*   motor(index, speed)        - Control velocity of motor 0 (left) or 1 (right)
/*   drive(trans_vel, rot_vel)  - Control robot translation and rotation
/*   stop()                     - Stop both motors
*/

/* Setup two PWM channels for motor control */

/*            Left      Right       */
int TOCx[2]   = {0x1018,   0x101A};        /* Index for timer register */
```

```
int dir_mask[2] = {0b00010000, 0b00100000}; /* Port D direction bits */
int pwm_mask[2] = {0b01000000, 0b00100000};        /* Port A PWM bits */

int init_motors()
{ bit_set(0x1009,0b110000);    /* Set PD4,5 as outputs for motor direction */
  poke(0x100C,0b01100000);   /* OC1M Output compare 1 affects PA5,6 */
  bit_set(  0x1020,0b10100000);          /* TCTL1 OC3 turns off PA5, OC2 PA6 */
  bit_clear(0x1020,0b01010000);          /* Use set and clear to avoid other bits */
  pokeword(0x1018,0);            /* When TCNT = 0, OC1 fires */
}

/* Make sure init_motors is called after a reset */

int init_motors_dummy = init_motors();

void stop()                          /* Stop both drive motors */
{ bit_clear(0x100D,pwm_mask[0]);      /* Let OC1 turn off motors rather */
  bit_clear(0x100D,pwm_mask[1]);      /* than turn them on */
}

/* Vel is in the range [-100, +100], index = 0 => Left, = 1 => Right */

void motor(int index, float vel)
{ float avel       ;          /* Absolute value of velocity */
  if (vel > 0.0)
    { bit_set(0x1008, dir_mask[index]);  /* Forward rotation */
      avel = vel; }
  else
    { bit_clear(0x1008, dir_mask[index]); /* Backward rotation */
      avel = (- vel); }
  if (avel < 1.0)             /* If we are going real slow */
```

```
    bit_clear(0x100D,pwm_mask[index]);        /* then just have OC1 turn off the motor */
  else
    bit_set(0x100D,pwm_mask[index]); /* Otherwise let OC1 control the motors */
  if (avel > 99.0)                    /* If we are gong real fast */
    avel = 99.0;              /* don't let the following multiply overflow */
  pokeword(TOCx[index], (int) (655.56 * avel));/* Compute TOCx value */
}

/* Use drive to control motion of the robot.  A positive rot_vel makes the robot
   turn left. */

void drive(float trans_vel, float rot_vel)
{ motor(0,trans_vel - rot_vel);
  motor(1,trans_vel + rot_vel);
}

/* Enable and activate debugging LEDs. */
void leds(int val)
{ poke(0x1009,0b111100);      /* Set port D for output */
  poke(0x1008,val << 2);      /* Shift number over */
}

/* Return a 3-bit value representing which of the bumper switches are closed */
int bumper()
{ int bmpr;
  bmpr = analog(3);                    /* Switch closure: */
  if    (bmpr <  11) return 0b000;     /* none */
  else if (bmpr <  32) return 0b001;   /* A  */
  else if (bmpr <  53) return 0b010;   /*  B  */
  else if (bmpr <  74) return 0b011;   /* AB  */
  else if (bmpr <  96) return 0b100;   /*  C  */
```

```
  else if (bmpr < 117) return 0b101;     /* A C */
  else if (bmpr < 132) return 0b110;     /*  BC */
  else                 return 0b111;  /* ABC - (mechanically impossible) */
}

/* ir_detect returns:
    0b00 => no reflection, 0b01 => reflection on right,
    0b10 => reflection on left, 0b11 => reflection on both sides */

int ir_detect()
{ int val1, val2, val3;
  val1 = peek(0x100A) & 0b10000;      /* IR Detector connected to PE4 */
  bit_set(0x1008,0b1000);             /* Turn on Left emitter, PD3 */
  msleep(1L);                         /* Wait 1 millisecond */
  val2 = peek(0x100A) & 0b10000;      /* Should be Low if signal detected */
  bit_clear(0x1008,0b1000);           /* Turn off Left emitter */
  bit_set(0x1008,0b0100);             /* Turn on Right emitter, PD2 */
  msleep(1L);                         /* Wait 1 millisecond */
  val3 = peek(0x100A) & 0b10000;
  bit_clear(0x1008,0b1100);           /* Turn emitters off */
  /* For detection, detector must be high when emitter is off, low when on */
  return ((val1 & ~val2) >> 3)   | ((val1 & ~val3) >> 4); /* HI -> LOW */
}
```

ภาคผนวก ข. การใช้อินเตอร์เลคทีฟ C

Before starting IC, you will need to know the name of the serial port you intend to use. The device names vary from system to system. Often, a serial port is set up to not give read and write permissions to standard users. Occasionally, a serial port is set up to run "getty", which means the system thinks a serial terminal (such as a VT100) is connected, and that users will wish to log in. If in doubt, you should talk to your local system administrator to help you get started using the serial port.

Here are some typical serial port names, but realize that names sometimes vary even between different machines of the same architecture, so the following are suggestions for names to try, rather than the final answers:

- Linux: `` `/dev/ttyS0', `/dev/ttyS1', ... ``
- SPARC running SunOS: `` `/dev/ttya', `/dev/ttyb' ``
- RS/6000: `` `/dev/tty0' ``
- NeXTStation: `` `/dev/ttya', `/dev/ttyb' ``
- DECstation and HP Snake: `` `/dev/tty00', `/dev/tty01' ``

You may want to confirm that the serial port works before you try to use IC to talk to a board, as it is good to keep the number of unknowns to one at a time. One way to do this is to hook the serial port on your host machine to a serial source that is known to work (for instance, a modem or another serial port) and run `kermit`.

It is critical to get this step right before you continue. If you have not used serial ports on a UNIX machine before, I highly recommend you find someone more experienced to help you.

Now you should be ready to plug in your board. Refer to the owner's manual of your board for instructions.

If your board does not have "p-code" loaded, you will have to download the p-code before you can run IC. (Try turning your board on; if a message appears that starts "IC v3.1", your board already has p-code loaded, and you can skip this step.) To download p-code, find the appropriate `.s19` file for your board in `/usr/local/lib/ic` (or wherever you placed the library files); this should be `pcode_hb.s19` for the Handy Board, `pcode_rw.s19` for the Rug Warrior, or `pcoder22.s19` for the 6.270 board. Run the `dl` program that you just installed with the `.s19` file you've found, like `dl pcode_hb.s19 -port /dev/cua0` (replace `/dev/cua0` with the serial port on your machine); follow the instructions that `dl` prints.

At this point you should be ready to run `ic`. Before you can run it, you'll have to enter a license key and name with `ic -config` (see section Configuring

IC under UNIX); you should have received a license key when you purchased IC.
Try running IC and typing "2+2"; you should get the response "Returned <int> 4".

Getting Started on the Mac

You should have a file named `ic_mac_3.1.sea.hqx` (the version number may be different). You can convert this to `ic_mac_3.1.sea` with any of several popular format conversion tools, such as Compact Pro. Once you have the file `ic_mac_3.1.sea`, simply execute it; it will ask you where you want to place the Interactive C folder, and then extract the IC files in that location. *Warning: if you're upgrading from a previous version of IC 3.1 or later, don't delete the old version until you've run it and written down the license key and name so that you can enter the information into your new copy.*
Plug the board into one of your computer's serial ports, and run IC. The first time you run IC, it will ask you for your license key and name; you should have received a license key when you purchased IC.
Try turning your board on. If your board has its "p-code" loaded, a message should come up that starts "IC v3.1"; if it does not, you will have to reload the p-code. You can do this with the "Reload pcode..." command (on the File menu); the program will guide you through the steps.
Now you should be ready to use IC. Try running IC and typing "2+2"; you should get the response "Returned <int> 4".

Getting Started under Windows

You should have a file named `icw31.exe` (the version number may be different). CD to the directory where you want to install IC, and run `icw31.exe`. This will unpack `ic.exe` and the libs directory.
Plug the board into one of your computer's serial ports, and run IC. The first time you run IC, it will ask you for your license key and name; you should have received a license key when you purchased IC.
Try turning your board on. If your board has its "p-code" loaded, a message should come up that starts "IC v3.1"; if it does not, you will have to reload the p-code. You can do this with the "Reload PCode..." command (on the Board menu); the program will guide you through the steps.
Now you should be ready to use IC. Try running IC and typing "2+2"; you should get the response "Returned <int> 4".

Using IC

When IC is running and attached to a 6811 system, C expressions, function calls, and IC commands may be typed at the "C>" prompt.

For example, to evaluate the arithmetic expression `1 + 2`, type the following:

```
C> 1 + 2
```

When this expression is typed, it is compiled by the console computer and then downloaded to the 6811 system for evaluation. The 6811 then evaluates the compiled form and returns the result, which is printed on the console computer's screen.

To evaluate a series of expressions, create a C block by beginning with an open curly brace `{` and ending with a close curly brace `}`. The following example creates a local variable `i` and prints 10 (the sum of `i + 7`) to the 6811's LCD screen:

```
C>{int i=3; printf("%d", i+7);}
```

IC Commands

IC responds to the following commands:

`load` *`filename`*

Load file. The command `load` *`filename`* compiles and loads the named file. The board must be attached for this to work. IC looks first in the local directory and then in the IC library path for files. Several files may be loaded into IC at once, allowing programs to be defined in multiple files.

`unload` *`filename`*

Unload file. The command `unload` *`filename`* unloads the named file, and re-downloads remaining files.

`list files`

The command `list files` displays the names of all files presently loaded into IC.

`list globals`

The command `list globals` displays the names of all currently defined global variables.

`list functions`

The command `list functions` displays the names of presently defined C functions.

`list defines`

The command `list defines` displays the names and values of all currently defined preprocessor macros.

`kill_all`

Kill all processes. The command `kill_all` kills all currently running processes.

`ps`

Print process status. The command `ps` prints the status of currently running processes.

`help`

Help. The command `help` displays a help screen of IC commands.

`quit`

Quit. The command `quit` exits IC. ^c can also be used.

Line Editing

IC has a built-in line editor and command history, allowing editing and re-use of previously typed statements and commands. The mnemonics for these functions are based on standard Emacs control key assignments.

To scan forward and backward in the command history, type ^p or **uparrow** for backward, and ^n or **downarrow** for forward.

*Under Windows, only the functionality described in the rest of this section is not available (only ^p, ^n, **uparrow**, and **downarrow** are available). However, normal Windows editing methods can be used.*

Under UNIX, an earlier line in the command history can be retrieved by typing the exclamation point followed by the first few characters of the line to retrieve, and then the space bar. For example, if you had previously typed the command `C> load foo.c`, then typing `C>!lo` followed by a space would retrieve the line `C> load foo`.

The following are the keystroke mappings understood by IC for the Mac and UNIX versions.

Key	Function
del	backward delete character
ctrl-d	forward delete character
ctrl-b , back arrow	go backward a character
ctrl-f , forward arrow	go forward a character
ctrl-a	go to beginning of line
ctrl-e	go to end of line
ctrl-k	kill line (from cursor forward)
ESC-d	forward kill word
ESC-DEL	backward kill word
up arrow, **ctrl-p**	history last
down arrow, **ctrl-n**	history next

The Mac and UNIX versions of IC do parenthesis-balance-highlighting as expressions are typed.

The main() Function

After functions have been downloaded to the board, they can be invoked from the IC prompt. If one of the functions is named `main()`, it will automatically be run when the board is reset.

To reset the board *without* running the `main()` function (for instance, when hooking the board back to the computer), hold down the board's **`Start`** button (or **`Escape`** on a 6.270 board) while pressing reset. The board will reset without running `main()`.

IC versus Standard C

The IC programming language is based loosely on ANSI C. However, there are major differences.

Many of these differences arise from the desire to have IC be "safer" than standard C. For instance, in IC, array bounds are checked at run time; for this reason, arrays cannot be converted to pointers in IC. Also, in IC, pointer arithmetic is not allowed.

Other differences are due to the desire that the IC runtime be small and efficient. For instance, the IC printf function does not understand many of the more exotic formatting options specified by ANSI C.

Yet other differences are due to the desire that IC be simpler than standard C. This is the reason for the global scope of all declarations.

In the rest of this document, when we refer to "C", the statement applies to both IC and standard C. When we wish to specify one or the other, we will refer to either "IC" or "standard C". When no such qualifiers are present, you should assume that we are talking about IC.

A Quick C Tutorial

Most C programs consist of function definitions and data structures. Here is a simple C program that defines a single function, called `main`.

```
void main()
{
    printf("Hello, world!\n");
}
```

All functions must have a return type. Since `main` does not return a value, it uses `void`, the null type, as its return type. Other types include integers (`int`) and floating point numbers (`float`). This *function declaration* information must precede each function definition.

Immediately following the function declaration is the function's name (in this case, `main`). Next, in parentheses, are any arguments (or inputs) to the function. `main` has none, but a empty set of parentheses is still required.

After the function arguments is an open curly-brace `{`. This signifies the start of the actual function code. Curly-braces signify program *blocks*, or chunks of code.

Next comes a series of C *statements*. Statements demand that some action be taken. Our demonstration program has a single statement, a `printf` (formatted print). This will print the message "`Hello, world!`" to the LCD display. The `\n` indicates end-of-line.

The `printf` statement ends with a semicolon (`;`). *All* C statements must be ended by a semicolon. Beginning C programmers commonly make the error of omitting the semicolon that is required at the end of each statement.

The `main` function is ended by the close curly-brace `}`.

Let's look at an another example to learn some more features of C. The following code defines the function *square*, which returns the mathematical square of a number.

```
int square(int n)
{
    return(n * n);
}
```

The function is declared as type `int`, which means that it will return an integer value. Next comes the function name `square`, followed by its argument list in parentheses. `square` has one argument, `n`, which is an integer. Notice how declaring the type of the argument is done similarly to declaring the type of the function.

When a function has arguments declared, those argument variables are valid within the "scope" of the function (i.e., they only have meaning within the function's own code). Other functions may use the same variable names independently.

The code for `square` is contained within the set of curly braces. In fact, it consists of a single statement: the `return` statement. The `return` statement exits the function and returns the value of the C *expression* that follows it (in this case "`n * n`").

Expressions are evaluated according set of precendence rules depending on the various operations within the expression. In this case, there is only one operation (multiplication), signified by the "`*`", so precedence is not an issue.

Let's look at an example of a function that performs a function call to the `square` program.

```
float hypotenuse(int a, int b)
{
    float h;
```

```
    h = sqrt((float)(square(a) + square(b)));

    return(h);
}
```

This code demonstrates several more features of C. First, notice that the floating point variable `h` is defined at the beginning of the `hypotenuse` function. In general, whenever a new program block (indicated by a set of curly braces) is begun, new local variables may be defined.

The value of `h` is set to the result of a call to the `sqrt` function. It turns out that `sqrt` is a built-in IC function that takes a floating point number as its argument.

We want to use the `square` function we defined earlier, which returns its result as an integer. But the `sqrt` function requires a floating point argument. We get around this type incompatibility by *coercing* the integer sum `(square(a) + square(b))` into a float by preceding it with the desired type, in parentheses. Thus, the integer sum is made into a floating point number and passed along to `sqrt`.

The `hypotenuse` function finishes by returning the value of `h`.

This concludes the brief C tutorial.

Data Objects

Variables and constants are the basic data objects in a C program. Declarations list the variables to be used, state what type they are, and may set their initial value.

Variables

Variable names are case-sensitive. The underscore character is allowed and is often used to enhance the readability of long variable names. C keywords like `if`, `while`, etc. may not be used as variable names.

Global variables and functions may not have the same name. In addition, if a local variable is named the same as a global or a function, the use of that global or function is prevented within the scope of the local variable.

Declaration

In C, variables can be declared at the top level (outside of any curly braces) or at the start of each block (a functional unit of code surrounded by curly braces). In general, a variable declaration is of the form:

```
<type> <variable name>;
or
<type> <variable name>=<initialization data>;
```

In IC, <type> can be `int`, `long`, `float`, `char`, or `struct <struct name>`, and determines the *primary type* of the variable declared. This form changes somewhat when dealing with pointer and array declarations, which are explained in a later section, but in general this is the way you declare variables.

Local and Global Scopes

If a variable is declared within a function, or as an argument to a function, its binding is *local*, meaning that the variable has existence only within that function definition.

If a variable is declared outside of a function, it is a global variable. It is defined for all functions, including functions which are defined in files other than the one in which the global variable was declared.

Variable Initialization

Local and global variables can be initialized to a value when they are declared. If no initialization value is given, the variable is initialized to zero. All global variable declarations must be initialized to constant values. Local variables may be initialized to the value of arbitrary expressions including any globals, function calls, function arguments, or locals which have already been initialized.

Here is a small example of how initialized declarations are used.

```
int i=50;       /* declare i as global integer --
initial value 50 */
long j=100L;    /* declare j as global long --
initial value 100 */
int foo()
{
   int x;       /* declare x as local integer with
initial value 0 */
   long y=j;    /* declare y as local integer with
initial value j */
}
```

Local variables are initialized whenever the function containing them runs. Global variables are initialized whenever a reset condition occurs. Reset conditions occur when:

- Code is downloaded;
- The `main()` procedure is run;
- System hardware reset occurs.

Persistent Global Variables

A special *persistent* form of global variable, has been implemented for IC. A persistent global may be initialized just like any other global, but its value is

only initialized when the code is downloaded and not on any other reset conditions. If no initialization information is included for a persistent its value will be initialized to zero on download, but left unchanged on all other reset conditions.

To make a persistent global variable, prefix the type specifier with the keyword `persistent`. For example, the statement

```
persistent int i=500;
```

creates a global integer called `i` with the initial value 500.

Persistent variables keep their state when the board is turned off and on, when `main` is run, and when system reset occurs. Persistent variables will lose their state when code is downloaded as a result of loading or unloading a file. However, it is possible to read the values of your persistents in IC if you are still running the same IC session from which the code was downloaded. In this manner you could read the final values of calibration persistents, for example, and modify the initial values given to those persistents appropriately.

Persistent variables were created with two applications in mind:

- Calibration and configuration values that do not need to be re-calculated on every reset condition.
- Robot learning algorithms that might occur over a period when the robot is turned on and off.

Constants

Integer Constants

Integers constants may be defined in decimal integer format (e.g., `4053` or `-1`), hexadecimal format using the "`0x`" prefix (e.g., `0x1fff`), and a non-standard but useful binary format using the "`0b`" prefix (e.g., `0b1001001`). Octal constants using the zero prefix are not supported.

Long Integer Constants

Long integer constants are created by appending the suffix "`l`" or "`L`" (upper- or lower-case alphabetic L) to a decimal integer. For example, `0L` is the long zero. Either the upper or lower-case "L" may be used, but upper-case is the convention for readability.

Floating Point Constants

Floating point numbers may use exponential notation (e.g., "`10e3`" or "`10E3`") or may contain a decimal period. For example, the floating point zero can be given as "`0.`", "`0.0`", or "`0E1`", but not as just "`0`". *Since the 6811 has no floating point hardware, floating point operations are much slower than integer operations, and should be used sparingly.*

Characters and String Constants

Quoted characters return their ASCII value (e.g., `'x'`).
Character string constants are defined with quotation marks, e.g., `"This is a character string."`.

NULL

The special constant NULL has the value of zero and can be assigned to and compared to pointer or array variables (which will be described in later sections). In general, you cannot convert other constants to be of a pointer type, so there are many times when NULL can be useful.
For example, in order to check if a pointer has been initialized you could compare its value to NULL and not try to access its contents if it was NULL. Also, if you had a defined a linked list type consisting of a value and a pointer to the next element, you could look for the end of the list by comparing the next pointer to NULL.

Data Types

IC supports the following data types:

16-bit Integers

16-bit integers are signified by the type indicator `int`. They are signed integers, and may be valued from -32,768 to +32,767 decimal.

32-bit Integers

32-bit integers are signified by the type indicator `long`. They are signed integers, and may be valued from -2,147,483,648 to +2,147,483,647 decimal.

32-bit Floating Point Numbers

Floating point numbers are signified by the type indicator `float`. They have approximately seven decimal digits of precision and are valued from about 10^-38 to 10^38.

8-bit Characters

Characters are an 8-bit number signified by the type indicator `char`. A character's value typically represents a printable symbol using the standard ASCII character code, but this is not necessary; characters can be used to refer to arbitrary 8-bit numbers.

Pointers

IC pointers are 16-bit numbers which represent locations in memory. Values in memory can be manipulated by calculating, passing and *dereferencing* pointers representing the location where the information is stored.

Arrays

Arrays are used to store homogenous lists of data (meaning that all the elements of an array have the same type). Every array has a length which is

determined at the time the array is declared. The data stored in the elements of an array can be set and retrieved in the same manner that other variables can be.

Structures

Structures are used to store non-homogenous but related sets of data. Elements of a structure are referenced by name instead of number and may be of any supported type. Structures are useful for organizing related data into a coherent format, reducing the number of arguments passed to functions, increasing the effective number of values which can be returned by functions, and creating complex data representations such as directed graphs and linked lists.

Pointers

The address where a value is stored in memory is known as the *pointer* to that value. It is often useful to deal with pointers to objects, but great care must be taken to insure that the pointers used at any point in your code really do point to valid objects in memory. Attempts to refer to invalid memory locations could corrupt your memory. Most computing environments that you are probably used to return helpful messages like `Segmentation Violation' or `Bus Error' on attempts to access illegal memory. However, no such safety net exists in the 6.270 system and invalid pointer dereferencing is very likely to go undetected and cause serious damage to your data, your program, or even the pcode interpreter.

Pointer Safety

In past versions of IC, you could not return pointers from functions or have arrays of pointers. In order to facilitate the use of structures, these features have been added to the current version. With this change, the number of opportunities to misuse pointers have increased. However, if you follow a few simple precautions you should do fine.

First, you should always check that the value of a pointer is not equal to NULL (a special zero pointer) before you try to access it. Variables which are declared to be pointers are initialized to NULL, so many uninitialized values could be caught this way.

Second, you should never use a pointer to a local variable in a manner which could cause it to be accessed after the function in which it was declared terminates. When a function terminates the space where its values were being stored is recycled. Therefore not only may dereferencing such pointers return incorrect values, but assigning to those addresses could lead to serious data corruption. A good way to prevent this is to never return the address of a local variable from the function which declares it and never store those

pointers in an object which will live longer than the function itself (a global pointer, array, or struct). Global variables and variables local to main will not move once declared and their pointers can be considered to be secure.
The type checking done by IC will help prevent many mishaps, but it will not catch all errors, so be careful.

Pointer Declaration and Use

A variable which is a pointer to an object of a given type is declared in the same manner as a regular object of that type, but with an extra * in front of the variable name.
The value stored at the location the pointer refers to is accessed by using the * operator before the expression which calculates the pointer. This process is known as dereferencing
The address of a variable is calculated by using the & operator before that variable, array element, or structure element reference.
There are two main differences between how you would use a variable of a given type and a variable declared as a pointer to that type.
For the following explanation, consider X and Xptr as defined as follows:

```
long X;
long *Xptr;
```

- Space Allocation -- Declaring an object of a given type, as `X` is of type `long`, allocates the space needed to store that value. Because an IC long takes four bytes of memory, four bytes are reserved for the value of `X` to occupy. However, a pointer like `Xptr` does not have the same amount of space allocated for it that is needed for an object of the type it points to. Therefore it can only safely refer to space which has already been allocated for globals (in a special section of memory reserved for globals) or locals (temporary storage on the stack).
- Initial Value -- It is always safe to refer to a non-pointer type, even if it hasn't been initialized. However pointers have to be specifically assigned to the address of legally allocated space or to the value of an already initialized pointer before they are safe to use.

So, for example, consider what would happen if the first two statements after X and Xptr were declared were the following:

```
X=50L;
*Xptr=50L;
```

The first statement is valid: it sets the value of `X` to 50L. The second statement would be valid if `Xptr` had been properly initialized, but in this case it is not. Therefore, this statement would corrupt memory.

Here is a sequence of commands you could try which illustrate how pointers and the * and & operators are used. It also shows that once a pointer has been set to point at a place in memory, references to it actually share the same memory as the object it points to:

```
X=50L;              /* set the memory allocated for X
to the value 50 */
Xptr=&X;    /* set Xptr to point to X */
*Xptr;              /* see that the value pointed at
by Xptr is 50 */
X=100L;             /* set X to the value 100 */
*Xptr;              /* see that the value pointed at
by Xptr changed to 100 */
*Xptr=200L; /* set the value pointed at by Xptr to
200 */
X;          /* see that the value in X changed to
200 */
```

Passing Pointers as Arguments

Pointers can be passed to functions and functions can change the values of the variables that are pointed at. This is termed *call-by-reference*; a reference, or pointer, to a variable is given to the function that is being called. This is in contrast to *call-by-value*, the standard way that functions are called, in which the value of a variable is given the to function being called.

The following example defines an `average_sensor` function which takes a port number and a pointer to an integer variable. The function will average the sensor and store the result in the variable pointed at by `result`. Function arguments are declared to be pointers by prepending a star to the argument name, just as is done for other variable declarations.

```
void average_sensor(int port, int *result)
{
  int sum= 0;
  int i;

  for (i= 0; i< 10; i++) sum += analog(port);

  *result=  sum/10;
}
```

Notice that the function itself is declared as a `void`. It does not need to return anything, because it instead stores its answer in the pointer variable that is passed to it.

The pointer variable is used in the last line of the function. In this statement, the answer `sum/10` is stored at the location pointed at by `result`. Notice that the asterisk is used to assign a value to the *location* pointed by `result`.

Returning Pointers from Functions

Pointers can also be returned from functions. Functions are defined to return pointers by preceeding the name of the function with a star, just like any other type of pointer declaration.

```
int right,left;

int *dirptr(int dir)
{
   if(dir==0) {
      return(&right);
   }
   if(dir==1) {
      return(&left);
   }
   return(NULL);
}
```

The function `dirptr` returns a pointer to the global `right` when its argument `dir` is 0, a pointer to `left` when its argument is 1, and NULL if its argument is other than 0 or 1.

Arrays

IC supports arrays of characters, integers, long integers, floating-point numbers, structures, pointers, and array pointers (multi-dimensional arrays). While unlike regular C arrays in a number of respects, they can be used in a similar manner. The main reasons that arrays are useful are that they allow you to allocate space for many instances of a given type, send an arbitrary number of values to functions, and iterate over a set of values.

Arrays in IC are different and incompatible with arrays in other versions of C. This incompatibility is caused by the fact that references to IC arrays are checked to insure that the reference is truly within the bounds of that array. In order to accomplish this checking in the general case, it is necessary that the size of the array be stored with the contents of the array. *It is important to remember that an array of a given type and a pointer to the same type are incompatible types in IC, whereas they are largely interchangeable in regular C.*

Declaring and Initializing Arrays

Arrays are declared using square brackets. The following statement declares an array of ten integers:

```
int foo[10];
```
In this array, elements are numbered from 0 to 9. Elements are accessed by enclosing the index number within square brackets: `foo[4]` denotes the fifth element of the array `foo` (since counting begins at zero).
Arrays are initialized by default to contain all zero values. Arrays may also be initialized at declaration by specifying the array elements, separated by commas, within curly braces. If no size value is specified within the square brackets when the array is declared but initialization information is given, the size of the array is determined by the number of elements given in the declaration. For example,
```
int foo[]= {0, 4, 5, -8,  17, 301};
```
creates an array of six integers, with `foo[0]` equalling 0, `foo[1]` equalling 4, etc.
If a size is specified and initialization data is given, the length of the initialization data may not exceed the specified length of the array or an error results. If, on the other hand, you specify the size and provide fewer initialization elements than the total length of the array, the remaining elements are initialized to zero.
Character arrays are typically text strings. There is a special syntax for initializing arrays of characters. The character values of the array are enclosed in quotation marks:
```
char string[]= "Hello there";
```
This form creates a character array called `string` with the ASCII values of the specified characters. In addition, the character array is terminated by a zero. Because of this zero-termination, the character array can be treated as a string for purposes of printing (for example). Character arrays can be initialized using the curly braces syntax, but they will not be automatically null-terminated in that case. In general, printing of character arrays that are *not* null-terminated will cause problems.

Passing Arrays as Arguments

When an array is passed to a function as an argument, the array's pointer is actually passed, rather than the elements of the array. If the function modifies the array values, the array will be modified, since there is only one copy of the array in memory.
In normal C, there are two ways of declaring an array argument: as an array or as a pointer to the type of the array's elements. In IC array pointers are incompatible with pointers to the elements of an array so such arguments can only be declared as arrays.

As an example, the following function takes an index and an array, and returns the array element specified by the index:

```
int retrieve_element(int index, int array[])
{
    return array[index];
}
```

Notice the use of the square brackets to declare the argument `array` as a pointer to an array of integers.

When passing an array variable to a function, you are actually passing the value of the array pointer itself and not one of its elements, so no square brackets are used.

```
void foo()
{
    int array[10];

    retrieve_element(3, array);
}
```

Multi-dimensional Arrays

A two-dimensional array is just like a single dimensional array whose elements are one-dimensional arrays. Declaration of a two-dimensional array is as follows:

```
int k[2][3];
```

The number in the first set of brackets is the number of 1-D arrays of `int`. The number in the second set of brackets is the length of each of the 1-D arrays of `int`. In this example, `k` is an array containing two 1-D arrays; `k[0]` is a 1-D array of `int` of length 3; `k[0][1]` is an `int`. Arrays of with any number of dimensions can be generalized from this example by adding more brackets in the declaration.

Determining the size of Arrays at Runtime

An advantage of the way IC deals with arrays is that you can determine the size of arrays at runtime. This allows you to do size checking on an array if you are uncertian of its dimensions and possibly prevent your program from crashing.

Since `_array_size` is not a standard C feature, code written using this primitive will only be able to be compiled with IC .

The `_array_size` primitive returns the size of the array given to it regardless of the dimension or type of the array. Here is an example of declarations and interaction with the _array_size primitive:

```
int i[4]={10,20,30};
int j[3][2]={{1,2},{2,4},{15}};
```

```
int k[2][2][2];

_array_size(i);          /* returns 4 */
_array_size(j);  /* returns 3 */
_array_size(j[0]);       /* returns 2 */
_array_size(k);          /* returns 2 */
_array_size(k[0]);       /* returns 2 */
```

Structures

Structures are used to store non-homogenous but related sets of data. Elements of a structure are referenced by name instead of number and may be of any supported type. Structures are useful for organizing related data into a coherent format, reducing the number of arguments passed to functions, increasing the effective number of values which can be returned by functions, and creating complex data representations such as directed graphs and linked lists.

The following example shows how to define a structure, declare a variable of structure type, and access its elements.

```
struct foo {
    int i;
    int j;
};

struct foo f1;

void set_f1(int i,int j)
{
    f1.i=i;
    f1.j=j;
}
void get_f1(int *i,int *j)
{
    *i=f1.i;
    *j=f1.j;
}
```

The first part is the structure definition. It consists of the keyword `struct`, followed by the name of the structure (which can be any valid identifier), followed by a list of named elements in curly braces. This definition specifies the structure of the type `struct foo`.

Once there is a definition of this form, you can use the type `struct foo` just like any other type. The line `struct foo f1;` is a global variable declaration which declares the variable `f1` to be of type `struct foo`.

The dot operator is used to access the elements of a variable of structure type. In this case, `f1.i` and `f1.j` refer to the two elements of `f1`. You can treat the quantities `f1.i` and `f1.j` just as you would treat any variables of type `int` (the type of the elements was defined in the structure declaration at the top to be `int`).

Pointers to structure types can also be used, just like pointers to any other type. However, with structures, there is a special short-cut for referring to the elements of the structure pointed to.

```
struct foo *fptr;

void main()
{
   fptr=&f1;
   fptr->i=10;
   fptr->j=20;
}
```

In this example, `fptr` is declared to be a pointer to type `struct foo`. In `main`, it is set to point to the global `f1` defined above. Then the elements of the structure pointed to by `fptr` (in this case these are the same as the elements of `f1`), are set. The arrow operator is used instead of the dot operator because fptr is a pointer to a variable of type `struct foo`. Note that `(*fptr).i` would have worked just as well as `fptr->i`, but it would have been clumsier.

Note that only pointers to structures, not the structures themselves, can be passed to or returned from functions.

Complex Initialization examples

Complex types -- arrays and structures -- may be initialized upon declaration with a sequence of constant values contained within curly braces and separated by commas. Arrays of character may also be initialized with a quoted string of characters.

For initialized declarations of single dimensional arrays, the length can be left blank and a suitable length based on the initialization data will be assigned to it. *Multi-dimensional arrays must have the size of all dimensions specified when the array is declared.* If a length is specified, the initialization data may not overflow that length in any dimension or an error will result. However, the initialization data may be shorter than the specified size and the remaining entries will be initialized to 0.

Following is an example of legal global and local variable initializations:

```
/* declare many globals of various types */
```

```
int i=50;

int *ptr=NULL;

float farr[3]={ 1.2, 3.6, 7.4 };
int tarr[2][4]={ { 1, 2, 3, 4 }, { 2, 4, 6, 8} };

char c[]="Hi there how are you?";
char carr[5][10]={"Hi","there","how","are","you"};

struct bar {
    int i;
    int *p;
    long j;} b={5, NULL, 10L};
struct bar barr[2] = { { 1, NULL, 2L }, { 3 } };

/* declare locals of various types */
int foo()
{
  int x;                    /* create local variable x
                               with initial value 0
*/
  int y= tarr[0][2];        /* create local variable y
                               with initial value 3
*/
  int *iptr=&i;             /* create a local pointer
to integer
                               which points to the
global i */
  int larr[2]={10,20};      /* create a local array
larr
                               with elements 10 and 20
*/
  struct bar lb={5,NULL,10L}; /* create a local
variable of type
                               struct bar with i=5 and
j=10 */
  char lc[]=carr[2];        /* create a local string
lc with
                               initial value "how" */
  ...
}
```

Statements and Expressions

Operators act upon objects of a certain type or types and specify what is to be done to them. Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

Operators

Each of the data types has its own set of operators that determine which operations may be performed on them.

Integer Operations

The following operations are supported on integers:

- **Arithmetic.** addition +, subtraction −, multiplication *, division /.
- **Comparison.** greater-than >, less-than <, equality ==, greater-than-equal >=, less-than-equal <=.
- **Bitwise Arithmetic.** bitwise-OR |, bitwise-AND &, bitwise-exclusive-OR ^, bitwise-NOT ~.
- **Boolean Arithmetic.** logical-OR ||, logical-AND &&, logical-NOT !. When a C statement uses a boolean value (for example, `if`), it takes the integer zero as meaning false, and any integer other than zero as meaning true. The boolean operators return zero for false and one for true. Boolean operators && and || will stop executing as soon as the truth of the final expression is determined. For example, in the expression `a && b`, if `a` is false, then `b` does not need to be evaluated because the result must be false. The && operator therefore will not evaluate `b`.

Long Integers

A subset of the operations implemented for integers are implemented for long integers: arithmetic addition +, subtraction −, and multiplication *, and the integer comparison operations. Bitwise and boolean operations and division are not supported.

Floating Point Numbers

IC uses a package of public-domain floating point routines distributed by Motorola. This package includes arithmetic, trigonometric, and logarithmic functions. Since floating point operations are implemented in software, they are much slower than the integer operations; we recommend against using floating point if you're concerned about performance.

The following operations are supported on floating point numbers:

- **Arithmetic.** addition +, subtraction −, multiplication *, division /.
- **Comparison.** greater-than >, less-than <, equality ==, greater-than-equal >=, less-than-equal <=.

- **Built-in Math Functions.** A set of trigonometric, logarithmic, and exponential functions is supported. See section Floating Point Functions, for details.

Characters

Characters are only allowed in character arrays. When a cell of the array is referenced, it is automatically coerced into a integer representation for manipulation by the integer operations. When a value is stored into a character array, it is coerced from a standard 16-bit integer into an 8-bit character (by truncating the upper eight bits).

Assignment Operators and Expressions

The basic assignment operator is =. The following statement adds 2 to the value of a.

```
a = a + 2;
```

The abbreviated form

```
a += 2;
```

could also be used to perform the same operation.

All of the following binary operators can be used in this fashion:

```
+   -   *   /   %   <<   >>   &   ^   |
```

Increment and Decrement Operators

The increment operator "++" increments the named variable. For example, the statement "a++" is equivalent to "a= a+1" or "a+= 1".

A statement that uses an increment operator has a value. For example, the statement

```
a= 3;
printf("a=%d a+1=%d\n", a, ++a);
```

will display the text "a=3 a+1=4."

If the increment operator comes after the named variable, then the value of the statement is calculated *after* the increment occurs. So the statement

```
a= 3;
printf("a=%d a+1=%d\n", a, a++);
```

would display "a=3 a+1=3" but would finish with a set to 4.

The decrement operator "--" is used in the same fashion as the increment operator.

Data Access Operators

&

A single ampersand preceeding a variable, an array reference, or a structure element reference returns a pointer to the location in memory where that information is being stored. This should not be used on

arbitrary expressions as they do not have a stable place in memory where they are being stored.

`*`

A single star preceeding an expression which evaluates to a pointer returns the value which is stored at that address. This process of accessing the value stored within a pointer is known as dereferencing.

`[expr]`

An expression in square braces following an expression which evaluates to an array (an array variable, the result of a function which returns an array pointer, etc.) checks that the value of the expression falls within the bounds of the array and references that element.

`.`

A dot between a structure variable and the name of one of its fields returns the value stored in that field.

`->`

An arrow between a pointer to a structure and the name of one of its fields in that structure acts the same as a dot does, except it acts on the structure pointed at by its left hand side. Where f is a structure of a type with e as an element name, the two expressions `f.i` and `(&f)->i` are equivalent.

Precedence and Order of Evaluation

The following table summarizes the rules for precedence and associativity for the C operators. Operators listed earlier in the table have higher precedence; operators on the same line of the table have equal precedence.

Operator	**Associativity**
() []	**left to right**
! ~ ++ -- - (*type*)	**right to left**
*** / %**	**left to right**
+ -	**left to right**
<< >>	**left to right**
< <= > >=	**left to right**
== !=	**left to right**
&	**left to right**
^	**left to right**
|	**left to right**
&&	**left to right**
||	**right to left**
= += -= etc.	**right to left**
,	**left to right**

Control Flow

IC supports most of the standard C control structures. One notable exception is the `switch` statement, which is not supported.

Statements and Blocks

A single C statement is ended by a semicolon. A series of statements may be grouped together into a *block* using curly braces. Inside a block, local variables may be defined.

If-Else

The `if else` statement is used to make decisions. The syntax is:

```
if (expression)
   statement-1
else
   statement-2
```

expression is evaluated; if it is not equal to zero (e.g., logic true), then *statement-1* is executed.

The `else` clause is optional. If the `if` part of the statement did not execute, and the `else` is present, then *statement-2* executes.

While

The syntax of a `while` loop is the following:

```
while (expression)
   statement
```

`while` begins by evaluating *expression*. If it is false, then *statement* is skipped. If it is true, then *statement* is evaluated. Then the expression is evaluated again, and the same check is performed. The loop exits when *expression* becomes zero.

One can easily create an infinite loop in C using the `while` statement:

```
while (1)
   statement
```

For

The syntax of a `for` loop is the following:

```
for (expr-1;expr-2;expr-3)
   statement
```

This is equivalent to the following construct using `while`:

```
expr-1;
while (expr-2) {
   statement
   expr-3;
}
```

Typically, *expr-1* is an assignment, *expr-2* is a relational expression, and *expr-3* is an increment or decrement of some manner. For example, the following code counts from 0 to 99, printing each number along the way:

```
int i;
for (i= 0; i < 100; i++)
  printf("%d\n", i);
```

Break

Use of the `break` provides an early exit from a `while` or a `for` loop.

LCD Screen Printing

IC has a version of the C function `printf` for formatted printing to the LCD screen.

The syntax of `printf` is the following:

```
printf(format-string, [arg-1] , ... , [arg-N] )
```

This is best illustrated by some examples.

Printing Examples

Example 1: Printing a message. The following statement prints a text string to the screen.

```
printf("Hello, world!\n");
```

In this example, the format string is simply printed to the screen.

The character `\n` at the end of the string signifies *end-of-line*. When an end-of-line character is printed, the LCD screen will be cleared when a subsequent character is printed. Thus, most `printf` statements are terminated by a `\n`.

Example 2: Printing a number. The following statement prints the value of the integer variable `x` with a brief message.

```
printf("Value is %d\n", x);
```

The special form `%d` is used to format the printing of an integer in decimal format.

Example 3: Printing a number in binary. The following statement prints the value of the integer variable `x` as a binary number.

```
printf("Value is %b\n", x);
```

The special form `%b` is used to format the printing of an integer in binary format. Only the *low byte* of the number is printed.

Example 4: Printing a floating point number. The following statement prints the value of the floating point variable `n` as a floating point number.

```
printf("Value is %f\n", n);
```

The special form `%f` is used to format the printing of floating point number.

Example 5: Printing two numbers in hexadecimal format.

```
printf("A=%x  B=%x\n", a, b);
```

The form `%x` formats an integer to print in hexadecimal.

Formatting Command Summary

`%d`

Type: `int` Description: decimal number

`%x`

Type: `int` Description: hexadecimal number

`%b`

Type: `int` Description: low byte as binary number

`%c`

Type: `int` Description: low byte as ASCII character

`%f`

Type: `float` Description: floating point number

`%s`

Type: `char array` Description: char array (string)

Format Command	Data Type	Description
`%d`	**int**	**decimal number**
`%x`	**int**	**hexadecimal number**
`%b`	**int**	**low byte as binary number**
`%c`	**int**	**low byte as ASCII character**
`%f`	**float**	**floating point number**
`%s`	**char array**	**char array (string)**

Special Notes

- The final character position of the LCD screen is used as a system "heartbeat." This character continuously blinks between a large and small heart when the board is operating properly. If the character stops blinking, the board has failed.
- Characters that would be printed beyond the final character position are truncated.
- When using a two-line display, the `printf()` command treats the display as a single longer line.
- Printing of long integers is not presently supported.

Preprocessor

The preprocessor processes a file before it is sent to the compiler. The IC preprocessor allows definition of macros, and conditional compilation of sections of code. Using preprocessor macros for constants and function macros can make IC code more efficient as well as easier to read. Using `#if` to conditionally compile code can be very useful, for instance, for debugging purposes.

Preprocessor Macros

Preprocessor macros are defined by using the `#define` preprocessor directive at the start of a line. If a macro is defined anywhere in any of the

files loaded into IC, it can be used anywhere in any file. The following example shows how to define preprocessor macros.

```
#define RIGHT_MOTOR 0
#define LEFT_MOTOR  1

#define GO_RIGHT(power) (motor(RIGHT_MOTOR,
(power)))
#define GO_LEFT(power)  (motor(LEFT_MOTOR,(power)))

#define GO(left,right) {GO_LEFT(left); GO_RIGHT
(right);}

void main()
{
   GO(0,0);
}
```

Preprocessor macro definitions start with the `#define` directive at the start of a line, and continue to the end of the line. After `#define` is the name of the macro, such as `RIGHT_MOTOR`. If there is a parenthesis directly after the name of the macro, such as the `GO_RIGHT` macro has above, then the macro has arguments. The `GO_RIGHT` and `GO_LEFT` macros each take one argument. The `GO` macro takes two arguments. After the name and the optional argument list is the body of the macro.

Each time a macro is invoked, it is replaced with its body. If the macro has arguments, then each place the argument appears in the body is replaced with the actual argument provided.

Invocations of macros without arguments look like global variable references. Invocations of macros with arguments look like calls to functions. To an extent, this is how they act. However, macro replacement happens before compilation, whereas global references and function calls happen at run time. Also, function calls evaluate their arguments before they are called, whereas macros simply perform text replacement. For example, if the actual argument given to a macro contains a function call, and the macro instantiates its argument more than once in its body, then the function would be called multiple times, whereas it would only be called once if it were being passed as a function argument instead.

Appropriate use of macros can make IC programs and easier to read. It allows constants to be given symbolic names without requiring storage and access time as a global would. It also allows macros with arguments to be

used in cases when a function call is desirable for abstraction, without the performance penalty of calling a function.
Macros defined in files can be used at the command line. Macros can also be defined at the commandline to be used in interactively, but these will not affect loads or compilation. To obtain a list of the currently defined macros, type `list defines` at the IC prompt.

Conditional compilation

It is sometimes desirable to conditionally compile code. The primary example of this is that you may want to perform debugging output sometimes, and disable it at other times. The IC preprocessor provides a convenient way of doing this by using the `#ifdef` directive.

```
void go_left(int power)
{
   GO_LEFT(power);
#ifdef DEBUG
   printf("Going Left\n");
   beep();
#endif
}
```

In this example, when the macro `DEBUG` is defined, the debugging message "Going Left" will be printed and the board will beep each time `go_left` is called. If the macro is not defined, the message and beep will not happen. Each `#ifdef` must be follwed by an `#endif` at the end of the code which is being conditionally compiled. The macro to be checked can be anything, and `#ifdef` blocks may be nested.
Unlike regular C preprocessors, macros cannot be conditionally defined. If a macro definition occurs inside an `#ifdef` block, it will be defined regardless of whether the `#ifdef` evaluates to true or false. The compiler will generate a warning if macro definitions occur within an `#ifdef` block.
The `#if`, `#else`, and `#elif` directives are also available, but are outside the scope of this document. Refer to a C reference manual for how to use them.

Comparison with regular C preprocessors

The way in which IC deals with loading multiple files is fundamentally different from the way in which it is done in standard C. In particular, when using standard C, files are compliled completely independently of each other, then linked together. In IC, on the other hand, all files are compiled together. This is why standard C needs function prototypes and `extern` global

definitions in order for multiple files to share functions and globals, while IC does not.

In a standard C preprocessor, preprocessor macros defined in one C file cannot be used in another C file unless defined again. Also, the scope of macros is only from the point of definition to the end of the file. The solution then is to have the prototypes, `extern` declarations, and macros in header files which are then included at the top of each C file using the `#include` directive. This style interacts well with the fact that each file is compiled independent of all the others.

However, since declarations in IC do not file scope, it would be inconsistent to have a preprocessor with file scope. Therefore, for consistency it was desirable to give IC macros the same behavior as globals and functions. Therefore, preprocessor macros have global scope. If a macro is defined anywhere in the files loaded into IC, it is defined everywhere. Therefore, the `#include` and `#undef` directives did not seem to have any appropriate purpose, and were accordingly left out.

The fact that `#define` directives contained within `#if` blocks are defined regardless of whether the `#if` evaluates to be true or false is a side effect of making the preprocessor macros have global scope.

Other than these modifications, the IC preprocessor should be compatible with regular C preprocessors.

The IC Library File

Library files provide standard C functions for interfacing with hardware on the robot controlier board. These functions are written either in C or as assembly language drivers. Library files provide functions to do things like control motors, make tones, and input sensors values.

IC automatically loads the library file every time it is invoked. Depending on which 6811 board is being used, a different library file will be required. IC may be configured to load different library files as its default; the Windows and Mac versions of IC will automatically load the correct library for the board you're using at the moment.

This documentation covers the libraries for the Handy Board and 6.270 board (Rev. 2.21) only; if you have another board, see your owner's manual for documentation.

To understand better how the library functions work, study of the library file source code is recommended. The main library file for the Rev. 2.21 6.270 Board is named `lib_r22.lis`; for the Handy Board, the main library file is named `lib_hb.lis`.

Output Control

DC Motors

The Handy Board has four motor ports, numbered from 0 to 3. The 6.270 board has 6 motor ports, numbered from 0 to 5; ports for motors 0 to 3 are located on the Microprocessor Board while motors 4 and 5 are located on the Expansion Board.

Motor may be set in a "forward" direction (corresponding to the green motor LED being lit) and a "backward" direction (corresponding to the motor red LED being lit).

The functions `fd(int m)` and `bk(int m)` turn motor `m` on forward or backward, respectively, at full power. The function `off(int m)` turns motor `m` off.

The power level of motors may also be controlled. This is done in software by turning a motor on and off rapidly (a technique called *pulse-width modulation*. The `motor(int m, int p)` function allows control of a motor's power level. Powers range from `100` (full on in the forward direction) to `-100` (full on the the backward direction). The system software actually only controls motors to seven degrees of power, but argument bounds of `-100` and `+100` are used.

`void fd(int m)`

Turns motor `m` on in the forward direction. Example: `fd(3);`

`void bk(int m)`

Turns motor `m` on in the backward direction. Example: `bk(1);`

`void off(int m)`

Turns off motor `m`. Example: `off(1);`

`void alloff()`

`void ao()`

Turns off all motors. `ao` is a short form for `alloff`.

`void motor(int m, int p)`

Turns on motor `m` at power level `p`. Power levels range from `100` for full on forward to `-100` for full on backward.

Servo Motor

Servos are motors with internal position feedback which you can accurately command to a given orientation. Servos will actively seek to move to and remain at the orientation they are commanded to go to. Servos are useful for aiming sensors or moving actuators through a limited arc. They are generally able to sweep through about 180 degrees and no more.

Library routines allows control of a single servo motor. The servo motor has a three-wire connection: power, ground, and control.

On a 6.270 board, there is a dedicated connection for the servo on the main board at the top of the bank of connectors which are above and to the left of the main power switch. A three prong connector with ground on the left, power in the middle, and control on the right should be used to plug the servo into its connector. So long as you are sure to get power in the middle, the servo will not be damaged by plugging it in backwards, but will simply not work until it is plugged in properly.

See *The Handy Board Technical Reference* for information on how to attach a servo motor to a Handy Board. To use these functions on a Handy Board, you must explicitly load the files `servo.c` and `servo.icb`.

The position of the servo motor shaft is controlled by a rectangular waveform that is generated on the **A7** pin. The duration of the positive pulse of the waveform determines the position of the shaft. The acceptable width of the pulse varies for different models of servos, but is approximately 700 timer cycles minimum and 4000 timer cycles maximum, where the 6811's timer runs at 2MHz. The pulse is repeated approximately every 20 milliseconds.

`void servo_on()`

Turns the servo signal on. You must call this function before the servo will move.

`void servo_off()`

Turns servo signal off. The servo will no longer try to move to any particular position and will move freely. When you are not actively using the servo, turning it off will save power and processor cycles.

`int servo(int period)`

Sets the high time of the servo signal to `period` timer cycles so long as that falls within the acceptable range for the servo. Otherwise it trunctates the value to the closest the servo is physically able to go to. It returns the thresholded version of the period you gave it. Remember that servos have a finite reaction time which, while very fast to human senses of time, is very slow to a processor. If you are resetting the servo angle in a tight loop it may well never catch up with you.

`int servo_rad(float angle)`

Sets the commanded orientation of the servo to approximately the angle in radians that it is given and returns the pulse width in timer counts which the servo was actually commanded with. The minimum pulse width is defined to be zero radians and the maximum is defined to be `pi` radians.

`int servo_deg(float angle)`

Sets the commanded orientation of the servo to approximately the angle in degrees that it is given and returns the pulse width in timer counts which

the servo was actually commanded with. The minimum pulse width is defined to be zero degrees and the maximum is defined to be 180 degrees.

`int radian_to_pulse(float angle)`

Converts the angle given in radians to the corresponding pulse width in timer counts. Input range is 0.0 to 3.14.

`int degree_to_pulse(float angle)`

Converts the angle given in degrees to the corresponding pulse width in timer counts. Input range is 0.0 to 180.0.

Unidirectional Drivers

LED Drivers

On the 6.270 board, there are two output ports located on the Expansion Board that are suitable for driving LEDs or other small loads. These ports draw their power from the motor battery and hence will only work when that battery is connected.

The Handy Board does not include these ports.

The following commands are used to control the LED ports:

`void led_out0(int s)`

Turns on LED0 port if `s` is non-zero; turns it off otherwise.

`void led_out1(int s)`

Turns on LED1 port if `s` is non-zero; turns it off otherwise.

Expansion Board Motor Ports

On the 6.270 board, motor ports 4 and 5, located on the Expansion Board, may also be used to control unidirectional devices, such as a solenoid, lamp, or a motor that needs to be driven in one direction only. Each of the two motor ports, when used in this fashion, can independently control two such devices.

As mentioned above, the Handy Board does not have these ports.

To use the ports unidirectionally, the two-pin header directly beneath the motor 4 and 5 LEDs is used.

`void motor4_left(int s)`

Turns on left side of motor 4 port if `s` is non-zero; turns it off otherwise.

`void motor4_right(int s)`

Turns on right side of motor 4 port if `s` is non-zero; turns it off otherwise.

`void motor5_left(int s)`

Turns on left side of motor 5 port if `s` is non-zero; turns it off otherwise.

`void motor5_right(int s)`

Turns on right side of motor 5 port if `s` is non-zero; turns it off otherwise.

Sensor Input

Digital Input

`int digital(int p)`

Returns the value of the sensor in sensor port `p`, as a true/false value (1 for true and 0 for false). Sensors are expected to be *active low*, meaning that they are valued at zero volts in the active, or true, state. Thus the library function returns the inverse of the actual reading from the digital hardware: if the reading is zero volts or logic zero, the `digital()` function will return true.

The 6.270 board has four DIP switches on the expansion board. The Handy Board does not have DIP switches.

`int dip_switch(int sw)`

Returns value of DIP switch `sw` on the expansion board. Switches are numbered from 1 to 4 as per labelling on the actual switch. Result is 1 if the switch is in the position labelled "on," and 0 if not.

`int dip_switches()`

Returns value on DIP switches as a four-bit binary number. Left-most switch is most significant binary digit. "On" position is binary one.

Both the Handy Board and the 6.270 board have two buttons. On the Handy Board, the buttons are labelled **`Start`** and **`Stop`**; on the 6.270 board, the buttons are labelled **`Escape`** and **`Choose`**.

`int stop_button() (or choose_button() for the 6.270 board)`

Returns value of button labelled **`Stop`** (or **`Choose`**): 1 if pressed and 0 if released. Example:

```
/*  wait until stop button pressed  */
while (!stop_button()) {}
```

`int start_button() (or escape_button())`

Returns value of button labelled **`Start`** (or **`Escape`**). Example:

```
/*  wait for button to be pressed; then
    wait for it to be released so that
    button press is debounced              */
while (!start_button()) {}
while (start_button()) {}
```

The Handy Board has two additional convenience functions; these currently do not have analogs for the 6.270 board.

`void stop_press()`

Waits for the **`Stop`** button to be pressed, then released. Then issues a short beep and returns. The code for `stop_press()` is as follows:

```
while (!stop_button());
while (stop_button());
beep();
```

`void start_press()`

Like `stop_press()`, but for the **`Start`** button.

Analog Inputs

`int analog(int p)`

Returns value of sensor port numbered `p`. Result is integer between 0 and 255. If the `analog()` function is applied to a port that is implemented digitally in hardware, then the value 255 is returned if the *hardware* digital reading is 1 (as if a digital switch is open, and the pull up resistors are causing a high reading), and the value 0 is returned if the *hardware* digital reading is 0 (as if a digital switch is closed and pulling the reading near ground). Ports are numbered as marked.

`int knob() (or frob_knob() on the 6.270 board)`

Returns a value from 0 to 255 based on the position of a potentiometer. On the 6.270 board, the potentiometer is labelled **`frob knob`**.

The 6.270 board contains experimental circuitry for detecting the force being applied by a motor. The Handy Board does not contain this circuitry.

`int motor_force(int m)`

Returns value of analog input sensing current level through motor `m`. Result is integer between 0 and 255, but typical readings range from about 40 (low force) to 100 (high force). The force-sensing circuitry functions properly only when motors are operated at full speed. The circuit returns invalid results when motors are pulse-width modulated because of spikes that occur in the feedback path. The force-sensing circuitry is implemented for motors 0 through 3.

Infrared Subsystem

The infrared subsystem is composed of two parts: an infrared transmitter, and infrared receivers. Software is provided to control transmission frequency and detection of infrared light at two frequencies.

Infrared Transmission

`void ir_transmit_on()`

Enables transmission of infrared light through **`ir out`** port.

`void ir_transmit_off()`

Disables transmission of infrared light through **`ir out`** port.

`void set_ir_transmit_period(int period)`

Sets infrared transmission period. `period` determines the delay in half-microseconds between transitions of the infrared waveform. If `period` is set to 10,000, a frequency of 100 Hz. will be generated. If `period` is set to 8,000, a frequency of 125 Hz. will be generated. The decoding software

is capable of detecting transmissions on either of these two frequencies only.

`void set_ir_transmit_frequency(int frequency)`

Sets infrared transmission frequency. `frequency` is measured in hertz. Upon a reset condition, the infrared transmission frequency is set for 100 Hz. and is disabled.

Infrared Reception

In a typical 6.270 application, one robot will be broadcasting infrared at 100 Hz. and will set its detection system for 125 Hz. The other robot will do the opposite. Each robot must physically shield its IR sensors from its own light; then each robot can detect the emissions of the other.

The infrared reception software employs a *phase-locked loop* to detect infrared signals modulated at a particular frequency. This program generates an internal squarewave at the desired reception frequency and attempts to lock this squarewave into synchronization with a waveform received by an infrared sensor. If the error between the internal wave and the external wave is below some threshold, the external wave is considered "detected." The software returns as a result the number of consecutive detections for each of the infrared sensor inputs.

While enabled, the infrared reception software requires a great deal of processor time. Therefore, it is desirable to disable the IR reception whenever it is not being actively used.

Up to four infrared sensors may be used. These are plugged into positions 4 through 7 of the digital input port. These ports and the remainder of the digital input port may be used without conflict for standard digital input while the infrared detection software is operating.

The following library functions control the infrared detection system:

`void ir_receive_on()`

Enables the infrared reception software. The default is disabled. When the software is enabled, between 20% and 30% of the 6811 processor time will be spent performing the detection function; therefore it should only be enabled if it is being used. *You must wait at least 100 milliseconds after starting the reception before the data is valid.*

`void ir_receive_off()`

Disables the infrared reception software.

`void set_ir_receive_frequency(int f)`

Sets the operating frequency for the infrared reception software. `f` should be 100 for 100 Hz. or 125 for 125 Hz. Default is 100.

`int ir_counts(int p)`

Returns number of consecutive squarewaves at operating frequency detected from port `p` of the digital input port. Result is number from 0 to 255. `p` must be 4, 5, 6, or 7 Random noise can cause spurious readings of 1 or 2 detections. The return value of `ir_counts()` should be greater than three before it is considered the result of a valid detection. *You must wait at least 100 milliseconds after starting the reception before the* `ir_counts()` `data is valid.`

Shaft Encoders

Shaft encoders can be used to count the number of times a wheel spins, or in general the number of digital pulses seen by an input. Two types of shaft encoders can be made using 6.270 sensors: optical encoders which use optical switches whose beam is periodically broken by a slotted wheel, or magnetic encoders which use Hall effect sensors which change state when a magnet on a shaft rotates past.

Shaft encoders are implemented using the input timer capture feature on the 6811. Therefore processing time is only used when a pulse is actually being recorded, and even very fast pulses can be counted. Digital ports 0 and 1 are two input capture channels which are available for use on the Handy Board and on the 6.270 board, so two channels of shaft encoding are supported. The encoding software keeps a running count of the number of pulses each enabled encoder has seen. The number of counts is set to 0 when a chanel is first enabled and when a user resets that channel. Because the counters are only 16-bits wide, they will overflow and the value will appear negative after 32,767 counts have been accumulated without a reset.

As shaft encoders are an optional feature, the library routines which read them are not loaded on start up.

In order to load the following routines for use in your programs, load the file `encoders.lis`. This file is in the standard IC library directory.

The actions of the shaft encoders are commanded and the results are read using the following routines. The argument `encoder` to each of the routines specifies which shaft encoder the function should affect. This value should be 0 for digital port 0 or one for digital port 1. Arguments out of the range 0 to 1 have no useful effect.

```
void enable_encoder(int encoder)
```

Enables the given encoder to start counting pulses and resets its counter to zero. By default encoders start in the disabled state and must be enabled before they start counting.

```
void disable_encoder(int encoder)
```

Disables the given encoder and prevents it from counting. Each shaft encoder uses processing time every time it receives a pulse while enabled, so they should be disabled when you no longer need the encoder's data.

`void reset_encoder(int encoder)`

Resets the counter of the given encoder to zero. For an enabled encoder, it is more efficient to reset its value than to use `enable_encoder()` to clear it.

`int read_encoder(int encoder)`

Returns the number of pulses counted by the given encoder since it was enabled or since the last reset, whichever was more recent.

Time Commands

System code keeps track of time passage in milliseconds. Library functions are provided to allow dealing with time in milliseconds (using long integers), or seconds (using floating point numbers).

`void reset_system_time()`

Resets the count of system time to zero milliseconds.

`long mseconds()`

Returns the count of system time in milliseconds. Time count is reset by hardware reset (i.e., pressing reset switch on board) or the function `reset_system_time()`. `mseconds()` is implemented as a C primitive (not as a library function).

`float seconds()`

Returns the count of system time in seconds, as a floating point number. Resolution is one millisecond.

`void sleep(float sec)`

Waits for an amount of time equal to or slightly greater than `sec` seconds. `sec` is a floating point number. Example:

```
/*  wait for 1.5 seconds  */
sleep(1.5);
```

`void msleep(long msec)`

Waits for an amount of time equal to or greater than `msec` milliseconds. `msec` is a long integer. Example:

```
/*  wait for 1.5 seconds  */
msleep(1500L);
```

Tone Functions

Two simple commands are provided for producing tones on the standard beeper.

`void beep()`

Produces a tone of 500 Hertz for a period of 0.3 seconds. Returns when the tone is finished.

`void tone(float frequency, float length)`

Produces a tone at pitch `frequency` Hertz for `length` seconds. Returns when the tone is finished. Both `frequency` and `length` are floats. In addition to the simple tone commands, the following functions can be used asynchronously to control the beeper driver.

`void set_beeper_pitch(float frequency)`

Sets the beeper tone to be `frequency` Hz. The subsequent function is then used to turn the beeper on.

`void beeper_on()`

Turns on the beeper at last frequency selected by the former function. The beeper remains on until the `beeper_off` function is executed.

`void beeper_off()`

Turns off the beeper.

Menuing and Diagnostics Functions

These functions are not loaded automatically, but they are available for you to use if you wish. They currently work only on the 6.270 board, but could probably be ported to the Handy Board without too much trouble. They provide a standardized user interface for prompting users for input using the **`Choose`** and **`Select`** buttons and the frob knob. You may wish to use this library for debugging the state of your robot while away from the terminal or for changing thresholds or gains on the fly.

menu.c

Load `menu.c` to be able to use these functions.

`int select_string(char choices[][],int n)`

Interactively selects a string from an array of string (two-dimensional array of characters) of length n and returns an integer when a button is pressed. If the button pressed was **`choose`**, it returns the index into the array of the selected string, otherwise it returns -1. Example of use:

```
char a[3][14]={"Analog Port ","Digital
Port ","Quit"};
int port,index=select_string(a,3);

if(index>-1 && index<2)
   port=select_int_value(a[index],0,27);
```

`int select_int_value(char s[],int min_val,int max_val)`

`float select_float_value(char s[],float min_val,float max_val)`

Interactively selects and returns a number between `min_val` and `max_val` which is selected by adjusting the frob knob until the

appropriate value is displayed then pressing a button. If **escape** was pressed, returns -1 (or -1.0) regardless of the value chosen. Otherwise returns the chosen value. Remember that the frob knob only returns one of 255 values, so if the range is greater than that not all values will be possible choices.

`int chosen_button()`

Checks the user buttons and returns CHOOSE_B if **choose** is pressed, ESCAPE_B if **escape** is pressed, and NEITHER_B if neither button is pressed. If both buttons are pressed, **choose** has priority.

`int wait_button(int mode)`

Waits for either user button to execute the action appropriate to `mode` then returns which button was pressed. The choices for `mode` are: DOWN_B -- wait until either button is pressed; UP_B -- wait until no buttons are pressed; CYCLE_B -- wait until a button is depressed and then all depressed buttons are released before returning.

diagnostic.c

Load `menu.c` and `diagnostic.c` to be able to use these functions. You can easily copy `diagnostic.c` and modify the `control_panel` function to call your own routines.

`void control_panel()`

General purpose control panel to let you view inputs, frob outputs, or set A to D thresholds. Pressing **escape** from the main menu or selecting "Quit" exits the control panel.

`int view_average_port(int port,int samples)`

Displays the analog reading of the given port until a button is pressed. If the button is **choose**, it then samples the reading at the given port, averages `samples` readings together, then prints and returns the average result. If the button pushed was **ecape**, it returns -1.

`void view_inputs()`

General purpose input status viewer using the standard menuing routines to show digital inputs, analog inputs, frob knob, dip switches, and motor force inputs. Pressing **escape** at any time exits the viewer.

`void frob_outputs()`

General purpose output frobber. Uses the standard menuing routines to let you control the motors, led outputs, ir output, and the servo. Pressing **escape** from the main menu or selecting "Quit" exits the frobber.

Multi-Tasking

Overview

One of the most powerful features of IC is its multi-tasking facility. Processes can be created and destroyed dynamically during run-time.
Any C function can be spawned as a separate task. Multiple tasks running the same code, but with their own local variables, can be created.
Processes communicate through global variables: one process can set a global to some value, and another process can read the value of that global.
Each time a process runs, it executes for a certain number of *ticks*, defined in milliseconds. This value is determined for each process at the time it is created. The default number of ticks is five; therefore, a default process will run for 5 milliseconds until its "turn" ends and the next process is run. All processes are kept track of in a *process table*; each time through the table, each process runs once (for an amount of time equal to its number of ticks).
Each process has its own *program stack.* The stack is used to pass arguments for function calls, store local variables, and store return addresses from function calls. The size of this stack is defined at the time a process is created. The default size of a process stack is 256 bytes.
Processes that make extensive use of recursion or use large local arrays will probably require a stack size larger than the default. Each function call requires two stack bytes (for the return address) plus the number of argument bytes; if the function that is called creates local variables, then they also use up stack space. In addition, C expressions create intermediate values that are stored on the stack.
It is up to the programmer to determine if a particular process requires a stack size larger than the default. A process may also be created with a stack size *smaller* than the default, in order to save stack memory space, if it is known that the process will not require the full default amount.
When a process is created, it is assigned a unique *process identification number* or *pid*. This number can be used to kill a process.

Creating New Processes

The function to create a new process is `start_process`.
`start_process` takes one mandatory argument--the function call to be started as a process. There are two optional arguments: the process's number of ticks and stack size. (If only one optional argument is given, it is assumed to be the ticks number, and the default stack size is used.)
`start_process` has the following syntax:

```
int start_process(function-call(...),[TICKS],
[STACK-SIZE])
```

`start_process` returns an integer, which is the process ID assigned to the new process.

The function call may be any valid call of the function used. The following code shows the function `main` creating a process:

```
void check_sensor(int n)
{
  while (1)
    printf("Sensor %d is %d\n", n, digital(n));
}

void main()
{
  start_process(check_sensor(2));
}
```

Normally when a C functions ends, it exits with a return value or the "void" value. If a function invoked as a process ends, it "dies," letting its return value (if there was one) disappear. (This is okay, because processes communicate results by storing them in globals, not by returning them as return values.) Hence in the above example, the `check_sensor` function is defined as an infinite loop, so as to run forever (until the board is reset or a `kill_process` is executed).

Creating a process with a non-default number of ticks or a non-default stack size is simply a matter of using `start_process` with optional arguments; e.g.

```
start_process(check_sensor(2), 1, 50);
```

will create a `check_sensor` process that runs for 1 milliseconds per invocation and has a stack size of 50 bytes (for the given definition of `check_sensor`, a small stack space would be sufficient).

Destroying Processes

The `kill_process` function is used to destroy processes. Processes are destroyed by passing their process ID number to `kill_process`, according to the following syntax:

```
int kill_process(int pid);
```

`kill_process` returns a value indicating if the operation was successful. If the return value is `0`, then the process was destroyed. If the return value is `1`, then the process was not found.

The following code shows the `main` process creating a `check_sensor` process, and then destroying it one second later:

```
void main()
{
  int pid;
```

```
  pid= start_process(check_sensor(2));
  sleep(1.0);
  kill_process(pid);
}
```

Process Management Commands

IC has two commands to help with process management. The commands only work when used at the IC command line. They are not C functions that can be used in code.

`kill_all`

kills all currently running processes.

`ps`

prints out a list of the process status. The following information is presented: process ID, status code, program counter, stack pointer, stack pointer origin, number of ticks, and name of function that is currently executing.

Process Management Library Functions

The following functions are implemented in the standard C library.

`void hog_processor()`

Allocates an additional 256 milliseconds of execution to the currently running process. If this function is called repeatedly, the system will wedge and only execute the process that is calling `hog_processor()`. Only a system reset will unwedge from this state. Needless to say, this function should be used with extreme care, and should not be placed in a loop, unless wedging the machine is the desired outcome.

`void defer()`

Makes a process swap out immediately after the function is called. Useful if a process knows that it will not need to do any work until the next time around the scheduler loop. `defer()` is implemented as a C built-in function.

Floating Point Functions

In addition to basic floating point arithmetic (addition, subtraction, multiplication, and division) and floating point comparisons, a number of exponential and transcendental functions are built in to IC:

`float sin(float angle)`

Returns sine of `angle`. Angle is specified in radians; result is in radians.

`float cos(float angle)`

Returns cosine of `angle`. Angle is specified in radians; result is in radians.

`float tan(float angle)`

Returns tangent of `angle`. Angle is specified in radians; result is in radians.

`float atan(float angle)`

Returns arc tangent of `angle`. Angle is specified in radians; result is in radians.

`float sqrt(float num)`

Returns square root of `num`.

`float log10(float num)`

Returns logarithm of `num` to the base 10.

`float log(float num)`

Returns natural logarithm of `num`.

`float exp10(float num)`

Returns 10 to the `num` power.

`float exp(float num)`

Returns *e* to the `num` power.

`(float) a ^ (float) b`

Returns `a` to the `b` power.

Memory Access Functions

IC has primitives for directly examining and modifying memory contents. These should be used with care as it is easy to corrupt memory and crash the system using these functions.

`int peek(int loc)`

Returns the byte located at address `loc`.

`int peekword(int loc)`

Returns the 16-bit value located at address `loc` and `loc+1`. `loc` has the most significant byte, as per the 6811 16-bit addressing standard.

`void poke(int loc, int byte)`

Stores the 8-bit value `byte` at memory address `loc`.

`void pokeword(int loc, int word)`

Stores the 16-bit value `word` at memory addresses `loc` and `loc+1`.

`void bit_set(int loc, int mask)`

Sets bits that are set in `mask` at memory address `loc`.

`void bit_clear(int loc, int mask)`

Clears bits that are set in `mask` at memory address `loc`.

Error Handling

There are two types of errors that can happen when working with IC: *compile-time* errors and *run-time* errors.

Compile-time errors occur during the compilation of the source file. They are indicative of mistakes in the C source code. Typical compile-time errors result from incorrect syntax or mis-matching of data types.

Run-time errors occur while a program is running on the board. They indicate problems with a valid C form when it is running. A simple example would be a divide-by-zero error. Another example might be running out of stack space, if a recursive procedure goes too deep in recursion.
These types of errors are handled differently, as is explained below.

Compile-Time Errors

When compiler errors occur, an error message is printed to the screen. All compile-time errors must be fixed before a file can be downloaded to the board.

Run-Time Errors

When a run-time error occurs, an error message is displayed on the LCD screen indicating the error number. If the board is hooked up to IC when the error occurs, a more verbose error message is printed on the terminal.
Here is a list of the run-time error codes:

1
no stack space for `start_process()`
2
no process slots remaining
3
array reference out of bounds
4
stack overflow error in running process
5
operation with invalid pointer
6
floating point underflow
7
floating point overflow
8
floating point divide-by-zero
9
number too small or large to convert to integer
10
tried to take square root of negative number
11
tangent of 90 degrees attempted
12
log or ln of negative number or zero
15

floating point format error in printf

16

integer divide-by-zero

Binary Programs

With the use of a customized 6811 assembler program, IC allows the use of machine language programs within the C environment. There are two ways that machine language programs may be incorporated:

Programs may be called from C as if they were C functions.

Programs may install themselves into the interrupt structure of the 6811, running repetitiously or when invoked due to a hardware or software interrupt.

When operating as a function, the interface between C and a binary program is limited: a binary program must be given one integer as an argument, and will return an integer as its return value. However, programs in a binary file can declare any number of global integer variables in the C environment. Also, the binary program can use its argument as a pointer to a C data structure.

The Binary Source File

Special keywords in the source assembly language file (or module) are used to establish the following features of the binary program:

Entry point

The entry point for calls to each program defined in the binary file.

Initialization entry point

Each file may have one routine that is called automatically upon a reset condition. (See section Local and Global Scopes, for a discussion of global variable initialization.) This initialization routine is particularly useful for programs which will function as interrupt routines.

C variable definitions

Any number of two-byte C integer variables may be declared within a binary file. When the module is loaded into IC, these variables become defined as globals in C.

To explain how these features work, let's look at a sample IC binary source program:

```
/* Sample icb file */

/* origin for module and variables */
        ORG     MAIN_START
```

```
/* program to return twice the argument passed to
us */
subroutine_double:
        ASLD
        RTS

/* declaration for the variable "foo" */
variable_foo:
        FDB     55

/* program to set the C variable "foo" */
subroutine_set_foo:
        STD     variable_foo
        RTS

/* program to retrieve the variable "foo" */
subroutine_get_foo:
        LDD     variable_foo
        RTS

/* code that runs on reset conditions */
subroutine_initialize_module:
        LDD     #69
        STD     variable_foo
        RTS
```

The first statement of the file ("`ORG MAIN_START`") declares the start of the binary programs. This line must precede the code itself itself.

The entry point for a program to be called from C is declared with a special form beginning with the text `subroutine_`. In this case, the name of the binary program is `double`, so the label is named `subroutine_double`. As the comment indicates, this is a program that will double the value of the argument passed to it.

When the binary program is called from C, it is passed one integer argument. This argument is placed in the 6811's D register (also known as the "Double Accumulator") before the binary code is called.

The `double` program doubles the number in the D register. The `ASLD` instruction ("Arithmetic Shift Left Double [Accumulator]") is equivalent to multiplying by 2; hence this doubles the number in the D register.

The `RTS` instruction is "Return from Subroutine." All binary programs must exit using this instruction. When a binary program exits, the value in the D

register is the return value to C. Thus, the `double` program doubles its C argument and returns it to C.

Declaring Variables in Binary Files

The label `variable_foo` is an example of a special form to declare the name and location of a variable accessable from C. The special label prefix "`variable_`" is followed the name of the variable, in this case, "`foo`."
This label must be immediately followed by the statement `FDB <number>`. This is an assembler directive that creates a two-byte value (which is the initial value of the variable).
Variables used by binary programs must be declared in the binary file. These variables then become C globals when the binary file is loaded into C.
The next binary program in the file is named "`set_foo`." It performs the action of setting the value of the variable `foo`, which is defined later in the file. It does this by storing the D register into the memory contents reserved for `foo`, and then returning.
The next binary program is named "`get_foo`." It loads the D register from the memory reserved for `foo` and then returns.

Declaring an Initialization Program

The label `subroutine_initialize_module` is a special form used to indicate the entry point for code that should be run to initialize the binary programs. This code is run upon standard reset conditions: program download, hardware reset, or running of the `main()` function.
In the example shown, the initialization code stores the value 69 into the location reserved for the variable `foo`. This then overwrites the 55 which would otherwise be the default value for that variable.
Initialization of globals variables defined in an binary module is done differently than globals defined in C. In a binary module, the globals are initialized to the value declared by the `FDB` statement only when the code is downloaded to the 6811 board (not upon reset or running of main, like normal globals).
However, the initialization routine is run upon standard reset conditions, and can be used to initialize globals, as this example has illustrated.

Interrupt-Driven Binary Programs

Interrupt-driven binary programs use the initialization sequence of the binary module to install a piece of code into the interrupt structure of the 6811.
The 6811 has a number of different interrupts, mostly dealing with its on-chip hardware such as timers and counters. One of these interrupts is used by the 6.270 software to implement time-keeping and other periodic functions (such

as LCD screen management). This interrupt, dubbed the "System Interrupt," runs at 1000 Hertz.
Instead of using another 6811 interrupt to run user binary programs, additional programs (that need to run at 1000 Hz. or less) may install themselves into the System Interrupt. User programs would be then become part of the 1000 Hz interrupt sequence.
This is accomplished by having the user program "intercept" the original 6811 interrupt vector that points to 6.270 interrupt code. This vector is made to point at the user program. When user program finishes, it jumps to the start of the 6.270 interrupt code.

The top picture depicts the interrupt structure before user program installation. The 6811 vector location points to system software code, which terminates in a "return from interrupt" instruction.
The bottom picture illustrates the result after the user program is installed. The 6811 vector points to the user program, which exits by jumping to the system software driver. This driver exits as before, with the RTI instruction. Multiple user programs could be installed in this fashion. Each one would install itself ahead of the previous one. Some standard 6.270 library functions, such as the shaft encoder software, is implemented in this fashion.

```
* icb file.  "sysibeep.asm"

*
*    example of code installing itself into
*    SystemInt 1000 Hz interrupt
*
*    Fred Martin
*    Thu Oct 10 21:12:13 1991
*

#include <6811regs.asm>
```

```
        ORG     MAIN_START

subroutine_initialize_module:

#include <ldxibase.asm>
* X now has base pointer to interrupt vectors
($FF00 or $BF00)

* get current vector; poke such that when we
finish, we go there
        LDD     TOC4INT,X               ; SystemInt on
TOC4
        STD     interrupt_code_exit+1

* install ourself as new vector
        LDD     #interrupt_code_start
        STD     TOC4INT,X

        RTS

* interrupt program begins here
interrupt_code_start:
* frob the beeper every time called
        LDAA    PORTA
        EORA    #%00001000      ; beeper bit
        STAA    PORTA

interrupt_code_exit:
        JMP     $0000    ; this value poked in by
init routine
```

The above program installs itself into the System Interrupt. This program toggles the signal line controlling the piezo beeper every time it is run; since the System Interrupt runs at 1000 Hz., this program will create a continous tone of 500 Hz.

The first line after the comment header includes a file named "`6811regs.asm`". This file contains equates for all 6811 registers and interrupt vectors; most binary programs will need at least a few of these. It is simplest to keep them all in one file that can be easily included. (This and other files included by the `as11` assembler are located in the assembler's default library directory, which is `/mit/6.270/lib/as11/` on the MIT Athena system.)

The `subroutine_initialize_module` declaration begins the initialization portion of the program. The file " `ldxibase.asm`" is then included. This file contains a few lines of 6811 assembler code that perform the function of determining the base pointer to the 6811 interrupt vector area, and loading this pointer into the 6811 X register.

The following four lines of code install the interrupt program (beginning with the label `interrupt_code_start`) according to the method described above.

First, the existing interrupt pointer is fetched. As indicated by the comment, the 6811's TOC4 timer is used to implement the System Interrupt. The vector is poked into the JMP instruction that will conclude the interrupt code itself. Next, the 6811 interrupt pointer is replaced with a pointer to the new code. These two steps complete the initialization sequence.

The actual interrupt code is quite short. It toggles bit 3 of the 6811's PORTA register. The PORTA register controls the eight pins of Port A that connect to external hardware; bit 3 is connected to the piezo beeper.

The interrupt code exits with a jump instruction. The argument for this jump is poked in by the initialization program.

The method allows any number of programs located in separate files to attach themselves to the System Interrupt. Because these files can be loaded from the C environment, this system affords maximal flexibility to the user, with small overhead in terms of code efficiency.

The Binary Object File

The source file for a binary program must be named with the `.asm` suffix. Once the `.asm` file is created, a special version of the 6811 assembler program is used to construct the binary object code. This program creates a file containing the assembled machine code plus label definitions of entry points and C variables.

```
S1168020053900 37FD802239FC802239CC0045FD8022393C
S9030000FC
S116872B05390037FD872D39FC872D39CC0045FD872D39F4
S9030000FC
6811 assembler version 2.1  10-Aug-91
  please send bugs to Randy Sargent
(rsargent@athena.mit.edu)
  original program by Motorola.
subroutine_double 872b *0007
subroutine_get_foo 8733 *0021
subroutine_initialize_module 8737 *0026
subroutine_set_foo 872f *0016
```

```
variable_foo 872d *0012 0017 0022 0028
```

The program `as11_ic` is used to assemble the source code and create a binary object file. It is given the filename of the source file as an argument. The resulting object file is automatically given the suffix `.icb` (for IC Binary). The binary object file that is created from the `testicb.asm` example file is shown above. Currently, `as11_ic` runs only under UNIX; if you need to create `.icb` files from another platform, you can use our ICB Assembler server on the World Wide Web (http://www.newtonlabs.com/ic/icb.html) <http://www.newtonlabs.com/ic/icb.html>.

Loading an icb File

Once the `.icb` file is created, it can be loaded into IC just like any other C file. If there are C functions that are to be used in conjunction with the binary programs, it is customary to put them into a file with the same name as the `.icb` file, and then use a `.lis` file to loads the two files together.

Passing Array Pointers to a Binary Program

A pointer to an array is a 16-bit integer address. To coerce an array pointer to an integer, use the following form:

```
array_ptr= (int) array;
```

where `array_ptr` is an integer and `array` is an array.

When compiling code that performs this type of pointer conversion, IC must be used in a special mode. Normally, IC does not allow certain types of pointer manipulation. To compile this type of code, use the following invokation:

```
ic -wizard
```

Arrays are internally represented with a two-byte length value followed by the array contents.

IC File Formats and Management

This section explains how IC deals with multiple source files.

C Programs

All files containing C code must be named with the "`.c`" suffix.

Loading functions from more than one C file can be done by issuing commands at the IC prompt to load each of the files. For example, to load the C files named `foo.c` and `bar.c`:

```
C> load foo.c
C> load bar.c
```

Alternatively, the files could be loaded with a single command:

```
C> load foo.c bar.c
```

List Files

ภาคผนวก ค. โปรแกรมควบคุมหุ่นยนต์

```
/* Line follow */
float  motorR, Int_motorR = 100.0;
float  motorL, Int_motorL = 100.0;
int    motor1    = 1;       /* Left */
int    motor2    = 0;       /* Right */

int    FORWARD   = 1;
int    LEFT_TURN  = 2;
int    RIGHT_TURN = 3;
int    BACKWARD  = 4;
int    STOP      = 5;
int    RIGHT_ARC  = 6;

int    cruise_command   = FORWARD;
int    line_command     = STOP;
int    line_find_command = STOP;
int    ir_command       = STOP;

int    cruise_active    = 1;
int    line_active      = 0;
int    line_find_active  = 0;
int    white_find_active = 0;
int    ir_active        = 0;

int    operation;
int    photo_dead_zone =  30;
int    counter;

void move(int operation){
  if (operation == FORWARD){
    motor(motor1,100.0); motor(motor2,100.0);
  }
  else if (operation == LEFT_TURN){
    motor(motor1,(-45.0)); motor(motor2,45.0);
  }
```

```
    if (operation == FORWARD){
        motor(motor1,motorL); motor(motor2,motorR);
    }
    else if (operation == LEFT_TURN){
        motor(motor1,(0.0)); motor(motor2,45.0);
    }
    else if (operation == RIGHT_TURN){
        motor(motor1,45.0); motor(motor2,(0.0));
    }
    else if (operation == BACKWARD){
        motor(motor1,-(motorL));  motor(motor2,-(motorR));
        sleep(3.0);
    }
    else if (operation == STOP){
        motor(motor1,0.0); motor(motor2,0.0);
        sleep(5.0);
    }
}

void motor_control(){
  while(1){
    if  (sound_active)
        move(sound_command);
    else if (ir_active)
        move(ir_command);
    else if (bumper_active)
        move(bumper_command);
    else if (photo_active)
        move(photo_command);
    else if (linear_active)
        move(linear_command);
    defer();
  }
}
```

```
void ir()
{     int ir_val = 0,kl=0;
      while(1){
        if( (analog(4)) > 50 )
           ir_val++;
        else
           ir_val = 0;

        if( ir_val > 5 ){
          ir_active = 1;
          ir_command = RIGHT_TURN;
          while((sound_active == 0) && (kl<40) )
           {
            move(RIGHT_TURN);
            kl++;
           }
          kl = 0;
         }
        else
          ir_active = 0;
        defer();
     }
}

void linear(){
     while(1){
        linear_command = FORWARD;
        linear_active  = 1;
        defer();

     }
}

void  mic()
```

```
{     int  sound_val;
      while(1){
          sound_val = analog(2);
          if((sound_val > 133) || (sound_val < 120)){
             sound_active = 1;
             sound_command = STOP;
             }
          else
             sound_active = 0;
          defer();
      }
}


void photo(){
   int lpc, rpc, delta;      /* Left and Right Photocell */
   while(1){
      /* Read photocell, add cal constant */
      lpc   = analog(6)+photo_cal;
      rpc   = analog(5);
      delta = rpc - lpc;
      if (abs(delta) > photo_dead_zone) {
         if (delta > 0)
            photo_command = LEFT_TURN;
         else
            photo_command = RIGHT_TURN;
         photo_active = 1;
      } else
         photo_active = 0;
         defer();
   }
}

void bump()
{   int   bump_val=0;
```

```
    while(1){
     bump_val = analog(3);
       if( (analog(3) > 94)&&(analog(3) < 98)) {
          bumper_active=1;
          bumper_command = FORWARD_SPEED;
          }
     else  if(( analog(3) > 126 )&& (analog(3)< 130)) {
          bumper_active=1;
          bumper_command = LEFT_TURN;
          }
     else  if( (analog(3)> 62) && (analog(3) < 68)) {
          bumper_active=1;
          bumper_command = RIGHT_TURN;
          }
     else
          bumper_active=0;
     defer();
     }
}


int abs(int arg){
   if (arg < 0)
      return (- arg);
   else
      return (arg);
}

void main(){
      start_process(mic());
      start_process(ir());
      start_process(bump());
      start_process(linear());
      start_process(photo());
      start_process(motor_control());
```

```
}
```

ภาคผนวก ง. รายละเอียดของอุปกรณ์

DALLAS SEMICONDUCTOR

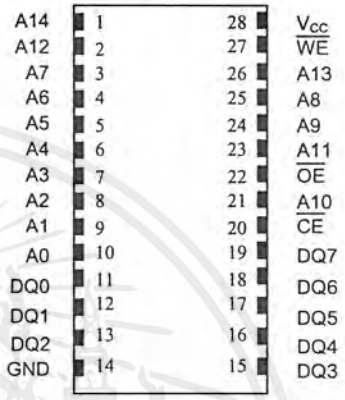
DS1230Y/AB
256k Nonvolatile SRAM

www.dalsemi.com

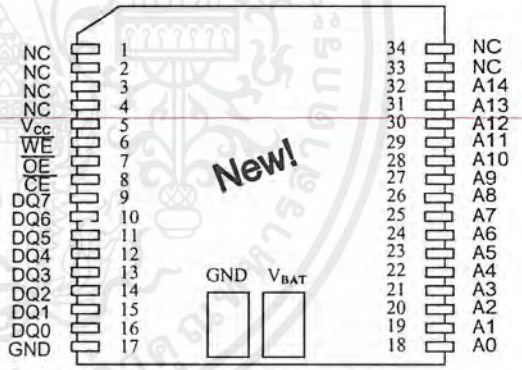
FEATURES

- 10 years minimum data retention in the absence of external power
- Data is automatically protected during power loss
- Replaces 32k x 8 volatile static RAM, EEPROM or Flash memory
- Unlimited write cycles
- Low-power CMOS
- Read and write access times as fast as 70 ns
- Lithium energy source is electrically disconnected to retain freshness until power is applied for the first time
- Full ±10% V_{CC} operating range (DS1230Y)
- Optional ±5% V_{CC} operating range (DS1230AB)
- Optional industrial temperature range of -40°C to +85°C, designated IND
- JEDEC standard 28-pin DIP package
- New PowerCap Module (PCM) package
  - Directly surface-mountable module
  - Replaceable snap-on PowerCap provides lithium backup battery
  - Standardized pinout for all nonvolatile SRAM products
  - Detachment feature on PowerCap allows easy removal using a regular screwdriver

PIN ASSIGNMENT



28-Pin ENCAPSULATED PACKAGE
740-mil EXTENDED



34-Pin POWERCAP MODULE (PCM)
(USES DS9034PC POWERCAP)

PIN DESCRIPTION

A0 - A14	- Address Inputs
DQ0 - DQ7	- Data In/Data Out
$\overline{CE}$	- Chip Enable
$\overline{WE}$	- Write Enable
$\overline{OE}$	- Output Enable
V_{CC}	- Power (+5V)
GND	- Ground
NC	- No Connect

DESCRIPTION

The DS1230 256k Nonvolatile SRAMs are 262,144-bit, fully static, nonvolatile SRAMs organized as 32,768 words by 8 bits. Each NV SRAM has a self-contained lithium energy source and control circuitry which constantly monitors V_{CC} for an out-of-tolerance condition. When such a condition occurs, the lithium energy source is automatically switched on and write protection is unconditionally enabled to prevent data corruption. DIP-package DS1230 devices can be used in place of existing 32k x 8 static RAMs directly conforming to the popular bytewide 28-pin DIP standard. The DIP devices also match the pinout of 28256 EEPROMs, allowing direct substitution while enhancing performance. DS1230 devices in the Low Profile Module package are specifically designed for surface-mount applications. There is no limit on the number of write cycles that can be executed and no additional support circuitry is required for microprocessor interfacing.

READ MODE

The DS1230 devices execute a read cycle whenever $\overline{WE}$ (Write Enable) is inactive (high) and $\overline{CE}$ (Chip Enable) and $\overline{OE}$ (Output Enable) are active (low). The unique address specified by the 15 address inputs (A_0 - A_{14}) defines which of the 32,768 bytes of data is to be accessed. Valid data will be available to the eight data output drivers within t_{ACC} (Access Time) after the last address input signal is stable, providing that $\overline{CE}$ and $\overline{OE}$ (Output Enable) access times are also satisfied. If $\overline{OE}$ and $\overline{CE}$ access times are not satisfied, then data access must be measured from the later-occurring signal ($\overline{CE}$ or $\overline{OE}$) and the limiting parameter is either t_{CO} for $\overline{CE}$ or t_{OE} for $\overline{OE}$ rather than address access.

WRITE MODE

The DS1230 devices execute a write cycle whenever the $\overline{WE}$ and $\overline{CE}$ signals are active (low) after address inputs are stable. The later-occurring falling edge of $\overline{CE}$ or $\overline{WE}$ will determine the start of the write cycle. The write cycle is terminated by the earlier rising edge of $\overline{CE}$ or $\overline{WE}$. All address inputs must be kept valid throughout the write cycle. $\overline{WE}$ must return to the high state for a minimum recovery time (t_{WR}) before another cycle can be initiated. The $\overline{OE}$ control signal should be kept inactive (high) during write cycles to avoid bus contention. However, if the output drivers are enabled ($\overline{CE}$ and $\overline{OE}$ active) then $\overline{WE}$ will disable the outputs in t_{ODW} from its falling edge.

DATA RETENTION MODE

The DS1230AB provides full functional capability for V_{CC} greater than 4.75 volts and write protects by 4.5 volts. The DS1230Y provides full functional capability for V_{CC} greater than 4.5 volts and write protects by 4.25 volts. Data is maintained in the absence of V_{CC} without any additional support circuitry. The nonvolatile static RAMs constantly monitor V_{CC}. Should the supply voltage decay, the NV SRAMs automatically write protect themselves, all inputs become "don't care," and all outputs become high-impedance. As V_{CC} falls below approximately 3.0 volts, a power switching circuit connects the lithium energy source to RAM to retain data. During power-up, when V_{CC} rises above approximately 3.0 volts the power switching circuit connects external V_{CC} to RAM and disconnects the lithium energy source. Normal RAM operation can resume after V_{CC} exceeds 4.75 volts for the DS1230AB and 4.5 volts for the DS1230Y.

FRESHNESS SEAL

Each DS1230 device is shipped from Dallas Semiconductor with its lithium energy source disconnected, guaranteeing full energy capacity. When V_{CC} is first applied at a level greater than 4.25 volts, the lithium energy source is enabled for battery back-up operation.

PACKAGES

The DS1230 devices are available in two packages: 28-pin DIP and 34-pin PowerCap Module (PCM). The 28-pin DIP integrates a lithium battery, an SRAM memory and a nonvolatile control function into a single package with a JEDEC-standard, 600-mil DIP pinout. The 34-pin PowerCap Module integrates SRAM memory and nonvolatile control along with contacts for connection to the lithium battery in the DS9034PC PowerCap. The PowerCap Module package design allows a DS1230 PCM device to be surface mounted without subjecting its lithium backup battery to destructive high-temperature reflow soldering. After a DS1230 PCM is reflow soldered, a DS9034PC PowerCap is snapped on top of the PCM to form a complete Nonvolatile SRAM module. The DS9034PC is keyed to prevent improper attachment. DS1230 PowerCap Modules and DS9034PC PowerCaps are ordered separately and shipped in separate containers. See the DS9034PC data sheet for further information.

ABSOLUTE MAXIMUM RATINGS*

Voltage on Any Pin Relative to Ground	-0.3V to +7.0V
Operating Temperature	0°C to 70°C, -40°C to +85°C for IND parts
Storage Temperature	-40°C to +70°C, -40°C to +85°C for IND parts
Soldering Temperature	260°C for 10 seconds

* This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operation sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods of time may affect reliability.

RECOMMENDED DC OPERATING CONDITIONS

(t_A: See Note 10)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
DS1230AB Power Supply Voltage	V_{CC}	4.75	5.0	5.25	V	
DS1230Y Power Supply Voltage	V_{CC}	4.5	5.0	5.5	V	
Logic 1	V_{IH}	2.2		V_{CC}	V	
Logic 0	V_{IL}	0.0		0.8	V	

DC ELECTRICAL CHARACTERISTICS

(V_{CC}=5V ± 5% for DS1230AB)
(t_A: See Note 10) (V_{CC}=5V ± 10% for DS1230Y)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
Input Leakage Current	I_{IL}	-1.0		+1.0	µA	
I/O Leakage Current $\overline{CE} \geq V_{IH} \leq V_{CC}$	I_{IO}	-1.0		+1.0	µA	
Output Current @ 2.4V	I_{OH}	-1.0			mA	
Output Current @ 0.4V	I_{OL}	2.0			mA	
Standby Current $\overline{CE}$ =2.2V	I_{CCS1}		5.0	10.0	mA	
Standby Current $\overline{CE}$ =V_{CC}-0.5V	I_{CCS2}		3.0	5.0	mA	
Operating Current	I_{CCO1}			85	mA	
Write Protection Voltage (DS1230AB)	V_{TP}	4.50	4.62	4.75	V	
Write Protection Voltage (DS1230Y)	V_{TP}	4.25	4.37	4.5	V	

CAPACITANCE

(t_A=25°C)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
Input Capacitance	C_{IN}		5	10	pF	
Input/Output Capacitance	$C_{I/O}$		5	10	pF	

AC ELECTRICAL CHARACTERISTICS

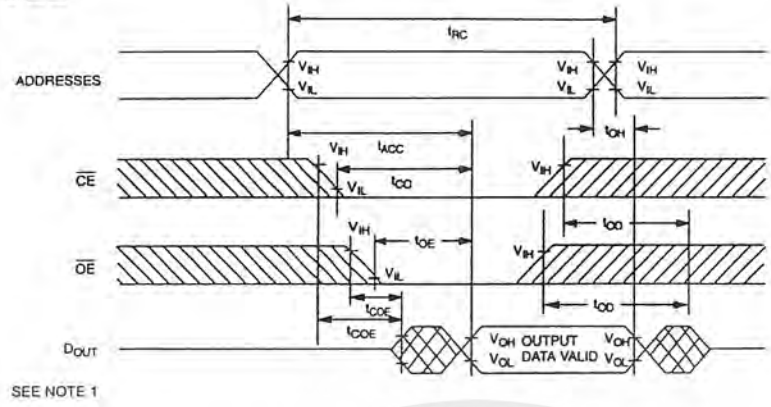
(V_{CC}=5V ± 5% for DS1230AB)
(t_A: See Note 10) (V_{CC}=5V ± 10% for DS1230Y)

PARAMETER	SYMBOL	DS1230AB-70 DS1230Y-70		DS1230AB-85 DS1230Y-85		DS1230AB-100 DS1230Y-100		UNITS	NOTES
		MIN	MAX	MIN	MAX	MIN	MAX		
Read Cycle Time	t_{RC}	70		85		100		ns	
Access Time	t_{ACC}		70		85		100	ns	
$\overline{OE}$ to Output Valid	t_{OE}		35		45		50	ns	
$\overline{CE}$ to Output Valid	t_{CO}		70		85		100	ns	
$\overline{OE}$ or $\overline{CE}$ to Output Active	t_{COE}	5		5		5		ns	5
Output High Z from Deselection	t_{OD}		25		30		35	ns	5
Output Hold from Address Change	t_{OH}	5		5		5		ns	
Write Cycle Time	t_{WC}	70		85		100		ns	
Write Pulse Width	t_{WP}	55		65		75		ns	3
Address Setup Time	t_{AW}	0		0		0		ns	
Write Recovery Time	t_{WR1} t_{WR2}	5 15		5 15		5 15		ns	12 13
Output High Z from $\overline{WE}$	t_{ODW}		25		30		35	ns	5
Output Active from $\overline{WE}$	t_{OEW}	5		5		5		ns	5
Data Setup Time	t_{DS}	30		35		40		ns	4
Data Hold Time	t_{DH1} t_{DH2}	0 10		0 10		0 10		ns	12 13

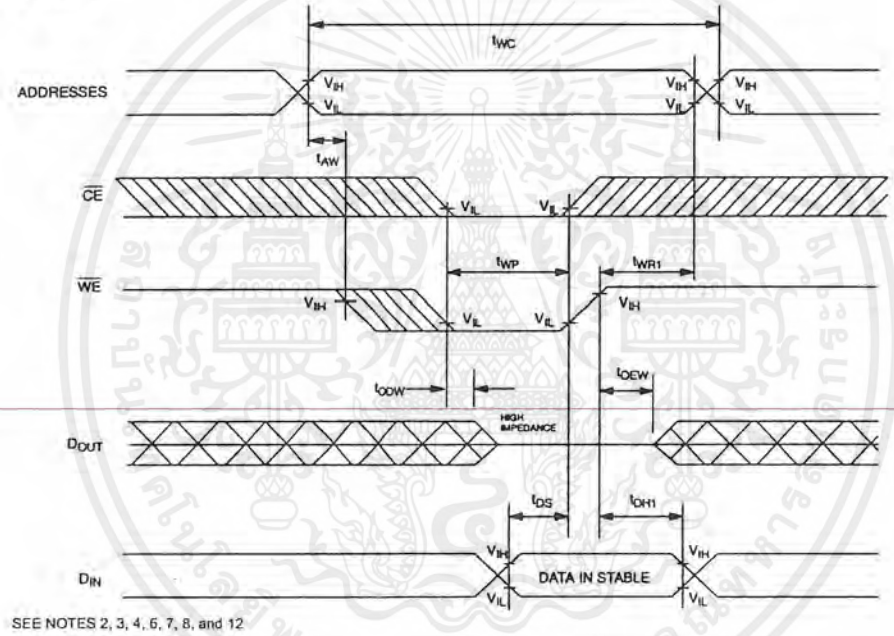
AC ELECTRICAL CHARACTERISTICS (cont'd)

PARAMETER	SYMBOL	DS1230AB-120 DS1230Y-120		DS1230AB-150 DS1230Y-150		DS1230AB-200 DS1230Y-200		UNITS	NOTES
		MIN	MAX	MIN	MAX	MIN	MAX		
Read Cycle Time	t_{RC}	120		150		200		ns	
Access Time	t_{ACC}		120		150		200	ns	
$\overline{OE}$ to Output Valid	t_{OE}		60		70		100	ns	
$\overline{CE}$ to Output Valid	t_{CO}		120		150		200	ns	
$\overline{OE}$ or $\overline{CE}$ to Output Active	t_{COE}	5		5		5		ns	5
Output High Z from Deselection	t_{OD}		35		35		35	ns	5
Output Hold from Address Change	t_{OH}	5		5		5		ns	
Write Cycle Time	t_{WC}	120		150		200		ns	
Write Pulse Width	t_{WP}	90		100		100		ns	3
Address Setup Time	t_{AW}	0		0		0		ns	
Write Recovery Time	t_{WR1} t_{WR2}	5 15		5 15		5 15		ns	12 13
Output High Z from $\overline{WE}$	t_{ODW}		35		35		35	ns	5
Output Active from $\overline{WE}$	t_{OEW}	5		5		5		ns	5
Data Setup Time	t_{DS}	50		60		80		ns	4
Data Hold Time	t_{DH1} t_{DH2}	0 10		0 10		0 10		ns	12 13

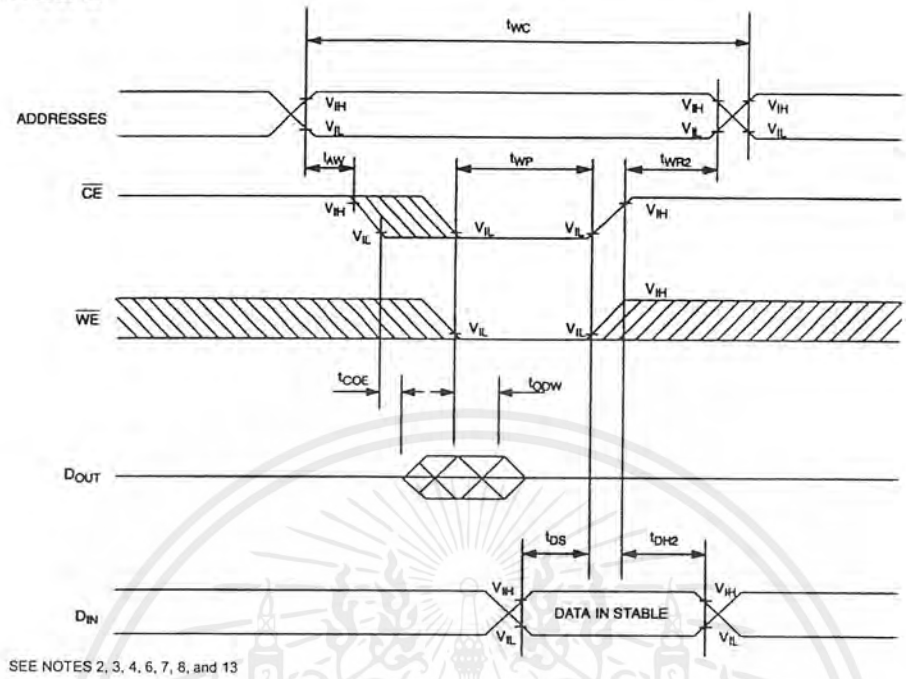
READ CYCLE



WRITE CYCLE 1

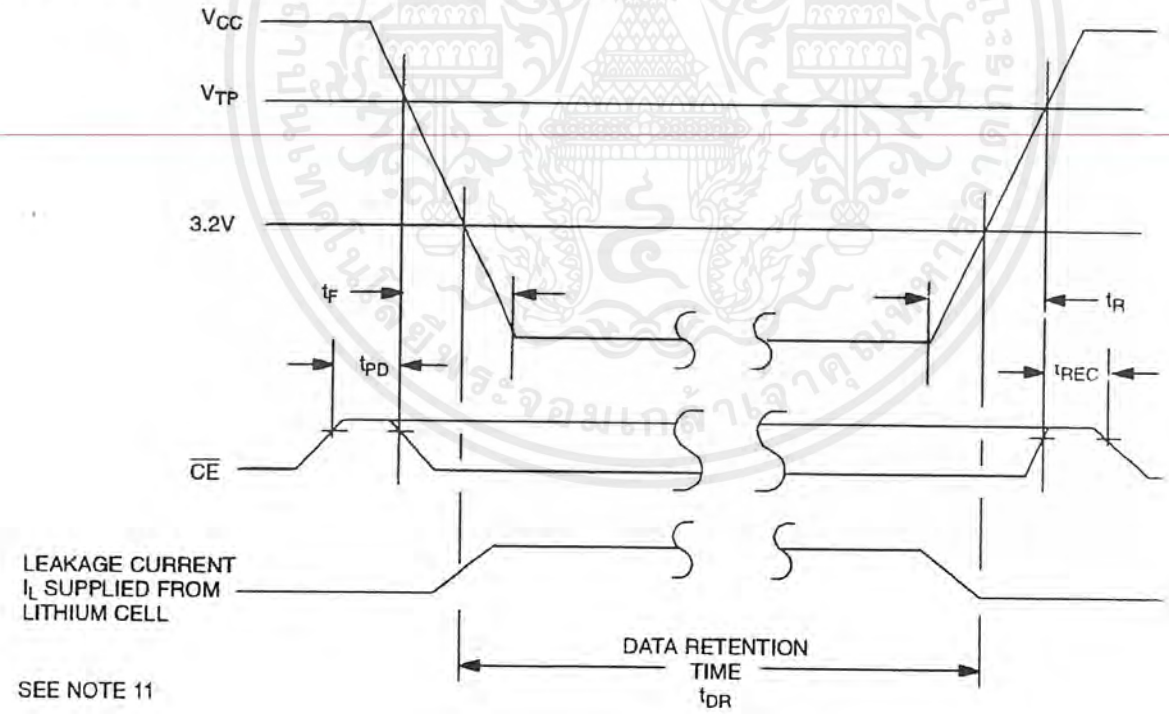


WRITE CYCLE 2



SEE NOTES 2, 3, 4, 6, 7, 8, and 13

POWER-DOWN/POWER-UP CONDITION



SEE NOTE 11

POWER-DOWN/POWER-UP TIMING

(t_A: See Note 10)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
CE, at V_{IH} before Power-Down	t_{PD}	0			µs	11
V_{CC} slew from V_{TP} to 0V ($\overline{CE}$ at V_{IH})	t_F	300			µs	
V_{CC} slew from 0V to V_{TP} ($\overline{CE}$ at V_{IH})	t_R	300			µs	
$\overline{CE}$ at V_{IH} after Power-Up	t_{REC}	2		125	ms	

(t_A=25°C)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
Expected Data Retention Time	t_{DR}	10			years	9

WARNING:

Under no circumstance are negative undershoots, of any amplitude, allowed when device is in battery backup mode.

NOTES:

1. $\overline{WE}$ is high for a Read Cycle.
2. $\overline{OE}$ = V_{IH} or V_{IL}. If $\overline{OE}$ = V_{IH} during write cycle, the output buffers remain in a high-impedance state.
3. t_{WP} is specified as the logical AND of $\overline{CE}$ and $\overline{WE}$. t_{WP} is measured from the latter of $\overline{CE}$ or $\overline{WE}$ going low to the earlier of $\overline{CE}$ or $\overline{WE}$ going high.
4. t_{DH}, t_{DS} are measured from the earlier of $\overline{CE}$ or $\overline{WE}$ going high.
5. These parameters are sampled with a 5 pF load and are not 100% tested.
6. If the $\overline{CE}$ low transition occurs simultaneously with or latter than the $\overline{WE}$ low transition, the output buffers remain in a high-impedance state during this period.
7. If the $\overline{CE}$ high transition occurs prior to or simultaneously with the $\overline{WE}$ high transition, the output buffers remain in high-impedance state during this period.
8. If $\overline{WE}$ is low or the $\overline{WE}$ low transition occurs prior to or simultaneously with the $\overline{CE}$ low transition, the output buffers remain in a high-impedance state during this period.
9. Each DS1230Y has a built-in switch that disconnects the lithium source until V_{CC} is first applied by the user. The expected t_{DR} is defined as accumulative time in the absence of V_{CC} starting from the time power is first applied by the user.
10. All AC and DC electrical characteristics are valid over the full operating temperature range. For commercial products, this range is 0°C to 70°C. For industrial products (IND), this range is -40°C to +85°C.
11. In a power-down condition the voltage on any pin may not exceed the voltage on V_{CC}.
12. t_{WR1} and t_{DH1} are measured from $\overline{WE}$ going high.
13. t_{WR2} and t_{DH2} are measured from $\overline{CE}$ going high.
14. DS1230 DIP modules are recognized by Underwriters Laboratory (U.L.®) under file E99151. DS1230 PowerCap modules are pending U.L. review. Contact the factory for status.

DC TEST CONDITIONS

Outputs Open
Cycle = 200 ns for operating current
All voltages are referenced to ground

AC TEST CONDITIONS

Output Load: 100 pF + 1TTL Gate
Input Pulse Levels: 0 - 3.0V
Timing Measurement Reference Levels
Input: 1.5V
Output: 1.5V
Input pulse Rise and Fall Times: 5 ns

ORDERING INFORMATION

DS1230 TTP - SSS - III

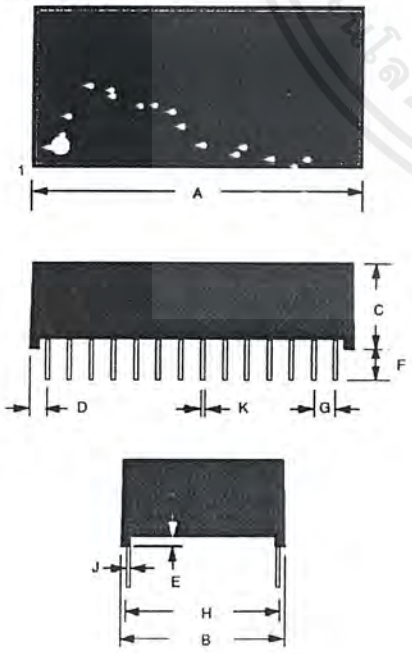
Operating Temperature Range
blank: 0° to 70°
IND: -40° to +85°C

Access Speed
70: 70 ns
85: 85 ns
100: 100 ns
120: 120 ns
150: 150 ns
200: 200 ns

Package Type
blank: 28-pin 600-mil DIP
P: 34-pin PowerCap Module

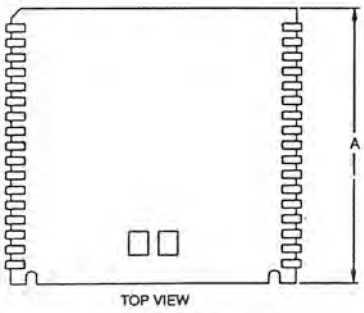
V_{CC} Tolerance
AB: ±5%
Y: ±10%

DS1230Y/AB NONVOLATILE SRAM, 28-PIN 740-MIL EXTENDED DIP MODULE

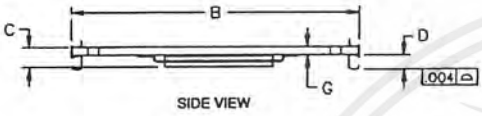


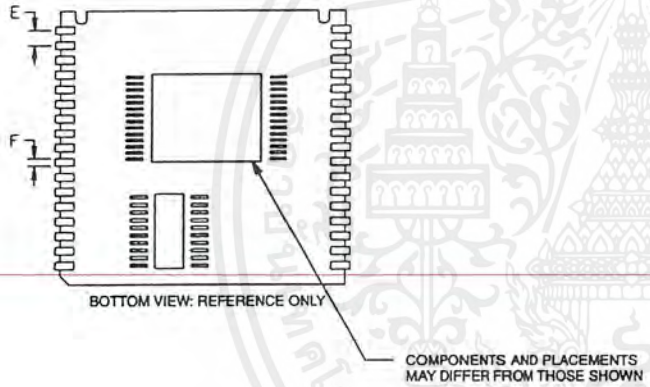
PKG	28-PIN	
DIM	MIN	MAX
A IN.	1.480	1.500
MM	37.60	38.10
B IN.	0.720	0.740
MM	18.29	18.80
C IN.	0.355	0.375
MM	9.02	9.52
D IN.	0.080	0.110
MM	2.03	2.79
E IN.	0.015	0.025
MM	0.38	0.63
F IN.	0.120	0.160
MM	3.05	4.06
G IN.	0.090	0.110
MM	2.29	2.79
H IN.	0.590	0.630
MM	14.99	16.00
J IN.	0.008	0.012
MM	0.20	0.30
K IN.	0.015	0.021
MM	0.38	0.53

DS1230Y/AB NONVOLATILE SRAM, 34-PIN POWERCAP MODULE

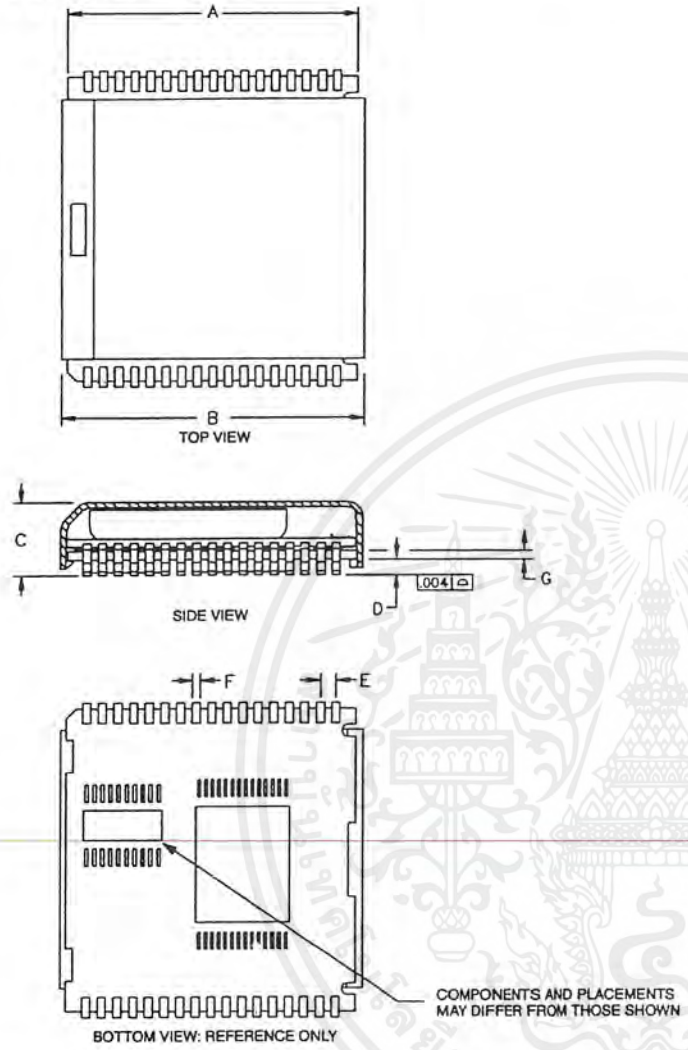


PKG DIM	INCHES		
	MIN	NOM	MAX
A	0.920	0.925	0.930
B	0.980	0.985	0.990
C	-	-	0.080
D	0.052	0.055	0.058
E	0.048	0.050	0.052
F	0.015	0.020	0.025
G	0.020	0.025	0.030





DS1230Y/AB NONVOLATILE SRAM, 34-PIN POWERCAP MODULE WITH POWERCAP



PKG DIM	INCHES		
	MIN	NOM	MAX
A	0.920	0.925	0.930
B	0.955	0.960	0.965
C	0.240	0.245	0.250
D	0.052	0.055	0.058
E	0.048	0.050	0.052
F	0.015	0.020	0.025
G	0.020	0.025	0.030

ASSEMBLY AND USE

Reflow soldering

Dallas Semiconductor recommends that PowerCap Module bases experience one pass through solder reflow oriented label-side up (live-bug).

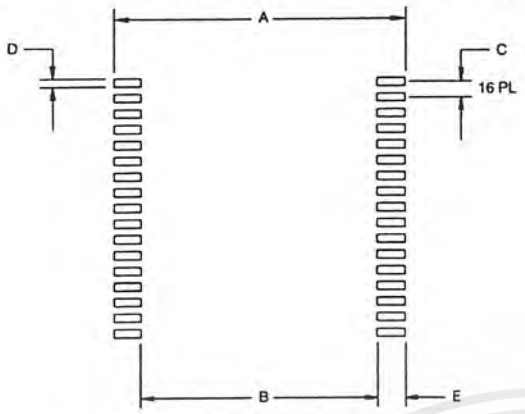
Hand soldering and touch-up

Do not touch soldering iron to leads for more than 3 seconds. To solder, apply flux to the pad, heat the lead frame pad and apply solder. To remove part, apply flux, heat pad until solder reflows, and use a solder wick.

LPM replacement in a socket

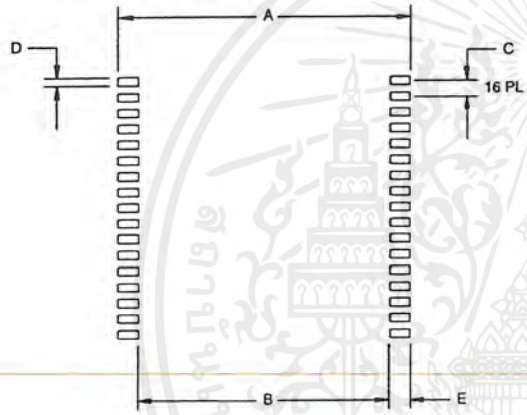
To replace a Low Profile Module in a 68-pin PLCC socket, attach a DS9034PC PowerCap to a module base then insert the complete module into the socket one row of leads at a time, pushing only on the corners of the cap. Never apply force to the center of the device. To remove from a socket, use a PLCC extraction tool and ensure that it does not hit or damage any of the module IC components. Do not use any other tool for extraction.

RECOMMENDED POWERCAP MODULE LAND PATTERN



PKG DIM	INCHES		
	MIN	NOM	MAX
A	-	1.050	-
B	-	0.826	-
C	-	0.050	-
D	-	0.030	-
E	-	0.112	-

RECOMMENDED POWERCAP MODULE SOLDER STENCIL



PKG DIM	INCHES		
	MIN	NOM	MAX
A	-	1.050	-
B	-	0.890	-
C	-	0.050	-
D	-	0.030	-
E	-	0.080	-

DALLAS SEMICONDUCTOR

DS275
Line-Powered RS-232 Transceiver Chip

www.dalsemi.com

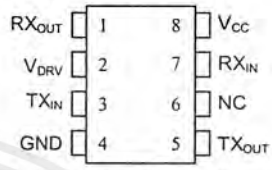
FEATURES

- Low-power serial transmitter/receiver for battery-backed systems
- Transmitter steals power from receive signal line to save power
- Ultra-low static current, even when connected to RS-232-E port
- Variable transmitter level from +5 to +12 volts
- Compatible with RS-232-E signals
- Available in 8-pin, 150 mil wide SOIC package (DS275S)
- Low-power CMOS

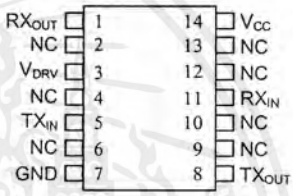
ORDERING INFORMATION

DS275	8-pin DIP
DS275S	8-pin SOIC
DS275E	14-pin TSSOP

PIN ASSIGNMENT



DS275 8-Pin DIP (300-mil)
DS275 8-Pin SOIC (150-mil)



DS275E 14-Pin TSSOP

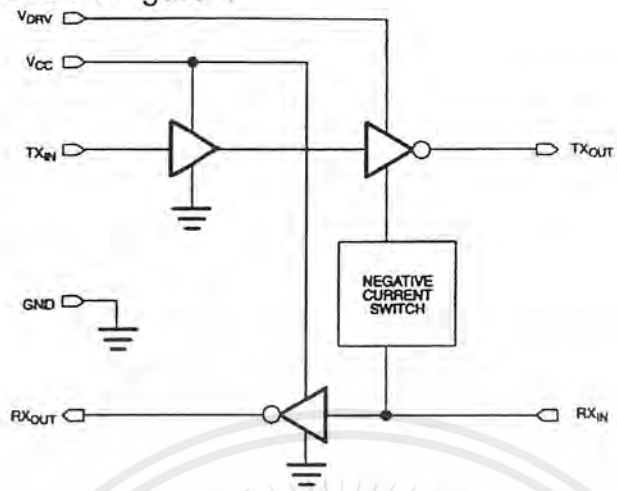
PIN DESCRIPTION

RX_{OUT}	- RS-232 Receiver Output
V_{DRV}	- Transmit driver +V
TX_{IN}	- RS-232 Driver Input
GND	- System Ground (0V)
TX_{OUT}	- RS-232 Driver Output
NC	- No Connection
RX_{IN}	- RS-232 Receive Input
V_{CC}	-System Logic Supply (+5V)

DESCRIPTION

The DS275 Line-Powered RS-232 Transceiver Chip is a CMOS device that provides a low-cost, very low-power interface to RS-232 serial ports. The receiver input translates RS-232 signal levels to common CMOS/TTL levels. The transmitter employs a unique circuit which steals current from the receive RS-232 signal when that signal is in a negative state (marking). Since most serial communication ports remain in a negative state statically, using the receive signal for negative power greatly reduces the DS275's static power consumption. This feature is especially important for battery-powered systems such as laptop computers, remote sensors, and portable medical instruments. During an actual communication session, the DS275's transmitter will use system power (5-12 volts) for positive transitions while still employing the receive signal for negative transitions.

DS275 BLOCK DIAGRAM Figure 1



OPERATION

Designed for the unique requirements of battery-backed systems, the DS275 provides a low-power half-duplex interface to an RS-232 serial port. Typically, a designer must use an RS-232 device which uses system power during both negative and positive transitions of the transmit signal to the RS-232 port. If the connector to the RS-232 port is left connected for an appreciable time after the communication session has ended, power will statically flow into that port, draining the battery capacity. The DS275 eliminates this static current drain by stealing current from the receive line (RXIN) of the RS-232 port when that line is at a negative level (marking). Since most asynchronous communication over an RS-232 connection typically remains in a marking state when data is not being sent, the DS275 will not consume system power in this condition. System power would only be used when positive-going transitions are needed on the transmit RS-232 output (TXOUT) when data is sent. However, since synchronous communication sessions typically exhibit a very low duty-cycle, overall system power consumption remains low.

RECEIVER SECTION

The RXIN pin is the receive input for an RS-232 signal whose levels can range from ±3 to ±15 volts. A negative data signal is called a mark while a positive data signal is called a space. These signals are inverted and then level-shifted to normal +5-volt CMOS/TTL logic levels. The logic output associated with RXIN is RXOUT which swings from +VCC to ground. Therefore, a mark on RXIN produces a logic 1 at RXOUT; a space produces a logic 0.

The input threshold of RXIN is typically around 1.8 volts with 500 millivolts of hysteresis to improve noise rejection. Therefore, an input positive-going signal must exceed 1.8 volts to cause RXOUT to switch states. A negative-going signal must now be lower than 1.3 volts (typically) to cause RXOUT to switch again. An open on RXIN is interpreted as a mark, producing a logic 1 at RXOUT.

TRANSMITTER SECTION

TXIN is the CMOS/TTL-compatible input for digital data from the user system. A logic 1 at TXIN produces a mark (negative data signal) at TXOUT while a logic 0 produces a space (positive data signal). As mentioned earlier, the transmitter section employs a unique driver design that uses the RXIN line for swinging to negative levels. The RXIN line must be in a marking or idle state to take advantage of this design; if RXIN is in a spacing state, TXOUT will only swing to ground. When TXOUT needs to transition to a positive level, it uses the VDRV power pin for this level. VDRV can be a voltage supply between 5 to 12

volts, and in many situations it can be tied directly to the +5 volt V_{CC} supply. *It is important to note that V_{DRV} must be greater than or equal to V_{CC} at all times.*

The voltage range on V_{DRV} permits the use of a 9-volt battery in order to provide a higher voltage level when TX_{OUT} is in a space state. When V_{CC} is shut off to the DS275 and V_{DRV} is still powered (as might happen in a battery-backed condition), only a small leakage current (about 50-100 nA) will be drawn. If TX_{OUT} is loaded during such a condition, V_{DRV} will draw current only if RX_{IN} is not in a negative state. During normal operation (V_{CC}=5 volts), V_{DRV} will draw less than 2 uA when TX_{OUT} is marking. Of course, when TX_{OUT} is spacing, V_{DRV} will draw substantially more current— about 3 mA, depending upon its voltage and the impedance that TX_{OUT} sees.

The TX_{OUT} output is slew rate-limited to less than 30 volts/us in accordance with RS-232 specifications. In the event TX_{OUT} should be inadvertently shorted to ground, internal current-limiting circuitry prevents damage, even if continuously shorted.

RS-232 COMPATIBILITY

The intent of the DS275 is not so much to meet all the requirements of the RS-232 specification as to offer a low-power solution that will work with most RS-232 ports with a connector length of less than 10 feet. As a prime example, the DS275 will not meet the RS-232 requirement that the signal levels be at least ±5 volts minimum when terminated by a 3 kΩ load and V_{DRV} = +5 volts. Typically a voltage of 4 volts will be present at TX_{OUT} when spacing. However, since most RS-232 receivers will correctly interpret any voltage over 2 volts as a space, there will be no problem transmitting data.

APPLICATIONS INFORMATION

The DS275 is designed as a low-cost, RS-232-E interface expressly tailored for the unique requirements of battery-operated handheld products. As shown in the electrical specifications, the DS275 draws exceptionally low operating and static current. During normal operation when data from the handheld system is sent from the TX_{OUT} output, the DS275 only draws significant V_{DRV} current when TX_{OUT} transitions positively (spacing). This current flows primarily into the RS-232 receiver's 3-7 kΩ load at the other end of the attaching cable. When TX_{OUT} is marking (a negative data signal), the V_{DRV} current falls dramatically since the negative voltage is provided by the transmit signal from the other end of the cable. This represents a large reduction in overall operating current, since typical RS-232 interface chips use charge-pump circuits to establish both positive and negative levels at the transmit driver output.

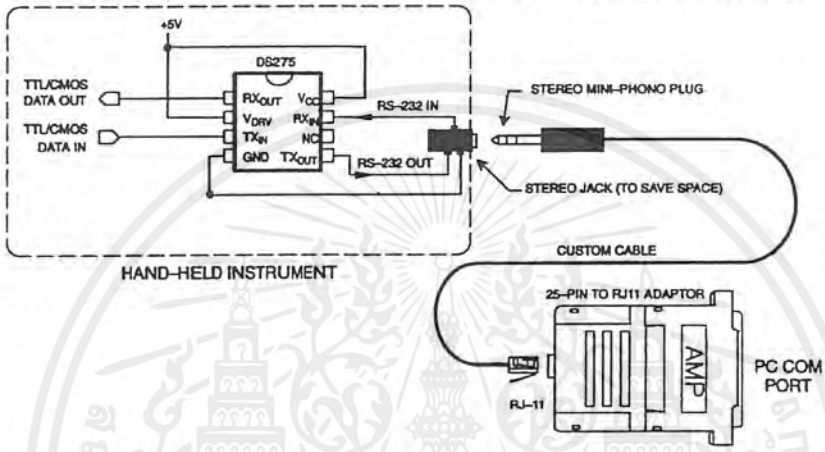
To obtain the lowest power consumption from the DS275, observe the following guidelines. First, to minimize V_{DRV} current when connected to an RS-232 port, always maintain TX_{IN} at a logic 1 when data is not being transmitted (idle state). This will force TX_{OUT} into the marking state, minimizing V_{DRV} current. Second, V_{DRV} current will drop to less than 100 nA when V_{CC} is grounded. Therefore, if V_{DRV} is tied directly to the system battery, the logic +5 volts can be turned off to achieve the lowest possible power state.

FULL-DUPLEX OPERATION

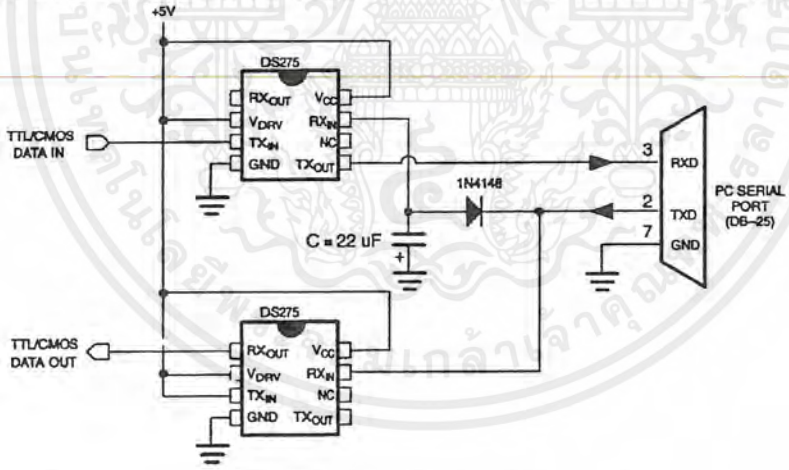
The DS275 is intended primarily for half-duplex operation; that is, RX_{IN} should remain idle in the marking state when transmitting data out TX_{OUT} and visa versa. However, the part can be operated full-duplex with most RS-232-E serial ports since signals swinging between 0 and +5V will usually be correctly interpreted by an RS-232-E receiver device. The 5-volt swing occurs when TX_{OUT} attempts to swing negative while RX_{IN} is at a positive voltage, which turns on an internal weak pulldown to ground for the TX_{OUT} driver's negative reference. So, transmit mark signals at TX_{OUT} may have voltage jumps from some negative value (corresponding to RX_{IN} marking) to approximately ground. One possible

problem that may occur in this case is if the receiver at the other end requires a negative voltage for recognizing a mark. In this situation, the full-duplex circuit shown in Figure 3 can be used as analternative. The 22 µF capacitor forms a negative-charge reservoir; consequently, when the TXD line is spacing (positive), TXOUT still has a negative source available for a time period determined by the capacitor and the load resistance at the other end (3-7 kΩ). This circuit was tested from 150-19,200 bps with error-free operation using a SN75154 Quad Line Receiver as the receiver for the TXOUT signal. Note that the SN75154 can have a marking input threshold below ground; hence there is the need for TXOUT to swing both positive and negative in full-duplex operation with this device.

HANDHELD RS-232-C APPLICATION USING A STEREO MINI-JACK Figure 2



FULL-DUPLEX CIRCUIT USING NEGATIVE-CHARGE STORAGE Figure 3



NOTE:

The capacitor stores negative charge whenever the TXD signal from the PC serial port is in a marking data state (a negative voltage that is typically -10 volts). The top DS275's TXOUT uses this negative charge reservoir when it is in a marking state. The capacitor will discharge to 0 volts when the TXD line is spacing (and TXOUT is still marking) at a time constant determined by its value and the value of the load resistance reflected back to TXOUT. However, when TXD is marking the capacitor will quickly charge back to -10 volts. Note that TXD remains in a marking state when idle, which improves the performance of this circuit.

ABSOLUTE MAXIMUM RATINGS*

VCC	-0.3 to +7.0 volts
VDRV	-0.3 to +13.0 volts

RX_{IN}	±15 volts
TX_{IN}	-0.3 to V_{CC} + 0.3 volts
TX_{OUT}	±15 volts
RX_{OUT}	-0.3 to V_{CC} + 0.3 volts
Storage Temperature	-55°C to +125°C
Operating Temperature	0°C to 70°C

* This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operation sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods of time may affect reliability.

RECOMMENDED DC OPERATING CONDITIONS

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
Logic Supply	V_{CC}	4.5	5.0	5.5	V	1
Transmit Driver Supply	V_{DRV}	4.5	5-12	13.0	V	1
Logic 1 Input	V_{IH}	2.0		V_{CC}+0.3	V	2
Logic 0 Input	V_{IL}	-0.3		+0.8	V	
RS-232 Input Range (RX_{IN})	V_{RS}	-15		+15	V	
Dynamic Supply Current						
$TX_{IN} = V_{CC}$	I_{DRV1}		400	800	µA	
	I_{CC1}		40	100	µA	3
TX_{IN} = GND	I_{DRV1}		3.8	5.0	µA	
	I_{CC1}		40	100	µA	
Static Supply Current						
$TX_{IN} = V_{CC}$	I_{DRV2}		1.5	10.0	µA	
	I_{CC2}		10.0	15.0	µA	4
TX_{IN} = GND	I_{DRV2}		3.8	5.0	mA	
	I_{CC2}		10.0	20.0	µA	
Driver Leakage Current (V_{CC}=0V)	I_{DRV3}		0.05	1.0	µA	5

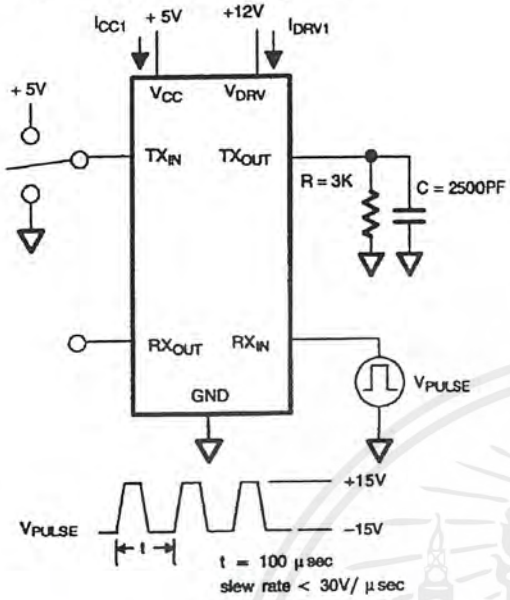
DC ELECTRICAL CHARACTERISTICS (0°C to 70°C; V_{CC} = V_{DRV} = 5V ± 10%)

PARAMETER	SYMBOL	MIN	TYP	MAX	UNITS	NOTES
TX_{OUT} Level High	V_{OTXH}	3.5	4.0	5.0	V	6
TX_{OUT} Level Low	V_{OTXL}	-8.5	-9.0		V	7
TX_{OUT} Short Circuit Current	I_{SC}		+60	+85	mA	
TX_{OUT} Output Slew Rate	t_{SR}			30	V/µs	
Propagation Delay	t_{PD}		5		µs	8
RX_{IN} Input Threshold Low	V_{TL}	0.8	1.2	1.6	V	
RX_{IN} Input Threshold High	V_{TH}	1.6	2.0	2.4	V	
RX_{IN} Threshold Hysteresis	V_{HYS}	0.5	0.8		V	9
RX_{OUT} Output Current @ 2.4V	I_{OH}	-1.0			mA	
RX_{OUT} Output Current @ 0.4V	I_{OL}			3.2	mA	

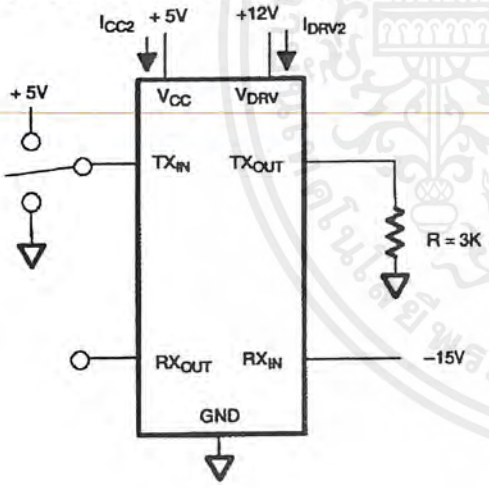
NOTES:

1. V_{DRV} must be greater than or equal to V_{CC}.
2. V_{CC} = V_{DRV} = 5V ± 10%.
3. See test circuit in Figure 4.
4. See test circuit in Figure 5.
5. See test circuit in Figure 6.
6. TX_{IN} = V_{IL} and TX $_{OUT}$ loaded by 3 kΩ to ground.
7. TX_{IN} = V_{IH}, RX_{IN} = -10 volts and TX_{OUT} loaded by 3 kΩ to ground.
8. TX_{IN} to TX_{OUT} - see Figure 7.
9. V_{HYS} = V_{TH} - V_{TL}.

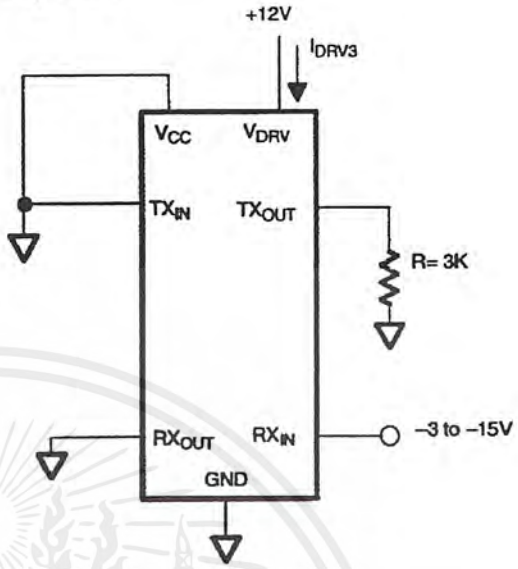
DYNAMIC OPERATING CURRENT TEST CIRCUIT Figure 4



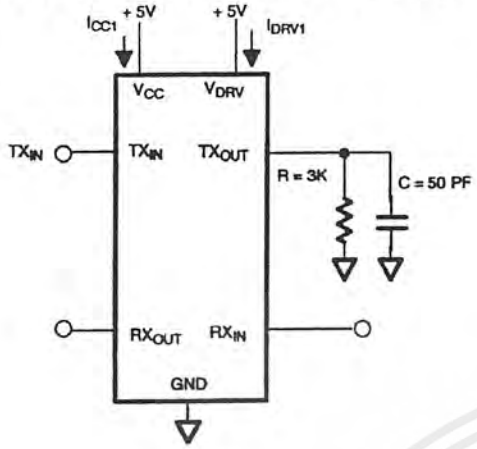
STATIC OPERATING CURRENT TEST CIRCUIT Figure 5

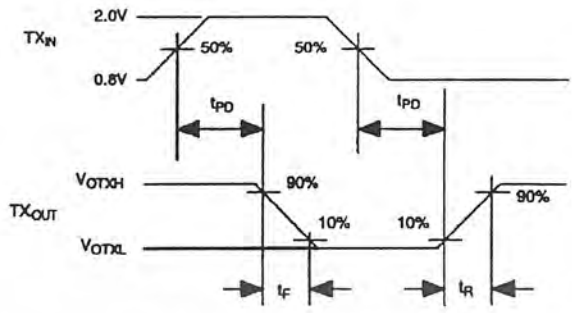


DRIVER LEAKAGE TEST CIRCUIT Figure 6



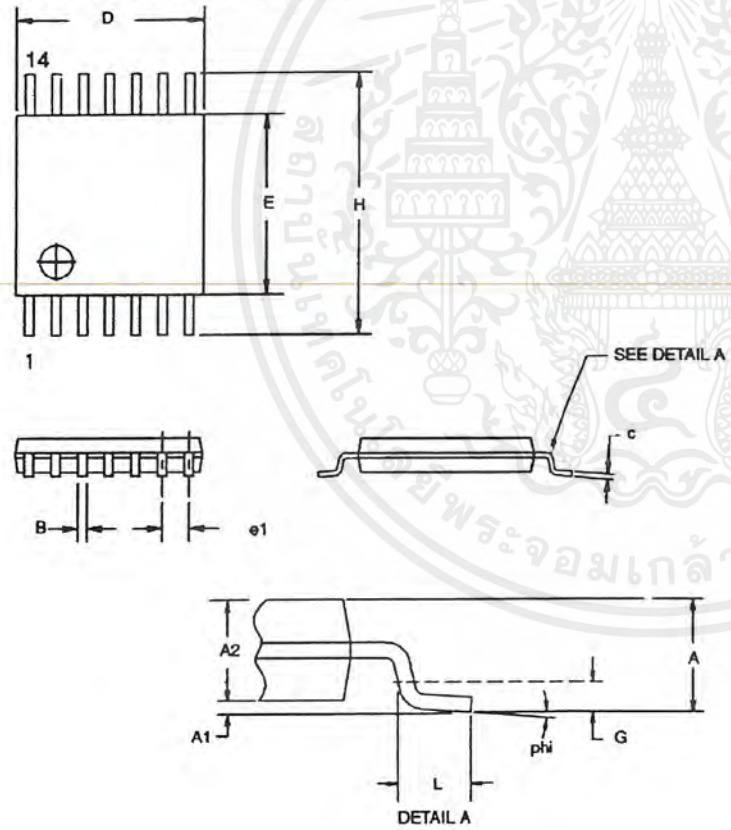
PROPAGATION DELAY TEST CIRCUIT Figure 7





$$t_{SR} = \frac{0.8(V_{OTXH} - V_{OTXL})}{t_F \text{ or } t_R}$$

DS275E 14-PIN TSSOP



DIM	14-PIN	
	MIN	MAX
A MM	-	1.10
A1 MM	0.05	-
A2 MM	0.75	1.05
B MM	0.18	0.30
C MM	0.09	0.18
D MM	4.90	5.10
E MM	4.40 NOM	
e1 MM	0.65 BSC	
G MM	0.25 REF	
H MM	6.25	6.55
L MM	0.50	0.70
phi	0°	8°

INTEGRATED CIRCUITS

UNITRODE

L293
L293D

Push-Pull Four Channel Driver

FEATURES

- Output Current 1A Per Channel (600mA for L293D)
- Peak Output Current 2A Per Channel (1.2A for L293D)
- Inhibit Facility
- High Noise Immunity
- Separate Logic Supply
- Over-Temperature Protection

DESCRIPTION

The L293 and L293D are quad push-pull drivers capable of delivering output currents to 1A or 600mA per channel respectively. Each channel is controlled by a TTL-compatible logic input and each pair of drivers (a full bridge) is equipped with an inhibit input which turns off all four transistors. A separate supply input is provided for the logic so that it may be run off a lower voltage to reduce dissipation.

Additionally the L293D includes the output clamping diodes within the IC for complete interfacing with inductive loads.

Both devices are available in 16-pin Batwing DIP packages. They are also available in Power SOIC and Hermetic DIL packages.

TRUTH TABLE

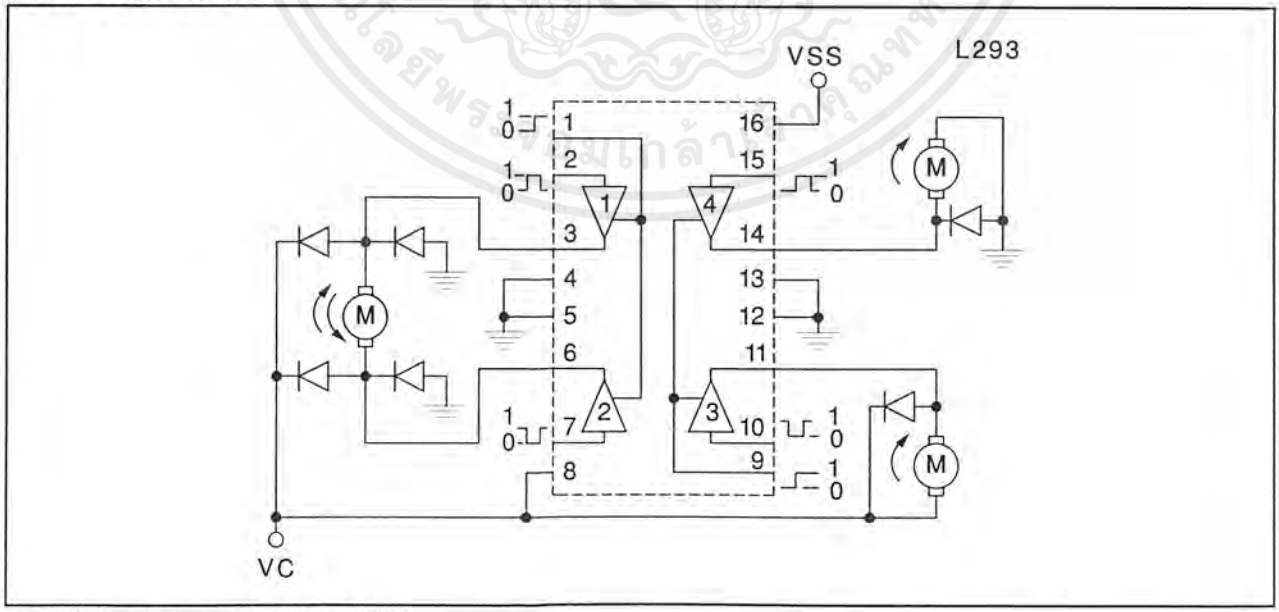
V_I (each channel)	V_{INH}*	V_O
H	H	H
L	H	L
H	L	X**
L	L	X**

**Relative to the considered channel*
***High output impedence*

ABSOLUTE MAXIMUM RATINGS

Collector Supply Voltage, Vc 36V
Logic Supply Voltage, Vss 36V
Input Voltage, V_I 7V
Inhibit Voltage, V_{INH} 7V
Peak Output Current (Non-Repetitive), I_{OUT} (L293) 2A
I_{OUT} (L293D) 1.2A
Total Power Dissipation
at Tground-pins = 80°C, N Batwing pkg, (Note) 5W
Storage and Junction Temperature, Tstg, T_J . . . . . -40 to +150°C
Note:Consult packaging section of Databook for thermal limitations and considerations of packages.

BLOCK DIAGRAM

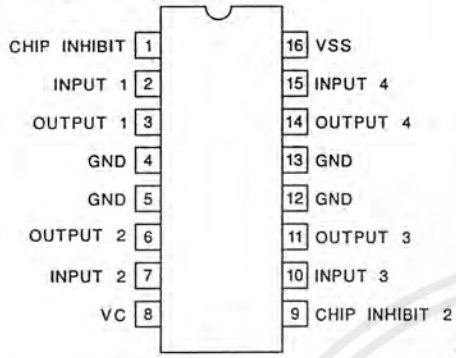


Note: Output diodes are internal in L293D.

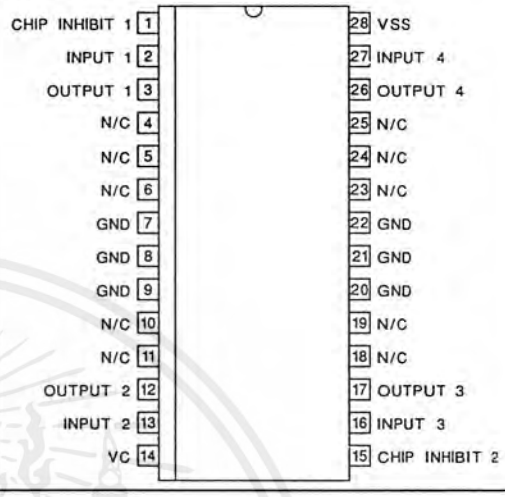
8/94

CONNECTION DIAGRAMS

DIL-16 (TOP VIEW)
N Package, SP Package



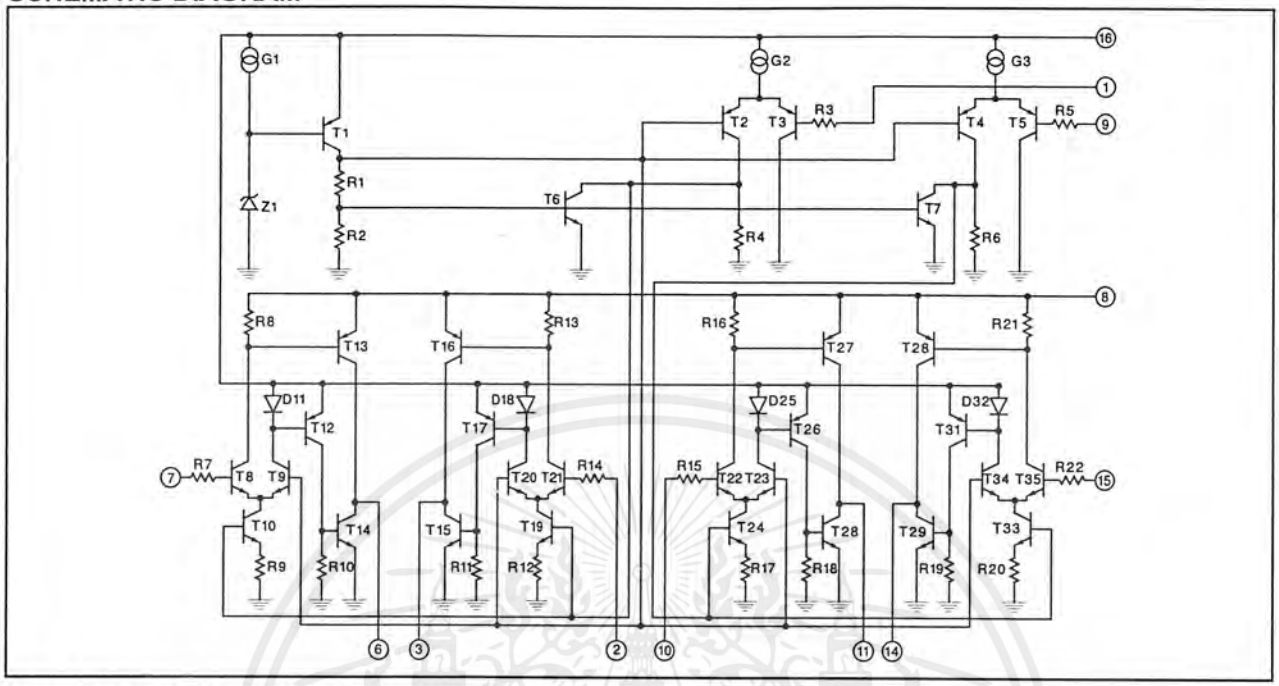
SOIC-28 (TOP VIEW)
DWP Package



ELECTRICAL CHARACTISTICS: (For each channel, V_C = 24V, V_{SS} = 5V, T_{AMB} = 25°C, unless otherwise specified; $T_A = T_J$)

PARAMETER	SYMBOL	TEST CONDITION	MIN.	TYP.	MAX.	UNITS
Collector Supply Voltage	V_C				36	V
Logic Supply Voltage	V_{SS}		4.5		36	V
Collector Supply Current	I_C	V_I = L, I_O = 0, V_{INH} = H		2	6	mA
		V_I = H, I_O = 0, V_{INH} = H		16	24	mA
		V_{INH} = L			4	mA
Total Quiescent Logic Supply Current	I_{SS}	V_I = L, I_O = 0, V_{INH} = H		44	60	mA
		V_I = H, I_O = 0, V_{INH} = H		16	22	mA
		V_{INH} = L		16	24	mA
Input Low Voltage	V_{IL}		-0.3		1.5	V
Input High Voltage	V_{IH}	$V_{SS} \leq 7V$	2.3		V_{SS}	V
		$V_{SS} \geq 7V$	2.3		7	V
Low Voltage Input Current	I_{IL}	V_I = 0V			-10	µA
High Voltage Input Current	I_{IH}	V_I = 4.5V		30	100	µA
Inhibit Low Voltage	$V_{INH, L}$		-0.3		1.5	V
Inhibit High Voltage	$V_{INH, H}$	$V_{SS} \leq 7V$	2.3		V_{SS}	V
		V_{SS} >7V	2.3		7	V
Low Voltage Inhibit Current	$V_{INH, L}$			-30	-100	µA
High Voltage Inhibit Current	$V_{INH, H}$				10	µA
Source Output Saturation Voltage	V_{CEsatH}	I_O = -1A (-0.6A for L293D)		1.4	1.8	V
Sink Output Saturation Voltage	V_{CEsatL}	I_O = 1A (0.6A for L293D)		1.2	1.8	V
Clamp Diode Forward Voltage (L293D only)	V_F	I_F = 0.6A		1.3		V
Rise Time	T_R	0.1 to 0.9 V_O (See Figure 1)		100		ns
Fall Time	T_F	0.9 to 0.1 V_O (See Figure 1)		350		ns
Turn-on Delay	T_{ON}	0. 5 V_I to 0.5 V_O (See Figure 1)		750		ns
Turn-off Delay	T_{OFF}	0. 5 V_I to 0.5 V_O (See Figure 1)		200		ns

SCHEMATIC DIAGRAM



APPLICATION INFORMATION

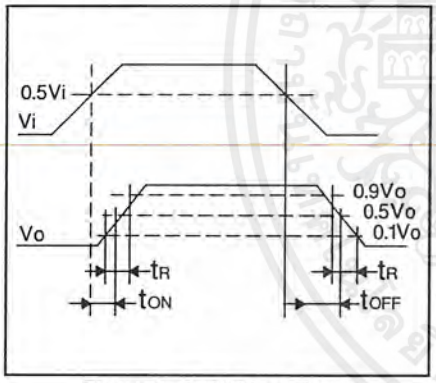


Figure 1: Switching Times

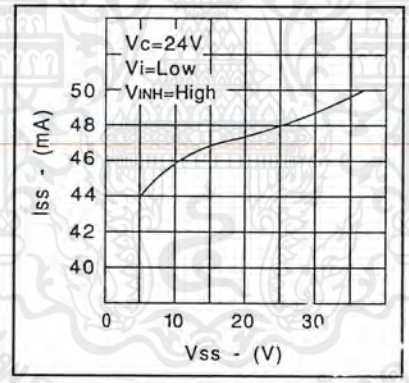


Figure 2: Quiescent Logic Supply Current vs Logic Supply Voltage

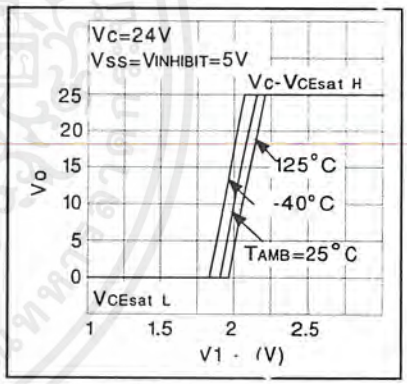


Figure 3: Output Voltage vs Input Voltage

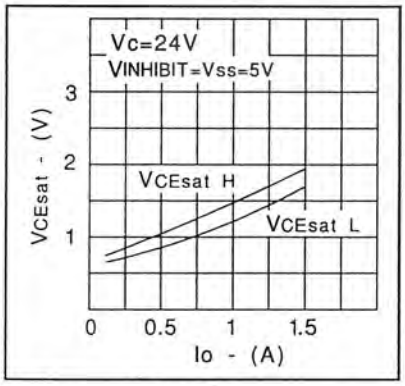


Figure 4: L293 Saturation vs Output Currrent

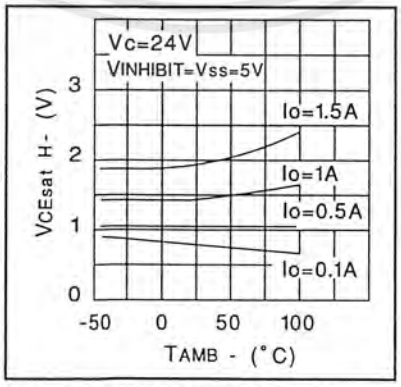


Figure 5: L293 Source Saturation vs Ambient Temperature

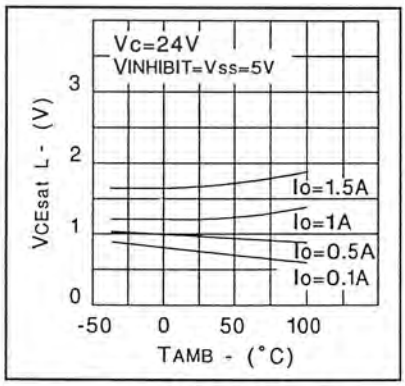


Figure 6: L293 Sink Saturation Voltage vs Ambient Temperature

NOTE: For L293D curves, multiply output current by 0.6.

APPLICATION INFORMATION (Cont.)

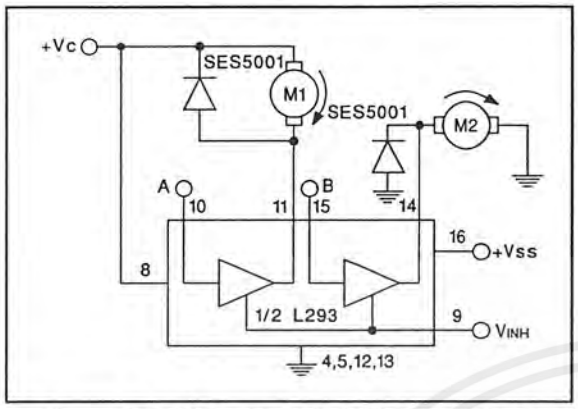


Figure 7: DC Motor Controls (with Connection to Ground and to Supply Voltage)

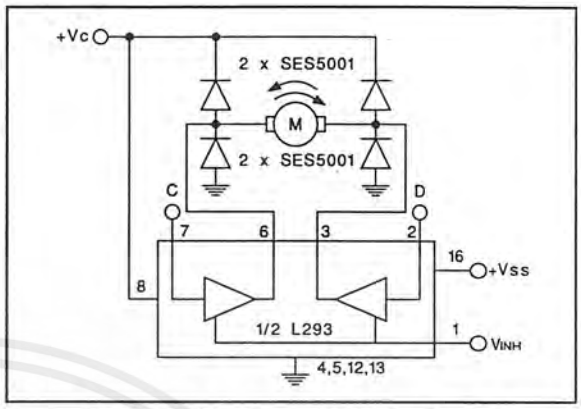


Figure 8: Bidirectional DC Motor Control

VINH	A	M1	B	M2
H	H	Fast Motor Stop	H	Run
H	L	Run	L	Fast Motor Stop
L	X	Free Running Motor Stop	X	Free Running Motor Stop

L = Low H = High X = Don't Care

INPUTS		FUNCTION
VINH = H	C = H; D = L	Turn Right
	C = L; D = H	Turn Left
	C = D	Fast Motor Stop
VINH = L	C = X; D = X	Free Running Motor Stop

L = Low H = High X = Don't Care

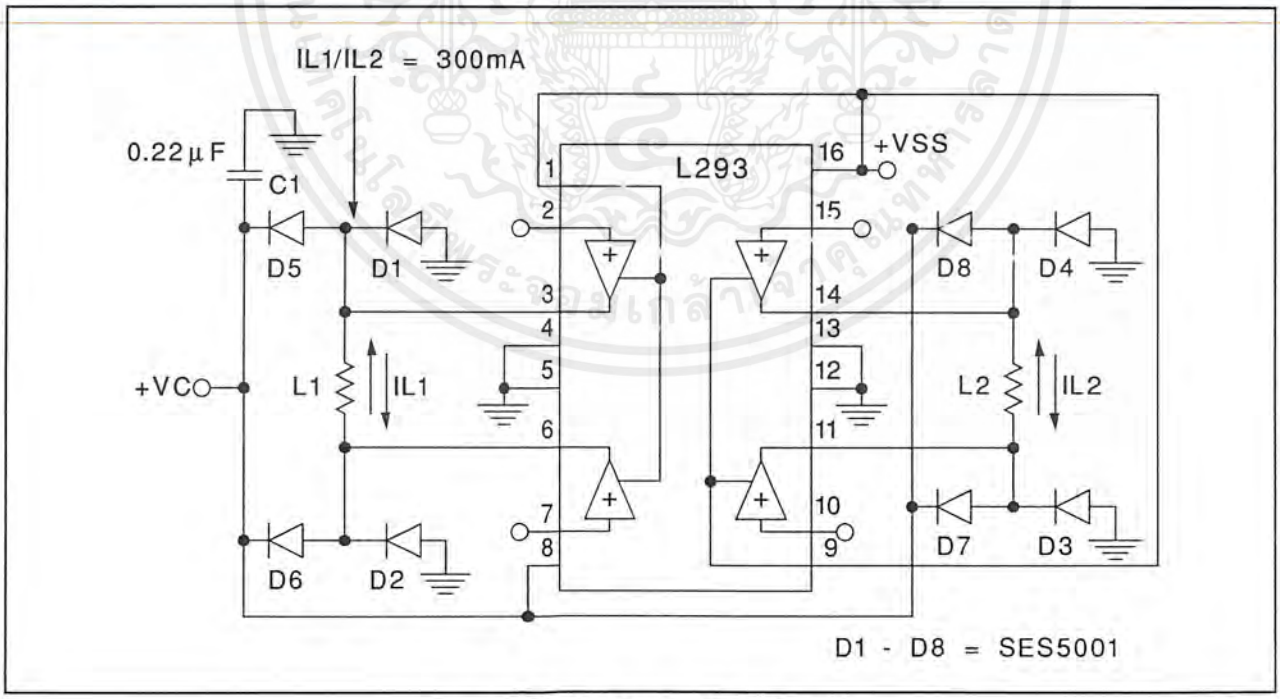


Figure 9: Bipolar Stepping Motor Control

MOUNTING INSTRUCTIONS

The Rthj-amp of the L293 can be reduced by soldering the GND pins to a suitable copper area of the printed circuit board or to an external heatsink.

The diagram of Figure 13 shows the maximum package power Ptot and the θJA as a function of the side "l" of two equal square copper areas having a thickness of 35µ (see Figure 10). In addition, it is possible to use an external heatsink (see Figure 11).

During soldering the pins' temperature must not exceed 260°C and the soldering time must not be longer than 12 seconds.

The external heatsink or printed circuit copper area must be connected to electrical ground.

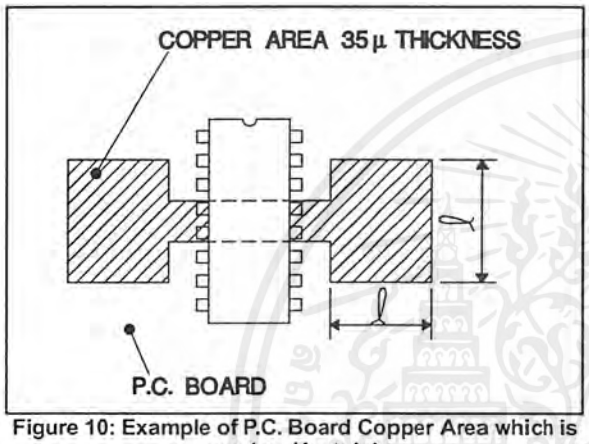


Figure 10: Example of P.C. Board Copper Area which is used as Heatsink

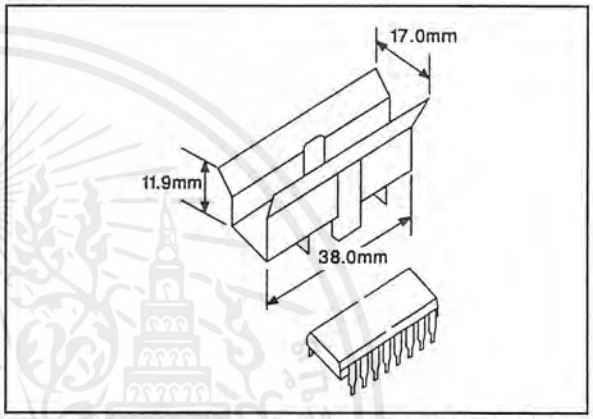


Figure 11: External Heatsink Mounting Example (θJA = 25°C/W)

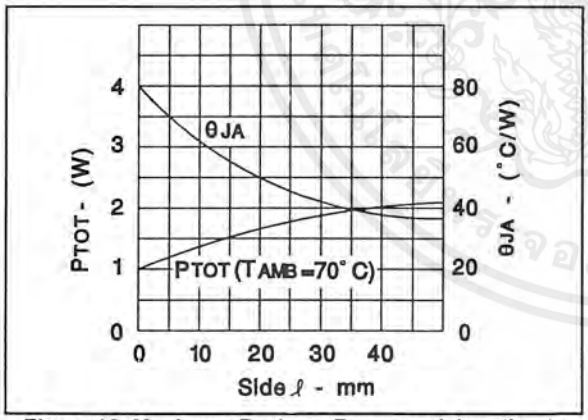


Figure 12: Maximum Package Power and Junction to Ambient Thermal Resistance

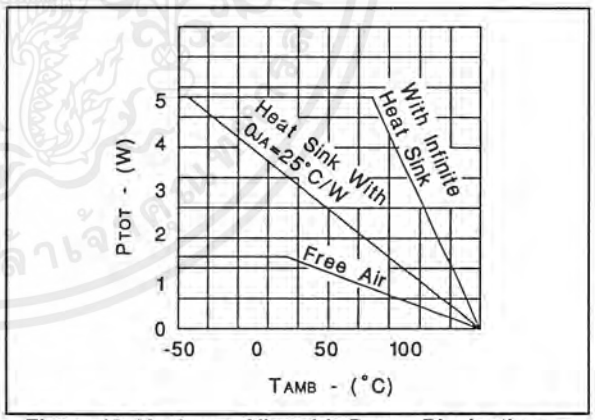


Figure 13: Maximum Allowable Power Dissipation vs Ambient Temperature

UNITRODE INTEGRATED CIRCUITS
7 CONTINENTAL BLVD. • MERRIMACK, NH 03054
TEL. (603) 424-2410 • FAX (603) 424-3460

National Semiconductor

January 2000

LM386
Low Voltage Audio Power Amplifier

General Description

The LM386 is a power amplifier designed for use in low voltage consumer applications. The gain is internally set to 20 to keep external part count low, but the addition of an external resistor and capacitor between pins 1 and 8 will increase the gain to any value up to 200.

The inputs are ground referenced while the output is automatically biased to one half the supply voltage. The quiescent power drain is only 24 milliwatts when operating from a 6 volt supply, making the LM386 ideal for battery operation.

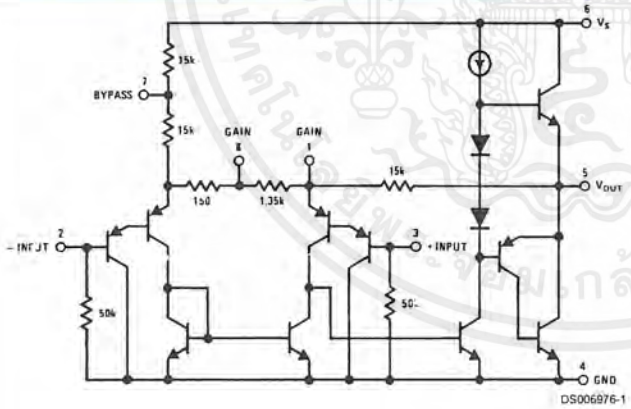
Features

- Battery operation
- Minimum external parts
- Wide supply voltage range: 4V–12V or 5V–18V
- Low quiescent current drain: 4 mA
- Voltage gains from 20 to 200
- Ground referenced input
- Self-centering output quiescent voltage
- Low distortion
- Available in 8 pin MSOP package

Applications

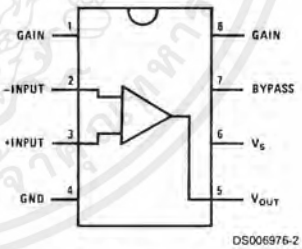
- AM-FM radio amplifiers
- Portable tape player amplifiers
- Intercoms
- TV sound systems
- Line drivers
- Ultrasonic drivers
- Small servo drivers
- Power converters

Equivalent Schematic and Connection Diagrams



DS006976-1

Small Outline, Molded Mini Small Outline, and Dual-In-Line Packages



DS006976-2

Top View
Order Number LM386M-1, LM386MM-1, LM386N-1, LM386N-3 or LM386N-4
See NS Package Number M08A, MUA08A or N08E

Absolute Maximum Ratings (Note 2)

If Military/Aerospace specified devices are required, please contact the National Semiconductor Sales Office/ Distributors for availability and specifications.

Supply Voltage	
(LM386N-1, -3, LM386M-1)	15V
Supply Voltage (LM386N-4)	22V
Package Dissipation (Note 3)	
(LM386N)	1.25W
(LM386M)	0.73W
(LM386MM-1)	0.595W
Input Voltage	±0.4V
Storage Temperature	−65°C to +150°C
Operating Temperature	0°C to +70°C
Junction Temperature	+150°C
Soldering Information	
Dual-In-Line Package	
Soldering (10 sec)	+260°C
Small Outline Package	
(SOIC and MSOP)	
Vapor Phase (60 sec)	+215°C
Infrared (15 sec)	+220°C

See AN-450 "Surface Mounting Methods and Their Effect on Product Reliability" for other methods of soldering surface mount devices.

Thermal Resistance	
θ_{JC} (DIP)	37°C/W
θ_{JA} (DIP)	107°C/W
θ_{JC} (SO Package)	35°C/W
θ_{JA} (SO Package)	172°C/W
θ_{JA} (MSOP)	210°C/W
θ_{JC} (MSOP)	56°C/W

Electrical Characteristics (Notes 1, 2)

T_A = 25°C

Parameter	Conditions	Min	Typ	Max	Units
Operating Supply Voltage (V_S)					
LM386N-1, -3, LM386M-1, LM386MM-1		4		12	V
LM386N-4		5		18	V
Quiescent Current (I_Q)	V_S = 6V, V_{IN} = 0		4	8	mA
Output Power (P_{OUT})					
LM386N-1, LM386M-1, LM386MM-1	V_S = 6V, R_L = 8Ω, THD = 10%	250	325		mW
LM386N-3	V_S = 9V, R_L = 8Ω, THD = 10%	500	700		mW
LM386N-4	V_S = 16V, R_L = 32Ω, THD = 10%	700	1000		mW
Voltage Gain (A_V)	V_S = 6V, f = 1 kHz		26		dB
	10 µF from Pin 1 to 8		46		dB
Bandwidth (BW)	V_S = 6V, Pins 1 and 8 Open		300		kHz
Total Harmonic Distortion (THD)	V_S = 6V, R_L = 8Ω, P_{OUT} = 125 mW f = 1 kHz, Pins 1 and 8 Open		0.2		%
Power Supply Rejection Ratio (PSRR)	V_S = 6V, f = 1 kHz, C_{BYPASS} = 10 µF Pins 1 and 8 Open, Referred to Output		50		dB
Input Resistance (R_{IN})			50		kΩ
Input Bias Current (I_{BIAS})	V_S = 6V, Pins 2 and 3 Open		250		nA

Note 1: All voltages are measured with respect to the ground pin, unless otherwise specified.

Note 2: Absolute Maximum Ratings indicate limits beyond which damage to the device may occur. Operating Ratings indicate conditions for which the device is functional, but do not guarantee specific performance limits. Electrical Characteristics state DC and AC electrical specifications under particular test conditions which guarantee specific performance limits. This assumes that the device is within the Operating Ratings. Specifications are not guaranteed for parameters where no limit is given, however, the typical value is a good indication of device performance.

Note 3: For operation in ambient temperatures above 25°C, the device must be derated based on a 150°C maximum junction temperature and 1) a thermal resistance of 107°C/W junction to ambient for the dual-in-line package and 2) a thermal resistance of 170°C/W for the small outline package.

Application Hints

GAIN CONTROL

To make the LM386 a more versatile amplifier, two pins (1 and 8) are provided for gain control. With pins 1 and 8 open the 1.35 kΩ resistor sets the gain at 20 (26 dB). If a capacitor is put from pin 1 to 8, bypassing the 1.35 kΩ resistor, the gain will go up to 200 (46 dB). If a resistor is placed in series with the capacitor, the gain can be set to any value from 20 to 200. Gain control can also be done by capacitively coupling a resistor (or FET) from pin 1 to ground.

Additional external components can be placed in parallel with the internal feedback resistors to tailor the gain and frequency response for individual applications. For example, we can compensate poor speaker bass response by frequency shaping the feedback path. This is done with a series RC from pin 1 to 5 (paralleling the internal 15 kΩ resistor). For 6 dB effective bass boost: R ≅ 15 kΩ, the lowest value for good stable operation is R = 10 kΩ if pin 8 is open. If pins 1 and 8 are bypassed then R as low as 2 kΩ can be used. This restriction is because the amplifier is only compensated for closed-loop gains greater than 9.

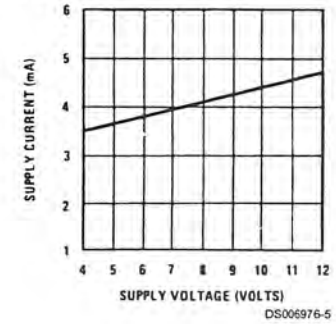
INPUT BIASING

The schematic shows that both inputs are biased to ground with a 50 kΩ resistor. The base current of the input transistors is about 250 nA, so the inputs are at about 12.5 mV when left open. If the dc source resistance driving the LM386 is higher than 250 kΩ it will contribute very little additional offset (about 2.5 mV at the input, 50 mV at the output). If the dc source resistance is less than 10 kΩ, then shorting the unused input to ground will keep the offset low (about 2.5 mV at the input, 50 mV at the output). For dc source resistances between these values we can eliminate excess offset by putting a resistor from the unused input to ground, equal in value to the dc source resistance. Of course all offset problems are eliminated if the input is capacitively coupled.

When using the LM386 with higher gains (bypassing the 1.35 kΩ resistor between pins 1 and 8) it is necessary to bypass the unused input, preventing degradation of gain and possible instabilities. This is done with a 0.1 µF capacitor or a short to ground depending on the dc source resistance on the driven input.

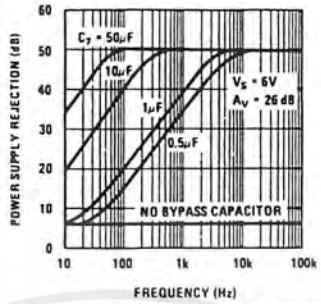
Typical Performance Characteristics

Quiescent Supply Current vs Supply Voltage



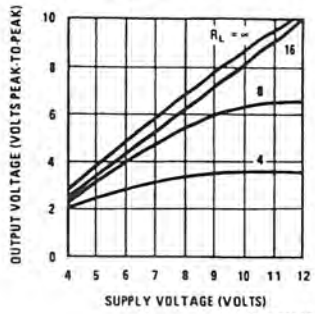
DS006976-5

Power Supply Rejection Ratio (Referred to the Output) vs Frequency



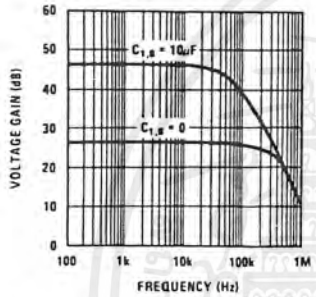
DS006976-12

Peak-to-Peak Output Voltage Swing vs Supply Voltage



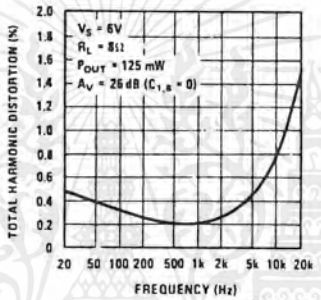
DS006976-13

Voltage Gain vs Frequency



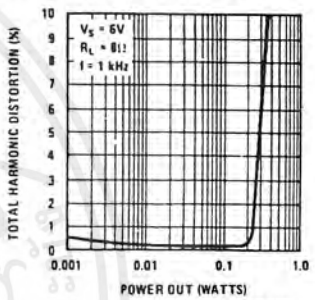
DS006976-14

Distortion vs Frequency



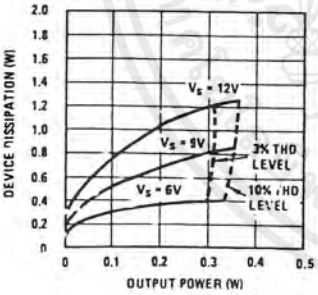
DS006976-15

Distortion vs Output Power



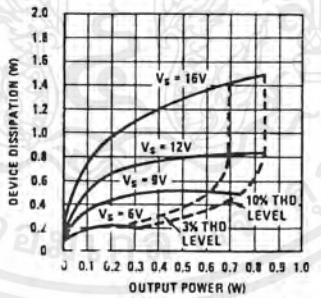
DS006976-16

Device Dissipation vs Output Power—4Ω Load



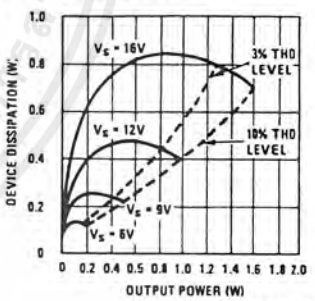
DS006976-17

Device Dissipation vs Output Power—8Ω Load



DS006976-18

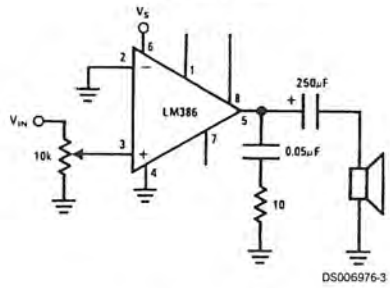
Device Dissipation vs Output Power—16Ω Load



DS006976-19

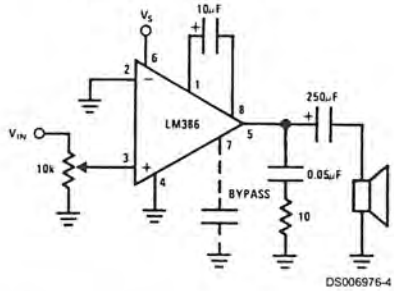
Typical Applications

Amplifier with Gain = 20 Minimum Parts



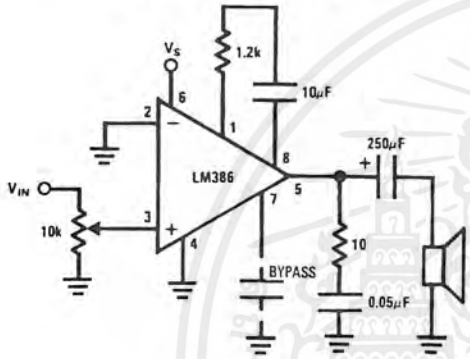
DS006976-3

Amplifier with Gain = 200



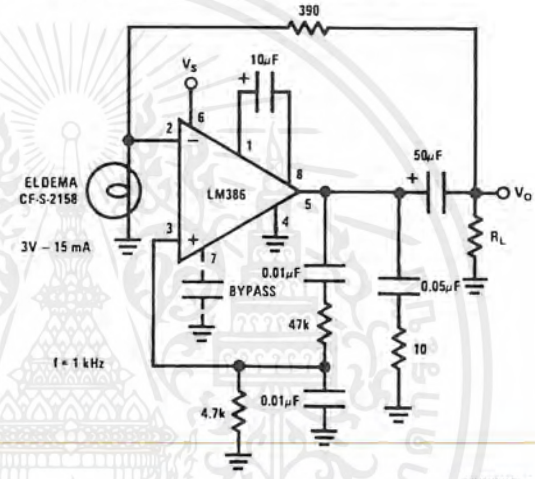
DS006976-4

Amplifier with Gain = 50



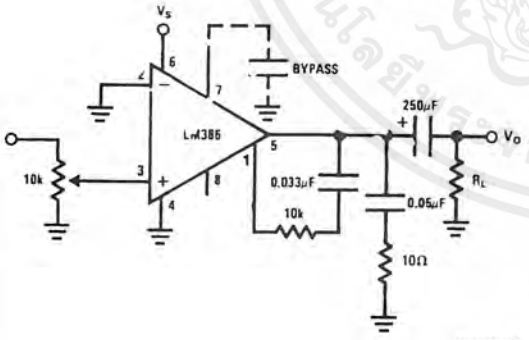
DS006976-6

Low Distortion Power Wienbridge Oscillator



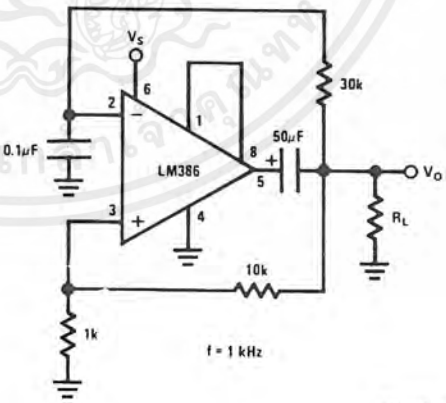
DS006976-7

Amplifier with Bass Boost



DS006976-8

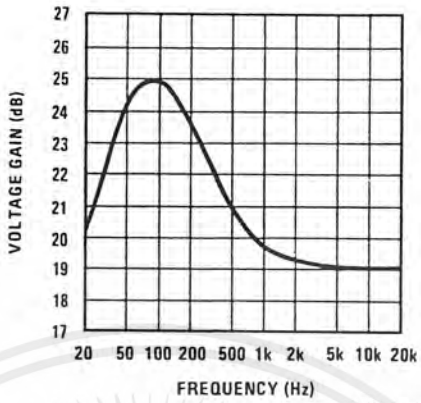
Square Wave Oscillator



DS006976-9

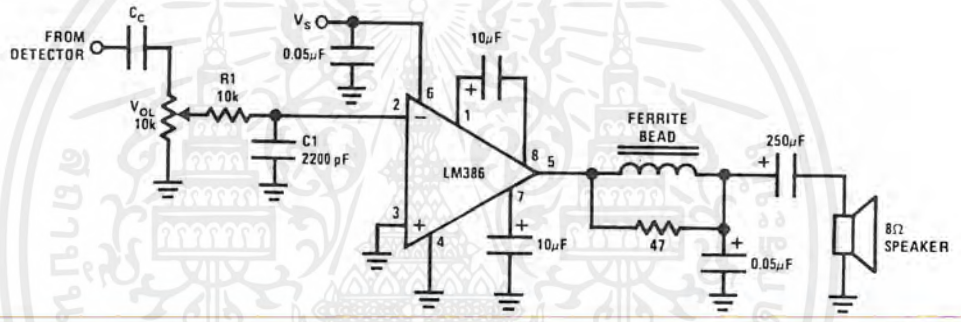
Typical Applications (Continued)

Frequency Response with Bass Boost



DS006976-10

AM Radio Power Amplifier



DS006976-11

Note 4: Twist Supply lead and supply ground very tightly.

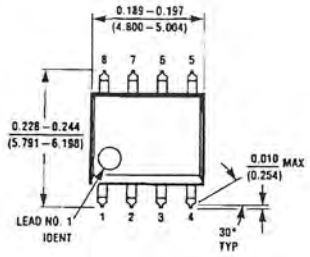
Note 5: Twist speaker lead and ground very tightly.

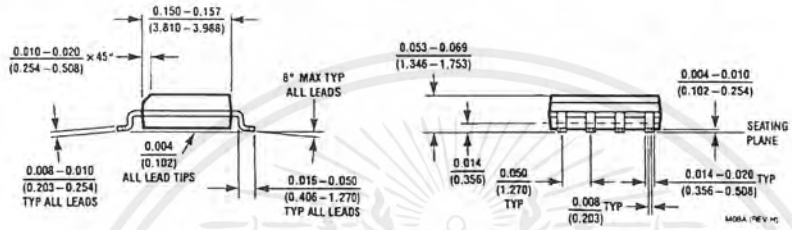
Note 6: Ferrite bead in Ferroxcube K5-001-001/3B with 3 turns of wire.

Note 7: R1C1 band limits input signals.

Note 8: All components must be spaced very closely to IC.

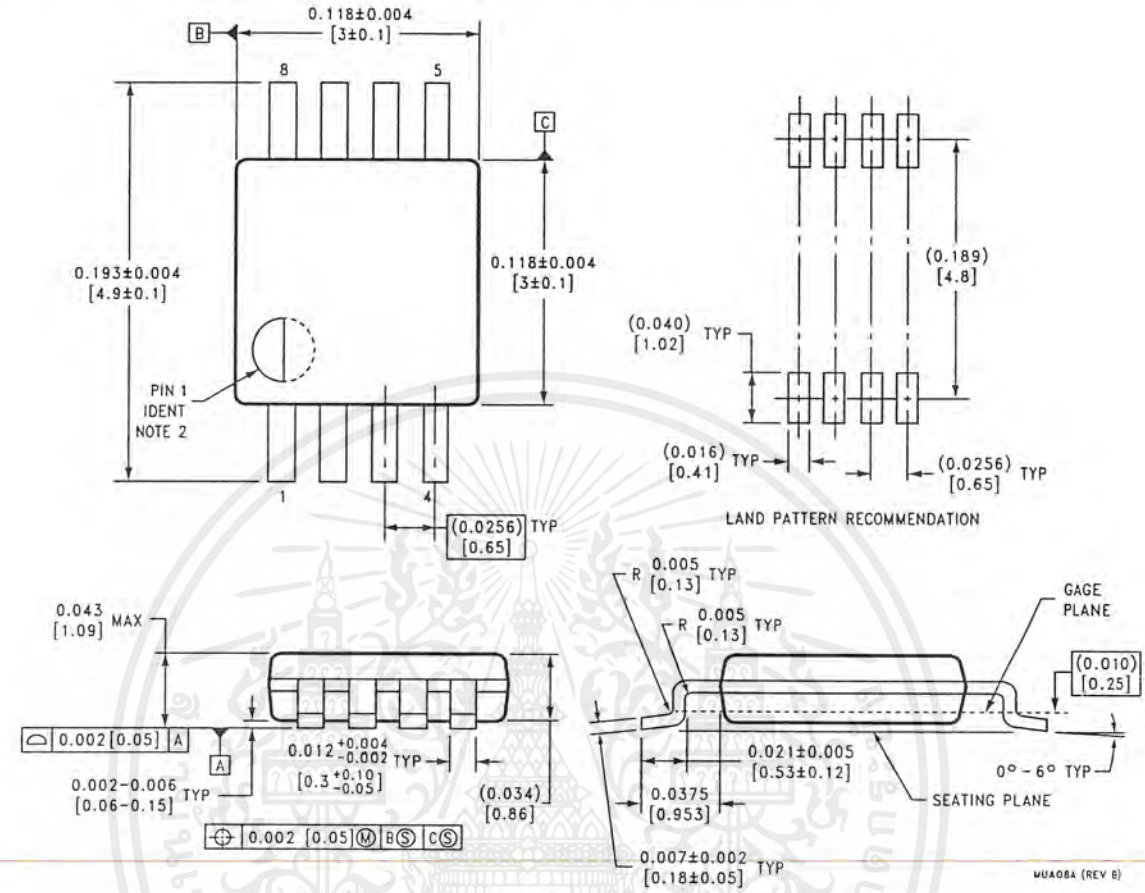
Physical Dimensions inches (millimeters) unless otherwise noted





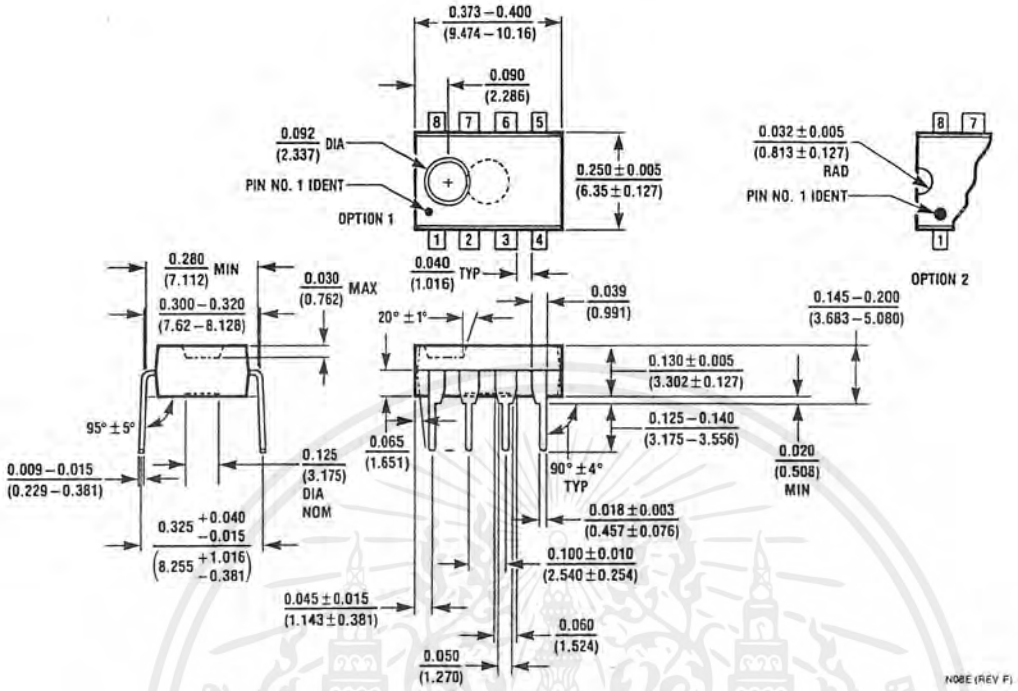
SO Package (M)
Order Number LM386M-1
NS Package Number M08A

Physical Dimensions inches (millimeters) unless otherwise noted (Continued)



8-Lead (0.118" Wide) Molded Mini Small Outline Package
Order Number LM386MM-1
NS Package Number MUA08A

Physical Dimensions inches (millimeters) unless otherwise noted (Continued)



Dual-In-Line Package (N)
Order Number LM386N-1, LM386N-3 or LM386N-4
NS Package Number N08E

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
Americas
Tel: 1-800-272-9959
Fax: 1-800-737-7018
Email: support@nsc.com

www.national.com

National Semiconductor Europe
Fax: +49 (0) 1 80-530 85 86
Email: europe.support@nsc.com
Deutsch Tel: +49 (0) 1 80-530 85 85
English Tel: +49 (0) 1 80-532 78 32
Français Tel: +49 (0) 1 80-532 93 58
Italiano Tel: +49 (0) 1 80-534 16 80

National Semiconductor Asia Pacific Customer Response Group
Tel: 65-2544466
Fax: 65-2504466
Email: sea.support@nsc.com

National Semiconductor Japan Ltd.
Tel: 81-3-5639-7560
Fax: 81-3-5639-7507

GP1U58Y Series

IR Detecting Unit For Remote Control

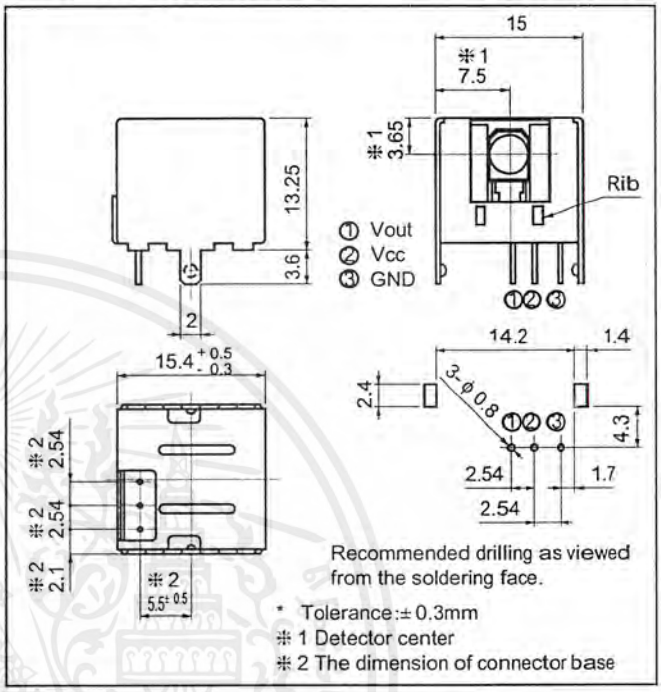
■ Features

1. Less sensitive to a fluorescent lamp driven by inverter
2. Various B.P.F. (Band Pass Filter) frequency
3. Built-in voltage regulator circuit

■ Applications

1. TVs
2. VCRs
3. Audio equipment
4. Air conditioners
5. CATV set top boxes
6. BS receivers
7. Multi-media equipments

■ Outline Dimensions (Unit : mm)



■ Absolute Maximum Ratings (Ta= 25°C)

Parameter	Symbol	Rating	Unit
Supply voltage	V_{CC}	0 to 6.3	V
*1 Operating temperature	T_{opr}	- 10 to + 70	°C
Storage temperature	T_{stg}	- 20 to + 70	°C
*2 Soldering temperature	T_{sol}	260	°C

*1 No dew formation
*2 For 5 seconds

■ Recommended Operating Conditions

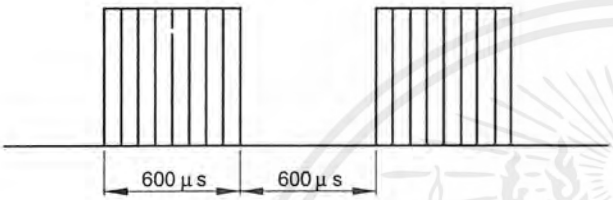
Parameter	Symbol	Value	Unit
Supply voltage	V_{CC}	4.7 to 5.3	V

■ Electrical Characteristics

(Ta= 25°C, V_{CC}= +5V)

Parameter	Symbol	Conditions	MIN.	TYP.	MAX.	Unit
Dissipation current	I_{CC}	No input light	-	-	5.0	mA
High level output voltage	V_{OH}	*3	V_{CC} - 0.5	-	-	V
Low level output voltage	V_{OL}		-	-	0.45	V
High level pulse width	T_1		400	-	800	μs
Low level pulse width	T_2		400	-	800	
B.P.F. center frequency	f_O	-	-	*4 40	-	kHz

*3 The burst wave as shown in the following figure shall be transmitted by the transmitter shown in Fig. 1.
*4 Diversified models with a different B.P.F frequency, as shown in a separate table, are also available.

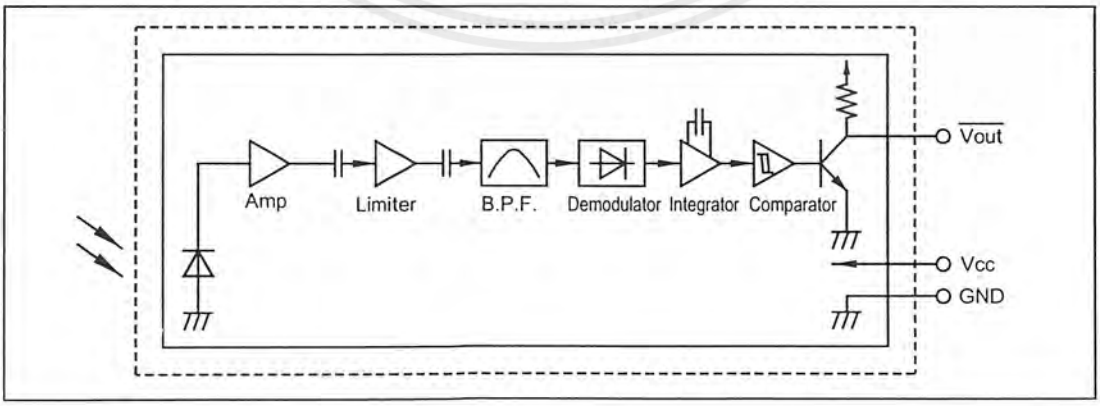


The value of f_O is shown in a separate table.
Duty 50%

■ Model Line-up

Model No.	B.P.F. frequency	Unit
GP1U58Y	40	kHz
GP1U580Y	36	
GP1U581Y	38	
GP1U582Y	36.7	
GP1U583Y	32.75	
GP1U587Y	56.8	

■ Internal Block Diagram



■ Performance

Using the transmitter shown in Fig. 1, the output signal of the light detecting unit is good enough to meet the following items in the standard optical system in Fig. 2.

(1) Linear reception distance characteristics

When L= 0.2 to 8m, Ee<10 lx and $\phi = 0°$ in Fig. 2, the output signal shall meet the electrical characteristcs in the attached list.

(2) Sensitivity angle reception distance characteristics

When L= 0.2 to 6m, Ee< 10 lx and $\phi <= 30°$ in Fig. 2, the output signal shall meet the electrical characteristics in the attached list.

(3) Anti outer peripheral light reception distance characteristics

When L= 0.2 to 4m, Ee<=300 lx and $\phi = 0°$ in Fig. 2, the output signal shall meet the electrical characteristics in the attached list.

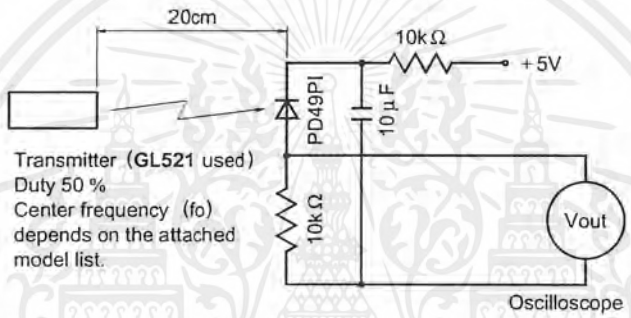


Fig. 1. Transmitter

In the above figure, the transmitter should be set so that the output V_{out} can be $40mV_{PP}$.
However, the **PD49PI** to be used here should be of the short-circuit current $I_{SC} = 2.6\mu A$ at E_V= 100 lx.
(E_V is an illuminance by CIE standard light source A (tungsten lamp).)

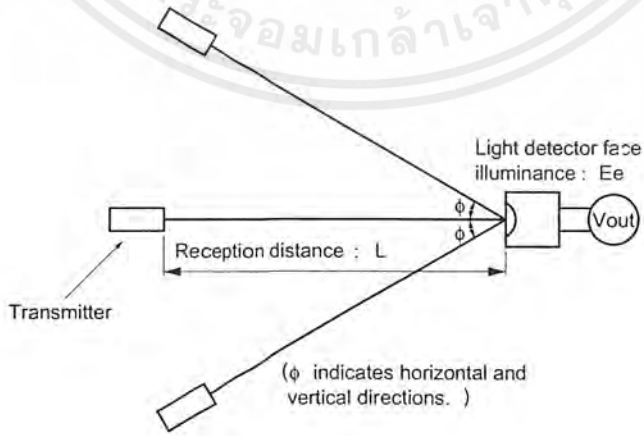


Fig. 2. Standard optical system

■ Precautions for Use

(1) Use the light emitting unit (remote control transmitter), in consideration of performance, characteristics and operating condition of light emitting device and the characteristics of the light detecting unit.

(2) Pay attention to a malfunction of the light detecting unit when the surface is stained with dust and refuse. Care must be taken not to touch the light detector surface. If it should be dirty, wipe off with soft cloth so as to prevent scratch. In case some solvents are required, use metyl alcohol, ethyl alcohol or isoprophyl alcohol. Also, protect the light detecting unit against flux and others.

(3) The shield case shall be grounded on PWB pattern.

(4) Do not apply unnecessary force to the terminals and case form outside.

(5) Do not push the light detector surface (photodiode) from outside.

(6) To avoid the electorstatic breakdown of IC, handle the unit under the condition of grounding with human body, soldering iron, etc.

(7) In case of adopting the infrared light detecting unit for the wireless remote control, use it in accordance with the transmission scheme and the signal format recommended in "Countermeasures for malfunction prevention of home appliances with infrared remote control" issued form Japan Association of Electrical Home Appliances (AEHA) in July 1987.

(8) As for other general cautions, refer to the chapter " Precautions for Use" (Page 78 to 93).

บรรณานุกรม

1. Joseph L. Jones , Anita M. Flynn, "Mobile Robots: Inspiration to Implementation", A K Peters Wellesley, Massachusetts,1993
2. ชัยวัฒน์ ลิ้มพรจิตรวิไล , "เรียนรู้และการใช้งานไมโครคอนโทรลเลอร์ 68HC11" ซีเอ็ดยูเคชั่น, กรุงเทพมหานคร,พ.ศ.2538
3. Fred Martin, "The 6.270 Robot Builder's Guide"
4. Gordon Mccomb ,"The Robot Builder's Bonanza: 99 Inexpensive Robotics Projects", Tab book Inc Blue Ridge Summit, PA
5. คู่มือ MC68HC11 HCMOS Single-Chip Microcontroller