การประยุกต์การใช้งานสำหรับมัลติคอร์ไมโครคอนโทรลเลอร์

APPLICATIONS FOR MULTICORE MICROCONTROLLER

ภัทธิยา คงสมบูรณ์

วิทวัส ศิริบุญลักษณ์กุล

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

สาขาวิชาวิศวกรรมศาสตร์อิเล็กทรอนิกส์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2552

APPLICATIONS FOR MULTICORE MICROCONTROLLER

PATTIYA KONGSOMBOON

WITTAWAS SIRIBOONLUKKOOL

THIS THESIS IS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF ENGINEERING IN ELECTRONICS ENGINEERING

FACULTY OF ENGINEERING

ปริญญานิพนธ์ ปีการศึกษา 2552

ภาควิชาอิเล็กทรอนิกส์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การประยุกต์การใช้งานสำหรับมัลติคอร์ไมโครคอนโทรลเลอร์

(Applications for multicore microcontroller)

ผู้จัดทำ

นางสาวภัททิยา คงสมบูรณ์ รหัส 49010695

นายวิทวัส ศิริบุญลักษณ์กุล รหัส 49010894

ลงชื่อ.....อาจารย์ที่ปรึกษา

(รศ.ดร.มนัส สังวรศิลป์)

การประยุกต์การใช้งานสำหรับมัลติคอร์ไมโครคอนโทรลเลอร์

นางสาวภัททิยา คงสมบูรณ์ รหัส49010695
นายวิทวัส ศิริบุญลักษณ์กุล รหัส 49010894
รศ.ดร.มนัส สังวรศิลป์ อาจารย์ที่ปรึกษา
ปีการศึกษา 2552

บทคัดย่อ

โครงงานนี้เป็น การศึกษาเรื่องการประยุกต์การใช้งานสำหรับมัลติคอร์ไมโครคอนโทรลเลอร์ ซึ่งเป็นการประยุกต์ใช้งานไมโครคอนโทรลเลอร์ขนาด 32 บิต ที่มีสถาปัตยกรรมที่พิเศษคือ มีซีพียูภายใน 8 ตัวหรือ 8 ค๊อก ที่สามารถทำงานแยกจากกันได้อย่างอิสระ หรือจะทำงานร่วมกันก็ได้ โดยได้ทำการศึกษาการใช้งานเบื้องต้นของมัลติคอร์ไมโครคอนโทรลเลอร์ ศึกษาการทำงานของระหว่างซีพียูแต่ละตัว และการนำไปประยุกต์ใช้งาน ซึ่งการพัฒนาโปรแกรมนั้นสามารถทำได้ด้วยโปรแกรมภาษาใหม่ ที่เรียกว่าภาษาสปิน และภาษาแอสแซมบลี โดยภาษาสปินนั้นเป็นภาษาสูงที่มีการทำงานแบบออบเจ็กต์ โดยได้ทำการออกแบบเป็นเครื่องเตือนรถไฟบริเวณทางข้าม

Applications for multicore microcontroller

Miss. Pattiya Kongsomboon ID49010695

Mr. Wittawas Siriboonlukkool ID49010894

Assoc. Prof. Manas Sangworasil Advisor

Educational Year 2552

Abstract

This project is study about Applications for multicore microcontroller. We applied multiprocessor microcontroller, The Propeller chip, is designed to provide high-speed processing for embedded systems while maintaining low current consumption and a small physical footprint. In addition to being fast, the Propeller chip provides flexibility and power through its eight processors, called cogs, which can perform simultaneous tasks independently or cooperatively. Two programming languages are available: Spin (a high-level object-based language) and Propeller Assembly. We used it to make the warning train at the crossroad.

กิตติกรรมประกาศ

รายงานฉบับนี้สำเร็จได้ด้วยความกรุณาจากอาจารย์ที่ปรึกษา รศ.ดร.มนัส สังวรศิลป์ ที่ให้คำปรึกษา คำแนะนำ และช่วยแก้ปัญหา ตลอดจนให้ความรู้และประสบการณ์ที่ดีแก่ข้าพเจ้า จนสำเร็จลุล่วงตามวัตถุประสงค์

ขอบคุณนายจีรพัฒน์ จักรโนวรรณ วิศวกรรมศาสตร์คอมพิวเตอร์ ที่ช่วยให้คำแนะนำ

นอกจากนี้ขอขอบพระคุณทุกๆท่านที่ให้ความช่วยเหลือ ตลอดจนให้คำแนะนำต่างๆจนทำให้รายงานฉบับนี้สำเร็จโดยสมบูรณ์ได้

คุณค่าและประโยชน์ อันพึงมีในรายงานฉบับนี้ ข้าพเจ้าขอมอบให้กับท่านผู้มีพระคุณทุกท่าน

นางสาวภัททิยา คงสมบูรณ์

นายวิทวัส ศิริบุญลักษณ์กุล

ผู้จัดทำ

สารบัญ

สารบัญ(ต่อ)

สารบัญรูปภาพ

สารบัญรูปภาพ(ต่อ)

สารบัญตาราง

บทที่ 1
บทนำ

1.1 แนวความคิดเริ่มต้น

ในปัจจุบันนี้ไมโครคอนโทรลเลอร์ได้มีการนำไปใช้กันอย่างแพร่หลาย และมีความสำคัญมากขึ้นเรื่อยๆ ทั้งนี้เนื่องมาจากสามารถนำไปประยุกต์ใช้งานได้อย่างกว้างขวาง โดยมักจะเป็นการนำไปใช้ฝังในระบบของอุปกรณ์อื่น ๆ (ระบบสมองกลฝังตัว: Embedded Systems) เพื่อใช้ควบคุมการทำงานบางอย่าง เช่น ใช้ในรถยนต์, เตาอบไมโครเวฟ, เครื่องปรับอากาศ, เครื่องซักผ้าอัตโนมัติ เป็นต้น เพราะว่าไมโครคอนโทรลเลอร์มีข้อดีเหมาะสมต่อการใช้ในงานควบคุมหลายประการ เช่น ชิพไอซีและระบบที่ได้มีขนาดเล็ก ระบบที่ได้มีราคาถูกกว่าการใช้ชิพไมโครโพรเซสเซอร์ วงจรที่ได้จะมีความซับซ้อนน้อย ช่วยลดข้อผิดพลาดที่อาจจะเกิดขึ้นได้ในการต่อวงจร มีคุณสมบัติเพิ่มเติมสำหรับงานควบคุมโดยเฉพาะซึ่งใช้งานได้ง่าย และช่วยลดระยะเวลาในการพัฒนาระบบได้

ด้วยเหตุนี้ผู้จัดทำจึงได้ศึกษาการทำงานของมัลติคอร์ไมโครคอนโทรลเลอร์ ซึ่งเป็นไมโครคอนโทรลเลอร์ที่ได้รับการพัฒนาขึ้นมาในหลายๆด้าน ทั้งจำนวนซีพียูที่เพิ่มขึ้น ความเร็วจึงมากขึ้น ซึ่งสามารถรองรับการใช้งานที่มากขึ้นในอนาคตได้ โดยผู้จัดทำได้มัลติคอร์ไมโครคอนโทรลเลอร์มาประยุกต์ใช้เป็นเครื่องเตือนรถไฟอัตโนมัติ โดยได้ทำเป็นแบบจำลอง ซึ่งผู้จัดทำคาดหวังว่าจะมีประโยชน์แก่สังคม และประเทศต่อไป

1.2วัตถุประสงค์

1.2.1 เพื่อศึกษา และเรียนรู้การใช้งานของมัลติคอร์ไมโครคอนโทรลเลอร์

1.2.2 เพื่อศึกษาการเขียนโปรแกรมควบคุมการทำงาน มัลติคอร์ไมโครคอนโทรลเลอร์ ด้วยภาษาสปิน

1.2.3เขียนโปรแกรมระบบเตือนรถไฟอัตโนมัติ

1.3ขอบเขตของโครงงาน

1.3.1 ศึกษาการทำงานของมัลติคอร์ไมโครคอนโทรลเลอร์ ของบริษัทPropeller

1.3.2 ออกแบบระบบการทำงานของชุดตรวจจับรถไฟ และส่วนแสดงผล

1.3.3 เขียนโปรแกรมควบคุมการทำงานระบบเตือนรถไฟอัตโนมัติ

1.3.4 ทดสอบเพื่อให้ได้ตามวัตถุประสงค์

1.4ประโยชน์ และผลที่คาดว่าจะได้รับ

- มีความรู้ความเข้าใจในระบบการทำงานของมัลติคอร์ไมโครคอนโทรลเลอร์มากขึ้น

- เป็นการส่งเสริมให้ผู้ทำการศึกษามีความคิดริเริ่มสร้างสรรค์ และเกิดทักษะซึ่งสามารถแก้ไขปัญหาได้

- สามารถประยุกต์ใช้ความก้าวหน้าของเทคโนโลยีใหม่ๆให้เกิดประโยชน์

-สามารถนำไปพัฒนาให้เหมาะสมกับแต่ละท้องที่ เพื่อความปลอดภัยแก่ประชาชนในการเดินทางสัญจร

1.5 เนื้อหาโดยสังเขป

บทที่ 1 กล่าวถึงแนวความคิดเริ่มต้น วัตถุประสงค์ ขอบเขต และประโยชน์ที่ได้รับ

บทที่ 2 กล่าวถึงทฤษฎี และหลักการ

บทที่ 3 กล่าวถึงการออกแบบ

บทที่ 4 กล่าวถึงการทดลอง และผลการทดลอง

บทที่ 5 กล่าวถึงบทสรุป ปัญหา และแนวทางแก้ไขปัญหา

บทที่ 2
ทฤษฎีและหลักการ

2.1 ไมโครคอนโทรลเลอร์ คืออะไร?

ไมโครคอนโทรลเลอร์ (Microcontroller) มาจากคำ 2 คำ คำหนึ่งคือ ไมโคร (Micro) หมายถึงขนาดเล็ก และคำว่า คอนโทรลเลอร์ (controller) หมายถึงตัวควบคุมหรืออุปกรณ์ควบคุม ดังนั้น ไมโครคอนโทรลเลอร์ จึงหมายถึงอุปกรณ์ควบคุมขนาดเล็ก แต่ในตัวอุปกรณ์ควบคุมขนาดเล็กนี้ ได้บรรจุความสามารถที่คล้ายคลึงกับระบบคอมพิวเตอร์ ที่คนโดยส่วนใหญ่คุ้นเคย กล่าวคือภายในไมโครคอนโทรลเลอร์ได้รวมเอาซีพียู, หน่วยความจำ และพอร์ต ซึ่งเป็นส่วนประกอบหลักสำคัญของระบบคอมพิวเตอร์เข้าไว้ด้วยกัน โดยทำการบรรจุเข้าไว้ในตัวถังเดียวกัน

โครงสร้างโดยทั่วไปของไมโครคอนโทรลเลอร์

โครงสร้างโดยทั่วไป ของไมโครคอนโทรลเลอร์นั้น สามารถแบ่งออกมาได้เป็น 5 ส่วนใหญ่ๆ ดังต่อไปนี้

1.หน่วยประมวลผลกลางหรือซีพียู (CPU: Central Processing Unit)

2.หน่วยความจำ (Memory) สามารถแบ่งออกเป็น 2 ส่วน คือ หน่วยความจำที่มีไว้สำหรับเก็บโปรแกรมหลัก (Program Memory) เปรียบเสมือนฮาร์ดดิสก์ของเครื่องคอมพิวเตอร์ตั้งโต๊ะ คือข้อมูลใดๆ ที่ถูกเก็บไว้ในนี้จะไม่สูญหายไปแม้ไม่มีไฟเลี้ยง อีกส่วนหนึ่งคือหน่วยความจำข้อมูล (Data Memory) ใช้เป็นเหมือนกระดาษทดในการคำนวณของซีพียู และเป็นที่พักข้อมูลชั่วคราวขณะทำงาน แต่หากไม่มีไฟเลี้ยง ข้อมูลก็จะหายไปคล้ายกับหน่วยความแรม (RAM) ในเครื่องคอมพิวเตอร์ทั่วๆ ไป

แต่สำหรับไมโครคอนโทรลเลอร์สมัยใหม่ หน่วยความจำข้อมูลจะมีทั้งที่เป็นหน่วยความจำแรม ซึ่งข้อมูลจะหายไปเมื่อไม่มีไฟเลี้ยง และเป็นอีอีพรอม (EEPROM: Erasable Electrically Read-Only Memory) ซึ่งสามารถเก็บข้อมูลได้แม้ไม่มีไฟเลี้ยง

3.ส่วนติดต่อกับอุปกรณ์ภายนอก หรือพอร์ต (Port) มี 2 ลักษณะคือ พอร์ตอินพุต (Input Port) และพอร์ตเอาต์พุต (Output Port) ส่วนนี้จะใช้ในการเชื่อมต่อกับอุปกรณ์ภายนอก ถือว่าเป็นส่วนที่สำคัญมาก ใช้ร่วมกันระหว่างพอร์ตอินพุตเพื่อรับสัญญาณ อาจจะด้วยการกดสวิตช์ เพื่อนำไปประมวลผลและส่งไปพอร์ตเอาต์พุต เพื่อแสดงผลเช่น การติดสว่างของหลอดไฟ เป็นต้น

4. ช่องทางเดินของสัญญาณ หรือบัส (BUS) คือเส้นทางการแลกเปลี่ยนสัญญาณข้อมูลระหว่าง ซีพียู หน่วยความจำและพอร์ต เป็นลักษณะของสายสัญญาณ จำนวนมากอยู่ภายในตัวไมโครคอนโทรลเลอร์ โดยแบ่งเป็นบัสข้อมูล (Data Bus), บัสแอดเดรส (Address Bus) และบัสควบคุม (Control Bus)

- บัสข้อมูลเป็นสายสัญญาณที่บรรจุข้อมูล เพื่อการประมวลผลทั้งหมด ขนาดของบัสจะขึ้นอยู่กับความสามารถการประมวลผลของซีพียู สำหรับในงานทั่วๆ ไป ขนาดของบัสข้อมูลจะเป็น 8 บิต และในปัจจุบันได้มีการพัฒนาขึ้นมาจนถึง 16 บิต, 32 บิต และ 64 บิต

- บัสแอดเดรสเป็นสายสัญญาณที่บรรจุค่าตำแหน่งของหน่วยความจำ โดยการติดต่อกับหน่วยความจำนั้น ซีพียู ต้องกำหนดตำแหน่งที่ต้องการอ่านหรือเขียนก่อน ดังนั้นจำนวนสายสัญญาณของแอดเดรสจึงต้องมีจำนวนมาก ยิ่งมากเท่าไหร่ ก็จะเป็นการแสดง ขนาดของหน่วยความจำที่ไมโครคอนโทรลเลอร์สามารถติดต่อได้ โดยสามารถคำนวนได้จาก

จำนวนแอดเดรสของหน่วยความจำ = 2 ยกกำลัง n (n คือจำนวนของเส้นทาง)

ยกตัวอย่าง ไมโครคอนโทรลเลอร์ตัวหนึ่งมีสายแอดเดรส 10 เส้น ดังนั้น ไมโครคอนโทรลเลอร์ตัวนี้ สามารถติดต่อกับหน่วยความจำได้ 2 ยกกำลัง 10 = 1,024 ตำแหน่ง

หากต้องการทราบความจุของหน่วยความจำจริงๆ จะต้องทราบถึงขนาดของบัสข้อมูลก่อนว่าเป็นเท่าใด หากเป็น 8 บิต ความจุของหน่วยความจำที่มีสายแอดเดรส 10 เส้น จะเท่ากับ 8x1024 = 8,192 บิต และ 1 กิโลไบต์ เท่ากับ 1,024 ไบต์ ดังนั้นไมโครคอน โทรลเลอร์ดังกล่าว จึงมีความจุของหน่วยความจำเท่ากับ 8,192 บิต หรือ 1,024 ไบต์ หรือ 1 กิโลไบต์

- บัสควบคุมเป็นกลุ่มของสายสัญญาณควบคุมการติดต่อทั้งหมดของซีพียูกับหน่วยความจำและพอร์ต สำหรับสายสัญญาณเลือกควบคุมหลักได้แก่ สายสัญญาณเลือก-อ่าน-เขียนหน่วยความจำ สายสัญญาณเลือกเลือกอ่าน-เขียน ข้อมูล กับพอร์ต

5.วงจรกำเนิดสัญญาณนาฬิกา นับเป็นส่วนประกอบที่สำคัญมากอีกส่วนหนึ่ง เนื่องจากการทำงานที่เกิดขึ้นในตัวไมโครคอนโทรลเลอร์ จะขึ้นอยู่กับการกำหนดจังหวะ หากสัญญาณนาฬิกามีความถี่สูง จังหวะการทำงานก็จะสามารถทำได้ถี่ขึ้นส่งผลให้ไมโครคอนโทรลเลอร์จัวนั้น มีความเร็วในการประมวลผลสูงตามไปด้วย

2.2 Propeller มัลติคอร์ไมโครคอนโทรลเลอร์

Propeller หรือ โพรเพลเลอร์ คือชื่อของไมโครคอนโทรลเลอร์ 32บิต อนุกรมใหม่จาก Parallax ที่มีสถาปัตยกรรมที่พิเศษคือ มีซีพียูภายใน 8 ตัวหรือ8 ค็อก (cog) ที่สามารถทำงานแยกจากกันอย่างอิสระ หรือร่วมกันทำงานก็ได้ นับเป็นแนวคิดใหม่ที่ร่วมสมัย และนับเป็นการปฏิวัติวงการไมโครคอนโทรลเลอร์ 32บิต ครั้งสำคัญ

การพัฒนาโปรแกรมสำหรับโพรเพลเลอร์ทำได้ด้วยโปรแกรมภาษาใหม่ ที่เรียกว่า สปิน (Spin) และภาษาแอสเซมบลี โดยภาษาสปินนั้นเป็นภาษาสูงที่มีการทำงานแบบออบเจ็กต์ (high-level object-based language)

2.2.1 คุณสมบัติเด่นของโพรเพลเลอร์

- ประกอบด้วย 8ซีพียู หรือเรียกว่า 8ค็อก ที่สามารถทำงานได้พร้อมกันอย่างเป็นอิสระ โดยมีการควบคุมการใช้ทรัพยากรร่วมกันผ่านทางตัวเชื่อมโยงกลางหรือ central hub

- มีความเร็วในการทำงานสูง และด้วยการทำงานที่เป็นอิสระของแต่ละค็อก ทำให้สามารถรองรับการตอบสนองต่อเหตุการต่างๆ ที่เกิดขึ้นในระบบได้อย่างรวดเร็วเพียงพอ จึงไม่ต้องใช้กระบวนการอินเตอรัปต์ช่วย ทำให้การเขียนโปรแกรมเพื่อรองรับการทำงานในแต่ละเหตุการณ์ลดความซับซ้อนลงได้อย่างมาก

- มีการใช้สัญญาณนาฬิกาของระบบร่วมกัน ทำให้สามารถอ้างอิงค่าเวลาหลักเดียวกันได้ ทำให้การทำงานของแต่ละค็อกมีจังหวะที่สอดคล้องกัน

- ภาษาสปินซึ่งมีลักษณะเป็นโปรแกรมภาษาสูงแบบออบเจ็กต์ได้รับการออกแบบให้ง่ายต่อการเรียนรู้ และสามารถรองรับการทำงานของโพรเพลเลอร์ได้อย่างมีประสิทธิภาพสูงสุด

-ภาษาแอสเซมบลีของโพรเพลเลอร์ได้จัดเตรียมคำสั่งที่ใช้ในการตรวจสอบเงื่อนไข และมีตัวแปรที่ใช้ในการตรวจสอบการทำงานอย่างสมบูรณ์ ทั้งยังสามารถรองรับการทำงานในลักษณะที่ต้องมีการตัดสินใจพร้อมๆ กันหลายเงื่อนไขด้วย พร้อมกันนั้นยังมีการคำนึงถึงการลดสัญญาณรบกวนและความเพี้ยนของสัญญาณที่เกิดขึ้นจากการประมวลผลคำสั่งและตัวคำสั่งเองมีรูปแบบการทำงานที่ตรงไปตรงมา ชัดเจน ส่งผลให้ผู้พัฒนาโปรแกรมสามารถลดเวลาในการเขียนโปรแกรมลงได้อย่างมาก

- แต่ละค็อกจะประกอบด้วยตัวประมวลผลหรือโปรเซสเซอร์ที่มีการทำงานเป็นอิสระมีแรม 2 กิโลไบต์ ที่เมื่อกำหนดให้ทำงานเป็นรีจิสเตอร์ 32 บิต จะได้ทั้งสิ้นถึง 512 ตัว มีโมดูลตัวนับความสามารถสูงที่ทำงานร่วมกับเฟสล็อกลูป ทำให้แต่ละค็อกทำงานได้เร็วถึง 80 MHz มีวงจรกำเนิดสัญญาณภาพและส่วนควบคุมพอร์ตอินพุตเอาต์พุตที่เป็นอิสระ

- สัญญาณนาฬิกาของระบบมาได้จาก 3 แหล่งคือ วงจรออสซิลเลเตอร์ RC ภายในเลือกได้ระหว่าง 12 หรือ 20 MHz, จากแหล่งกำเนิดสัญญาณนาฬิกาภายนอก และจากการทวีคูณความถี่ของคริสตอลด้วยวงจรเฟสล็อกลูป โดยปกติแล้วจะเลือใช้คริสตอล 5MHz แล้วเลือก PLLx16 ทำให้ได้สัญญาณนาฬิกาของระบบที่มีความถี่ 80MHz ในขณะที่ส่วนเชื่อมโยงกลางจะทำงานด้วยความถี่ที่ลดลงครึ่งหนึ่งของสัญญาณนาฬิกาหลัก

- มีขาพอร์ตอินพุตเอาต์พุตรวมกัน 32 ขา โดยกำหนดให้ใช้ 2 ขาสำหรับต่อกับหน่วยความจำอีอีพรอมสำหรับเก็บโปรแกรมของผู้ใช้งาน และอีก 2ขาสำหรับการดาวน์โหลดโปรแกรม สามารถขับกระแสซิงก์และซอร์สสูงสุด 40mA ต่อขา

- โพรเพลเลอร์ที่ใช้หน่วยความจำอีอีพรอมภายนอกในการเก็บโปรแกรมของผู้ใช้งานทำให้อายุการใช้งานของตัวชิปจึงไม่ขึ้นกับจำนวนรอบของการลบและโปรแกรมใหม่ของหน่วยความจำโปรแกรม

- การดาวน์โหลดโปรแกรมทำได้ง่ายมากเพียงต่อเข้ากับวงจรเชื่อมต่อพอร์ตอนุกรม อาทิ วงจรของไอซี MAX3232 หรือต่อผ่านชิปแปลงสัญญาณพอร์ต USB เป็นพอร์ตอนุกรมอย่าง FT232RL ไม่ต้องการเครื่องโปรแกรมใดๆเพิ่มเติม

- ด้วยความเร็วในการทำงานที่สูง และมีส่วนกำเนิดสัญญาณภาพที่มากถึง 8 ชุด ทำให้เหมาะสมอย่างมากในการนำโพรเพลเลอร์ไปใช้ในการกำเนิดสัญญาณภาพไม่ว่าจะแสดงผลด้วยจอโทรทัศน์ ด้วยสัญญาณวีดีโอ หรือแสดงผลด้วยจอ VGA ด้วยสัญญาณแม่สีแสง นั่นคือพื้นฐานหลักในการสร้างเครื่องเล่นวีดีโอเกม และการสร้างระบบนำเสนอ (presentation) ภาพกราฟฟิกด้วยไมโครคอนโทรลเลอร์เพียวตัวเดียว

- สามารถเชื่อมต่อกับคีย์บอร์ดและเมาส์ได้ และเมื่อรวมกับความสามารถการสร้างสัญญาณภาพได้ จึงสามารถนำโพรเพลเลอร์ไปสร้างเครื่องคอมพิวเตอร์ขนาดเล็กแบบใช้จอโทรทัศน์เป็นตัวแสดงผลได้

- ใช้ไฟเลี้ยงในย่าน 2.7 ถึง 3.6V กระแสไฟฟ้าสูงสุดเมื่อขับโหลดเต็มที่คือ 300mA

- มีตัวถังให้เลือกใช้ 3 แบบคือ DIP 40 ขา, LQFP 44 ขา และ QFN 44 ขา

- มีซอฟต์แวร์ Propeller IDE ที่มีประสิทธิภาพสูง สามารถสร้างระบบไฟล์ที่มีรูปการต่อวงจรได้สะดวก และช่วยให้การเรียนรู้ทำได้อย่างสมบูรณ์ครบวงจร และที่สำคัญยังสามารถดาวน์โหลดซอฟต์แวร์ได้ฟรี จาก www.parallax.com

2.2.2 คุณสมบัติทางเทคนิคของโพรเพลเลอร์

- เป็นไมโครคอนโทรลเลอร์ที่ภายในประกอบไปด้วยโปรเซสเซอร์ขนาด 32 บิตถึง 8 ชุด

- ทำงานที่แรงดัน 3.3 โวลต์ (2.7V ถึง 3.6V)

- ขาพอร์ตสามารถจ่ายกระแสซิงก์และซอร์สได้ 40mA ต่อขา และ100mA ต่อพอร์ต (8ขา)

- มีวงจรกำเนิดสัญญาณนาฬิกาภายใน 12MHz หรือ 20 kHz เลือกกำหนดค่าได้

- ทำงานด้วยสัญญาณนาฬิกาจากภายนอกความถี่ตั้งแต่ ดีซี ถึง 80MHz

- สามารถใช้คริสตอล 4MHz ถึง 8MHz ร่วมกับตัวคูณสัญญาณนาฬิกาความถี่สูงสุด 80 MHz

- สามารถเชื่อมต่อคริสตอลภายนอกได้ โดยไม่จำเป็นต้องเชื่อมต่อตัวเก็บประจุ

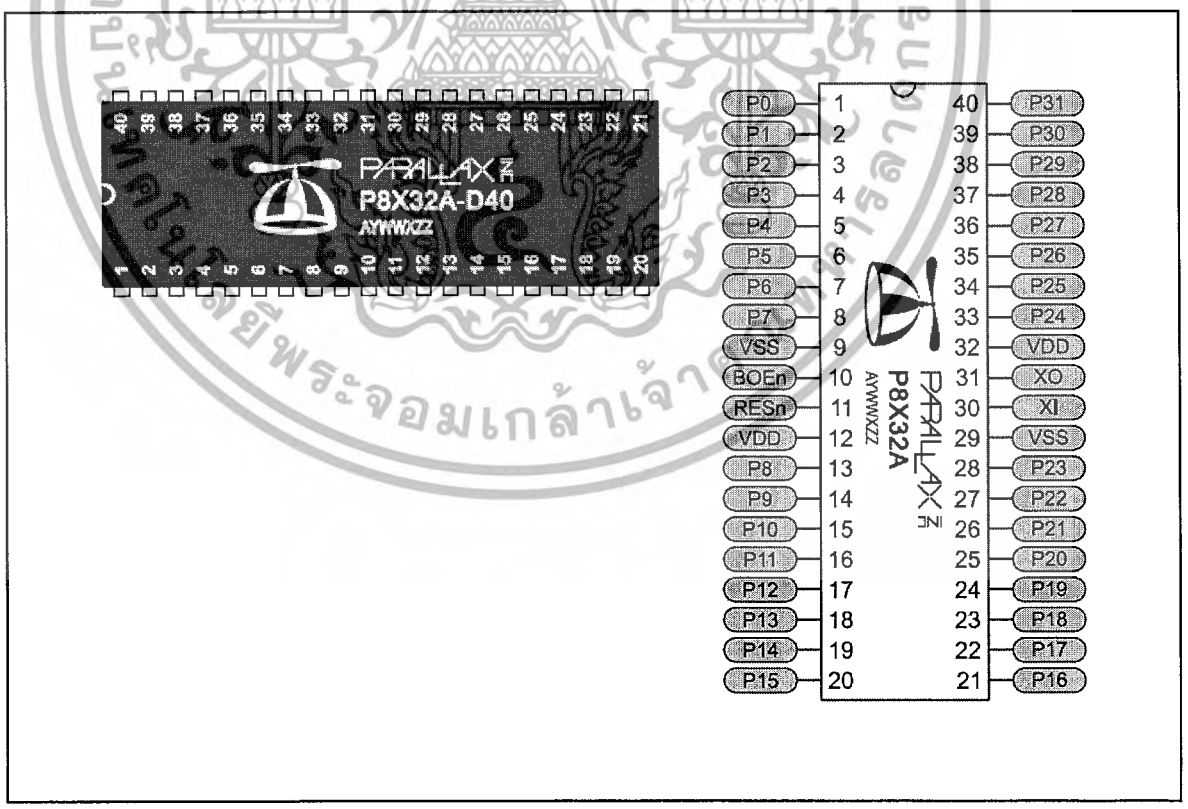
- หน่วยความจำของทั้งระบบ แบ่งเป็นหน่วยความจำอีอีพรอม 32 กิโลไบต์ (KB) และหน่วยความจำแรม 32KB

- ในแต่ละโปรเซสเซอร์มีหน่วยความจำแรม ตัวละ 2KB

- การจัดการหน่วยความจำเป็นแบบ 32บิต

- จำนวนพอร์ตอินพุตเอาท์พุต 32ขา

รูปร่าง และตำแหน่งขาของโพรเพลเลอร์แสดงในรูปที่ 2.1สำหรับตารางที่ 2.1แสดงรายละเอียดหน้าที่การทำงานของขาต่างๆ ของโพรเพลเลอร์ เบอร์ P8X32A



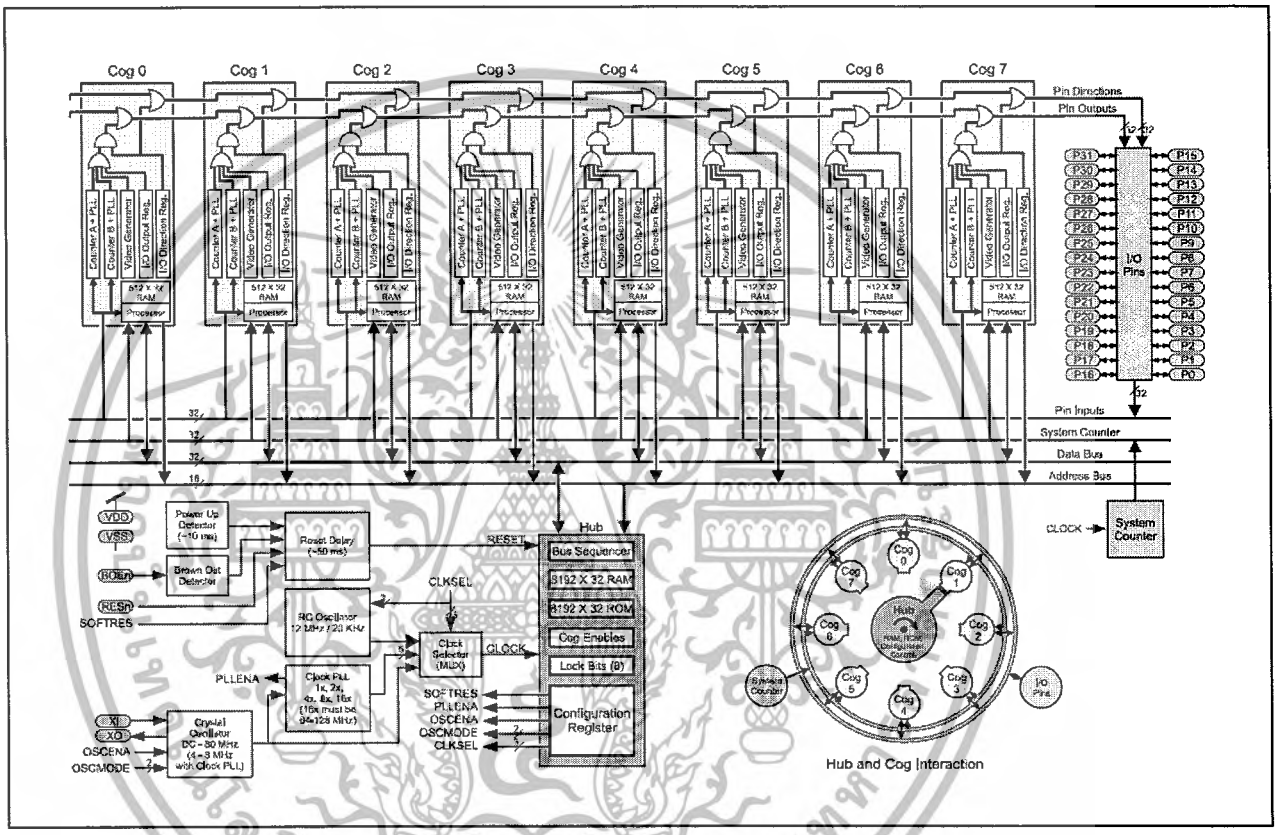
รูปที่ 2.1 แสดงตำแหน่งขา และรูปร่างของไมโครคอนโทรลเลอร์โพรเพลเลอร์แบบตัวถัง Dip 40ขา

ตารางที่ 2.1 แสดงรายละเอียดหน้าที่การทำงานของขาต่างๆ ของโพรเพลเลอร์

ตำแหน่งขา	หน้าที่
P0-P31	ขาพอร์ตอินพุตเอาท์พุทอเนกประสงค์ พอร์ต A สามารถจ่ายกระแส ซอร์ส/ซิงก์ ได้ 40mA ที่แรงดัน 3.3 VDC ระดับลอจิกมีจุดตัดที่ ½ VDD หรือ 1.6 VDC ที่แรงดันไฟเลี้ยง 3.3 V ขาพอร์ต บางขายังมีหน้าที่พิเศษ เมื่อจ่ายไฟครั้งแรกหรือรีเซ็ต (แต่สามารถสั่งให้ขาอินพุตเอาท์พุตได้ผ่านซอฟต์แวร์) P28 เป็นขา SCL ของ I2C สำหรับการเชื่อมต่ออีอีพรอมภายนอก P29 เป็นขา SDA ของ I2C สำหรับการเชื่อมต่ออีอีพรอมภายนอก P30 เป็นขา Tx ส่งข้อมูล สำหรับการสื่อสารอนุกรมกับคอมพิวเตอร์และดาวน์โหลดโปรแกรม P31 เป็นขา Rx ส่งข้อมูล สำหรับการสื่อสารอนุกรมกับคอมพิวเตอร์และดาวน์โหลดโปรแกรม
VDD	ขาไฟบวก 3.3V (2.7 – 3.6 VDC)
VSS	ขากราวด์
$\overline{\text{BOE}}$	ขาเอ็นเอเบิล Brown out (รีเซตเมื่อแรงดันต่ำกว่าที่กำหนด) ทำงานที่ลอจิก “0” ถ้าขานี้เป็น“0” ขารีเซตจะทำหน้าที่เป็นขาเอาท์พุต เพื่อแสดงสถานะ แต่ยังสามารถส่งลอจิก “0” ให้ขารีเซตเพื่อรีเซตไมโครคอนโทรลเลอร์ได้ ถ้าให้ขานี้เป็น “1” ขานี้จะทำหน้าที่เป็นขาอินพุทแบบชมิตทริกเกอร์
$\overline{\text{RES}}$	ขารีเซต ทำงานที่ลอจิก “0” เมื่อขานี้เป็น “0” Propeller จะถูกรีเซต Cog ทั้งหมดจะถูกดิสเอเบิล ขาพอร์ตอินพุตเอาท์พุตจะอยู่ในสถานะลอย Propeller จะรีเซตระยะเวลา 50mS เมื่อขารีเซตเปลี่ยนสถานะจาก “0” เป็น “1”
XI	ขาอินพุตของคริสตอล ใช้ต่อกับแหล่งกำเนิดสัญญาณนาฬิกาจากภายนอก (ขาXO ไม่ใช้งาน) หรือต่อกับขาด้านหนึ่งของคริสตอลหรือเรโวเนเตอร์ (ขาอีกข้างต่อกับ XO) โดยไม่จำเป็นต้องต่อตัวต้านทานหรือตัวเก็บประจุภายนอก
XO	ขาเอาท์พุตของคริสตอล ออกแบบมาเป็นขาป้อนกลับของคริสตอล การต่อคริสตอลเข้ากับขา XI และ XO จะต้องสัมพันธ์กับค่าที่กำหนดให้กับรีจิสเตอร์ CLK ด้วย

2.2.3 หลักการทำงานของโพรเพลเลอร์

จากรูปที่ 2.2 แสดงให้เห็นบล็อกไดอะแกรมการทำงานภายในของโพรเพลเลอร์ ซึ่งประกอบด้วยโปรเซสเซอร์ที่ทำงานแยกกันอิสระถึง 8 ชุด โดยจะเรียกโปรเซสเซอร์เหล่านี้ว่า Cog หมายเลข 0 ถึง 7



รูปที่ 2.2 บล็อกไดอะแกรมแสดงการทำงานภายในของโปรเซสเซอร์ทั้ง 8 ตัวของโพรเพลเลอร์

2.2.3.1 ค็อก (Cogs)

ในแต่ละค็อกจะประกอบไปด้วย หน่วยความจำแรม 2KB โดยกำหนดเป็นหน่วยความจำแบบ 32 บิต จำนวน 512 ตัว นอกจากนี้ภายในโปรเซสเซอร์แต่ละตัวยังมีโมดูลเคาน์เตอร์แบบพิเศษพร้อมเฟสล็อกลูป 2 ตัว โมดูลสร้างสัญญาณวิดีโอ รีจิสเตอร์พอร์ตอินพุตเอาท์พุต รีจิสเตอร์กำหนดทิศทางของพอร์ตอินพุตเอาท์พุต และรีจิสเตอร์ตัวอื่นๆ ซึ่งไม่ได้แสดงให้เห็นในบล็อกไดอะแกรม

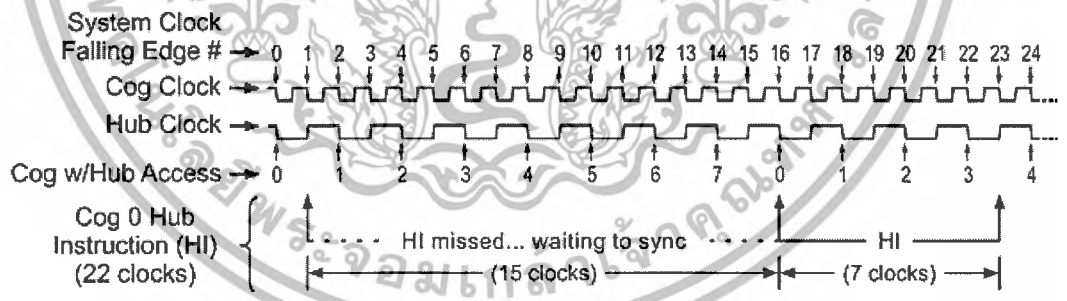
ค็อกทั้ง 8 ตัวทำงานด้วยวงจรกำเนิดสัญญาณนาฬิกาหลัก ซึ่งค็อกแต่ละตัวจะอ้างอิงการทำงานกันได้ด้วยสัญญาณนาฬิกา และจะเริ่มต้นทำงานพร้อมกันและใช้ทรัพยากรด้วยกัน ค็อกแต่ละตัวสามารถสั่งให้ทำงานหรือหยุดทำงานได้ในขั้นตอนการทำงานของโปรแกรม และสามารถ

ควบคุมให้แต่ละค็อกทำงานไปพร้อมๆกันได้ โดยจะทำงานเป็นอิสระหรือ เชื่อมโยงถึงกันได้ผ่านหน่วยความจำแรมหลัก (Main RAM) ซึ่งแยกไปต่างหาก

หน่วยความจำภายในค็อกแต่ละตัว เรียกว่า ค็อกแรม (Cog RAM) โดยค็อกแรมจะแบ่งหน่วยความจำเป็นรีจิสเตอร์ขนาด 32 บิต จำนวน 512 ตัว สามารถใช้งานได้อย่างอิสระ ยกเว้นรีจิสเตอร์ 16 ตำแหน่งสุดท้ายซึ่งสงวนไว้สำหรับรีจิสเตอร์ฟังก์ชั่นพิเศษ เช่น รีจิสเตอร์เคาน์เตอร์, รีจิสเตอร์พอร์ตอินพุตเอาท์พุต เป็นต้น

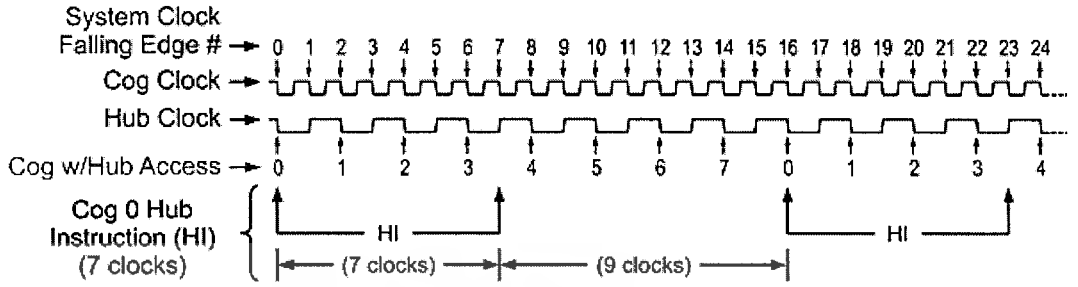
2.2.3.2 ฮับ (Hub): ส่วนเชื่อมโยงหลัก

ฮับทำหน้าที่จัดระเบียบการทำงานของระบบทั้งหมด โดยจะยอมให้ค็อกทีละตัวเท่านั้นที่จะติดต่อกับทรัพยากรหลักของระบบ โดยจะหมุนเวียนติดต่อกับค็อกตั้งแต่หมายเลข 0 ถึง 7 แล้วกลับไปที่หมายเลข 0 ใหม่เป็นลักษณะวนรอบ ส่วนของฮับและระบบบัสของมันทำงานด้วยความเร็วครึ่งหนึ่งของสัญญาณนาฬิกาของทั้งระบบ ทำให้ค็อก 1 ตัวจะถูกติดต่อทุกๆ 16 ไซเกิลของสัญญาณนาฬิกา และใช้เวลา 7 ไซเกิลเพื่อเอ็กซิคิวคำสั่ง ดังนั้น ฮับจะติดต่อกับค็อกตัวใดตัวหนึ่งได้อาจใช้คาบเวลาเพียง 7 ไซเกิล หรือนานถึง 22 ไซเกิล เนื่องจากจะต้องรอให้ฮับวนมาจนครบรอบ



Cog-Hub Interaction – Worst Case Scenario

รูปที่ 2.3 การทำงานของฮับติดต่อกับค็อกที่ผิดพลาด



Cog-Hub Interaction – Best Case Scenario

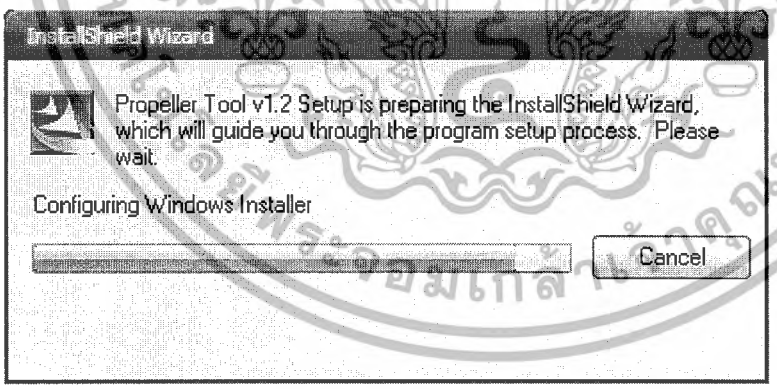
รูปที่ 2.4 การทำงานของฮับติดต่อกับค็อกที่ถูกต้อง

2.3การติดตั้งโปรแกรม และการใช้งานโปรแกรม

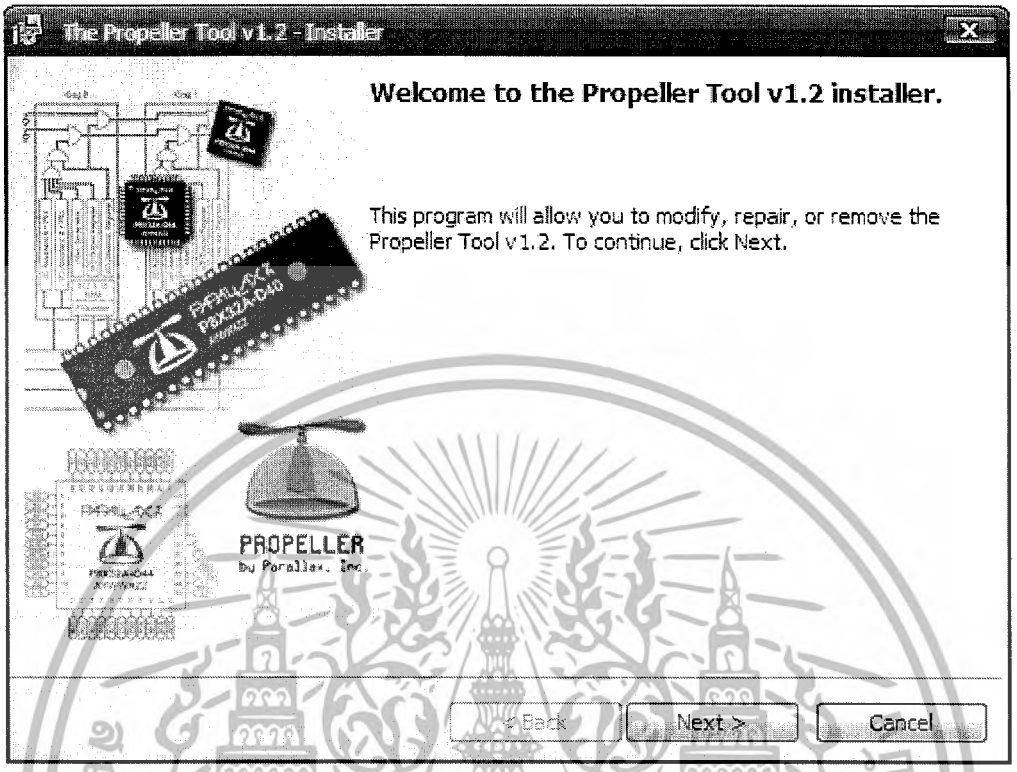
2.3.1 การติดตั้งซอฟต์แวร์ Propeller Tools

1.ดับเบิ้ลคลิกที่ Setup-Propeller-Tool-v1.2

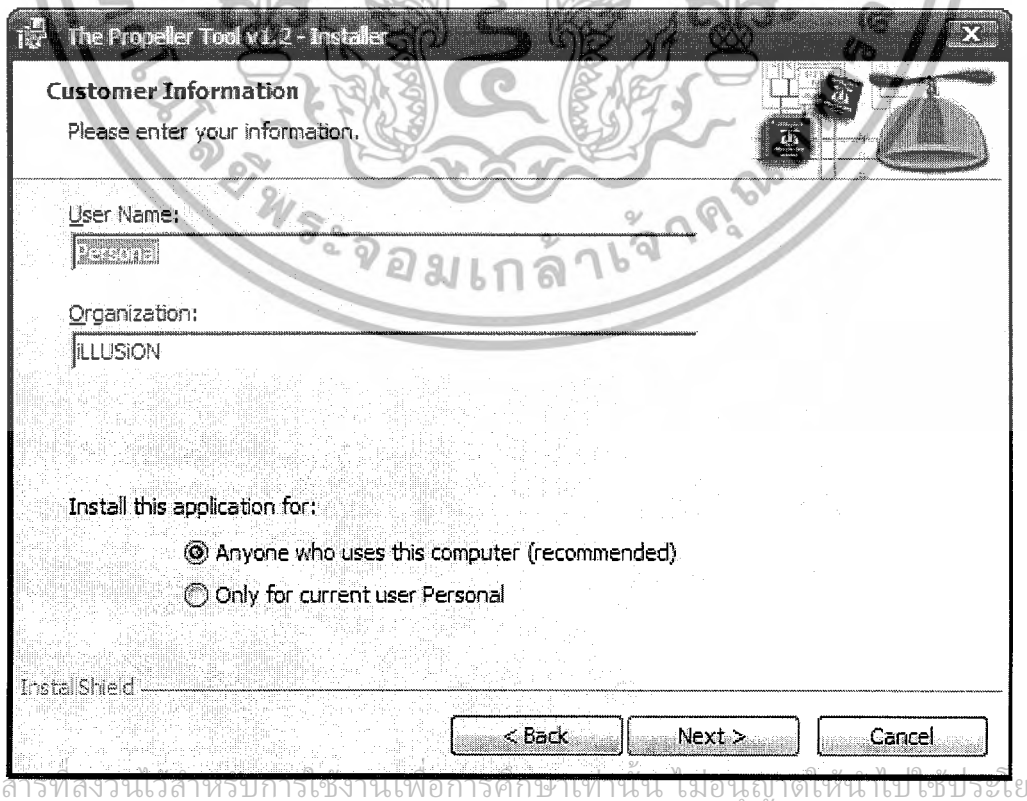
2.เครื่องจะทำการ Install Shield Wizard



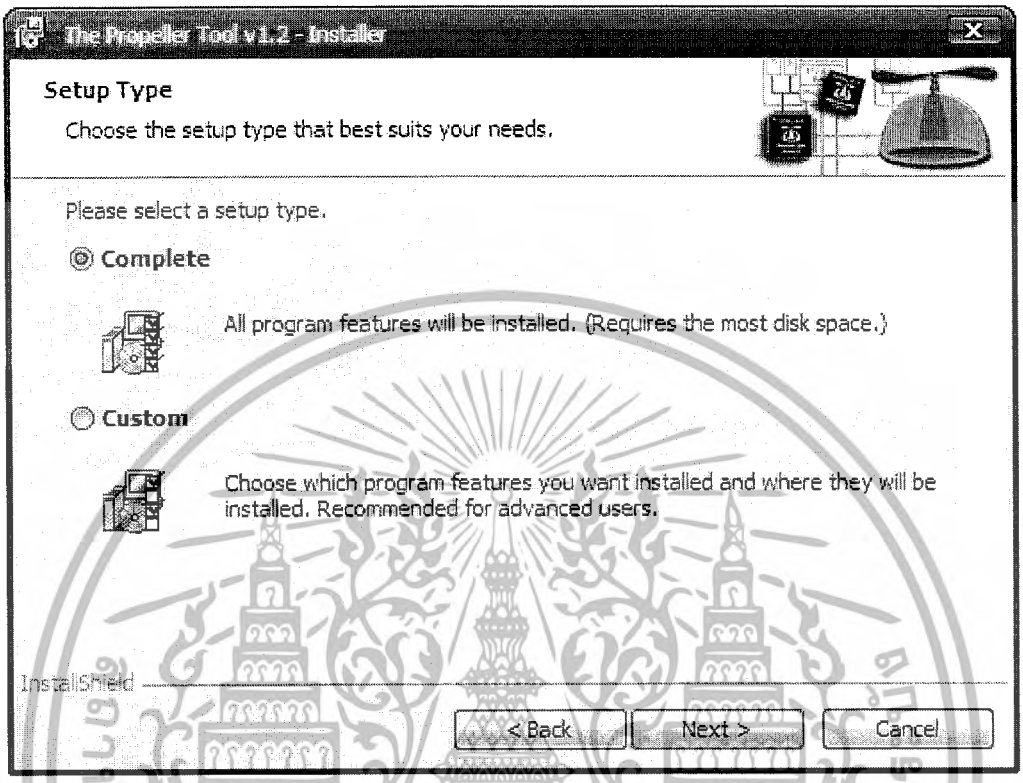
3.หลังการ Install Shield Wizard จะปรากฏหน้าต่าง The Propeller Tool v 1.2 – Installer ให้คลิก Next>



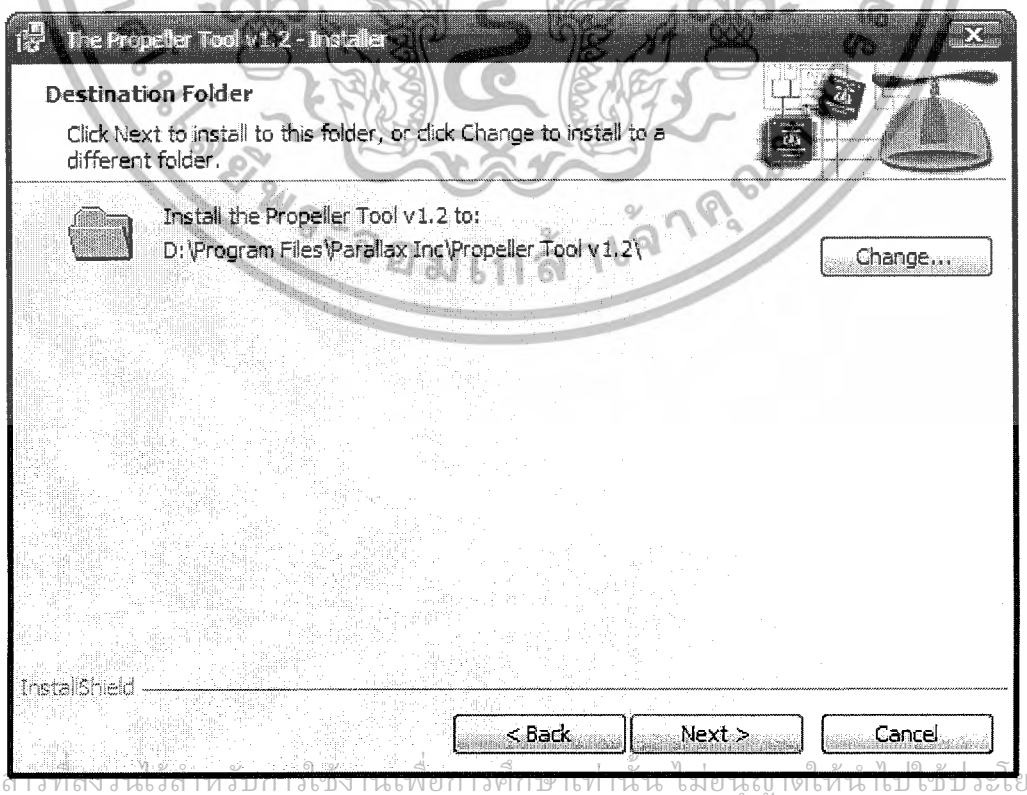
4.หน้าต่าง Customer Information ให้ตั้งชื่อ User Name และ Organization ที่ Install this application for ให้เลือก ◉ Anyone who uses this computer (recommended) แล้วคลิก Next>



5.ที่หน้าต่าง Setup Type เลือก Complete แล้วคลิก Next>

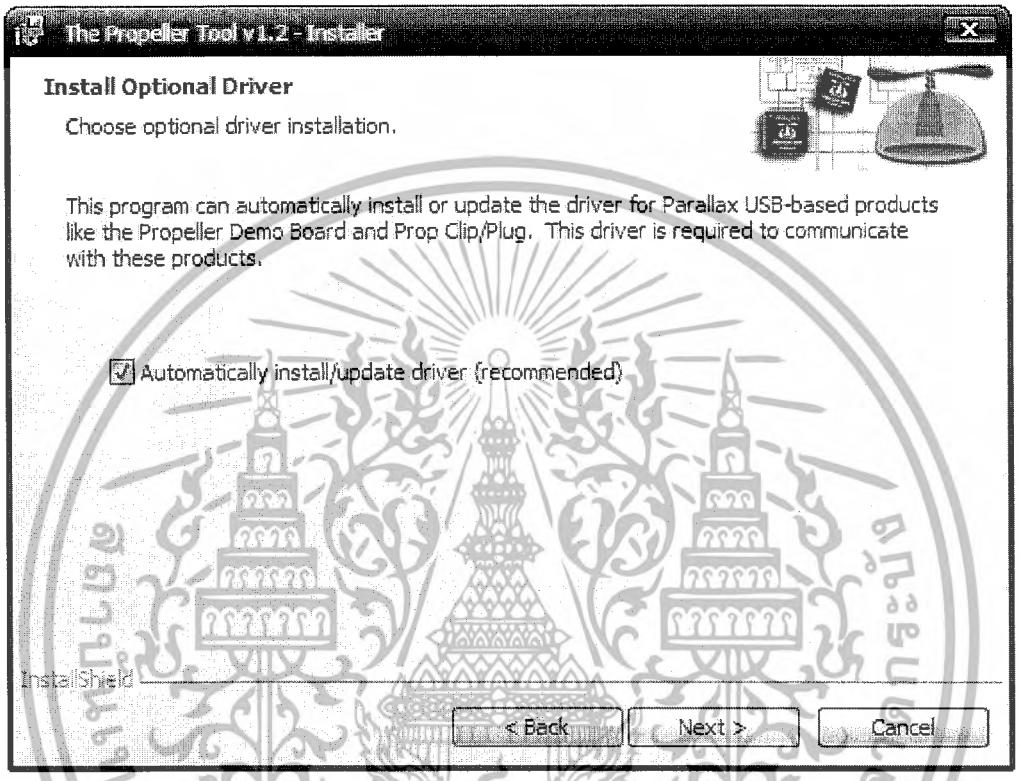


6.ที่หน้าต่าง Destination Folder ให้ทำการเลือกตำแหน่งที่จะจัดเก็บโปรแกรม ให้คลิก Change เพื่อทำการเลือกตำแหน่ง เมื่อเรียบร้อยแล้ว คลิก Next>

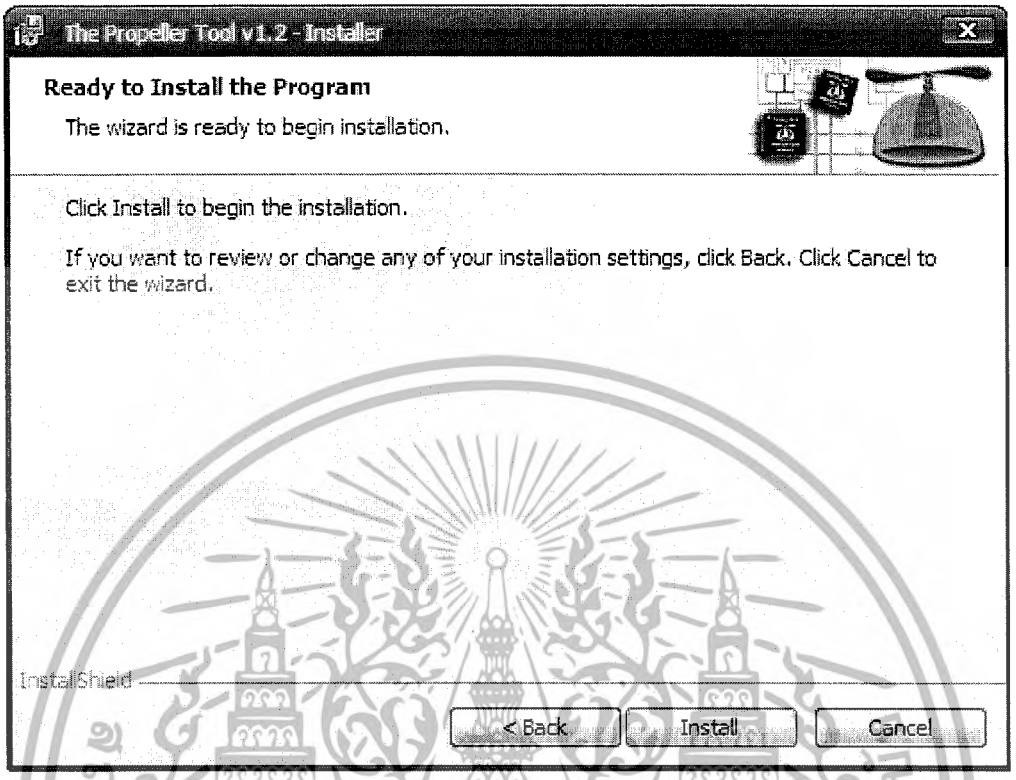


7.ที่หน้าต่าง Install Optional Driver ให้คลิกถูกที่ Automatically install/update driver (recommended)

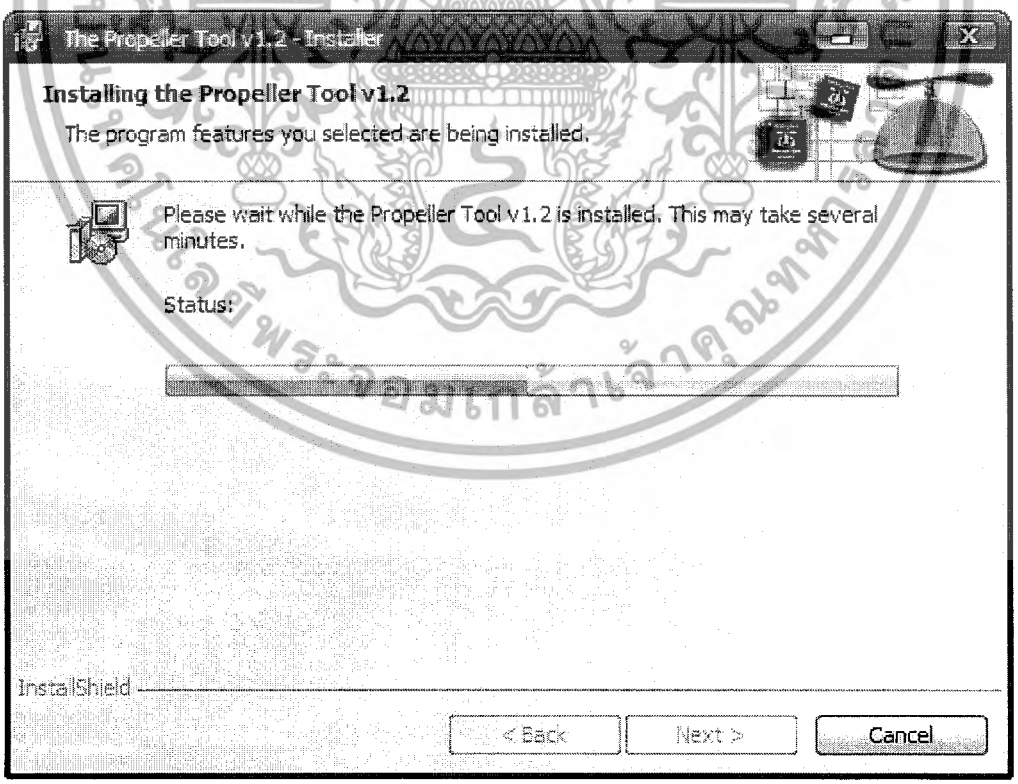
แล้วคลิก Next>



8.ที่หน้าต่าง Ready to Install the Program ให้คลิก Next>



9.เริ่มทำการติดตั้งโปรแกรม จะปรากฏหน้าต่าง Installing the Propeller Tool v1.2



10.เมื่อโปรแกรมทำการติดตั้งเสร็จเรียบร้อย จะปรากฏหน้าต่าง Propeller Tool 1.2 Installation Completed

ให้คลิก Finish

2.3.2 การใช้งานโปรแกรมPropeller Tool

โปรแกรมPropeller Tool เป็นโปรแกรมที่ใช้สำหรับชิปของโพลเพลเลอร์ ซึ่งมีข้อที่ควรรู้ดังนี้

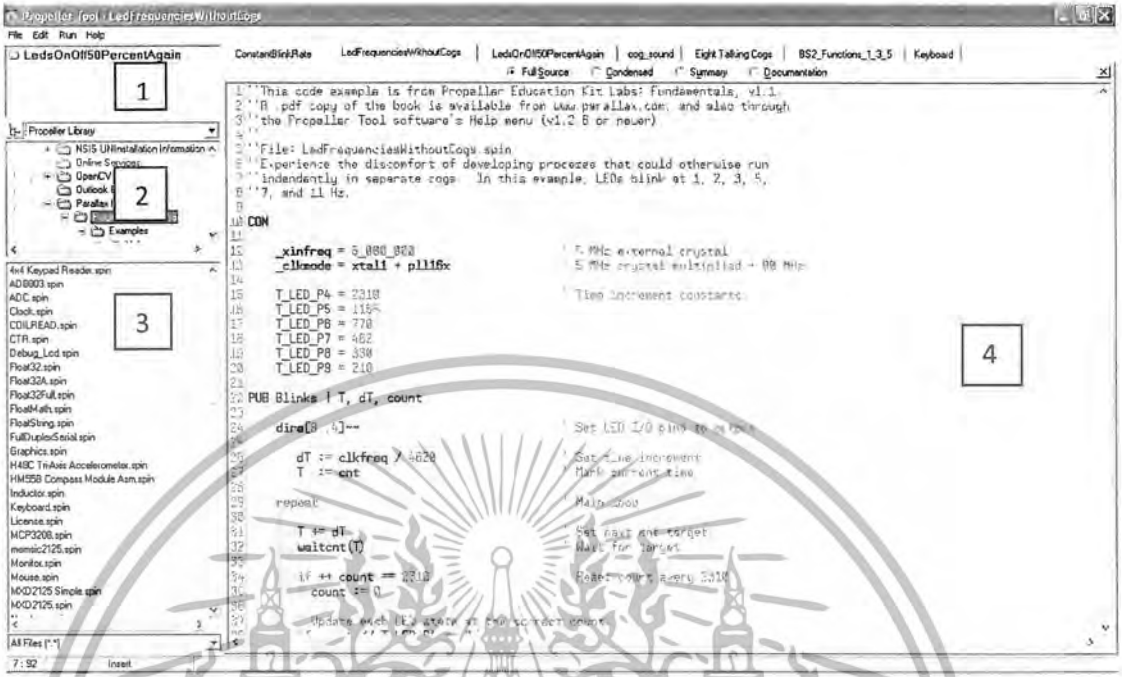
หน้าต่างหลักของโปรแกรมPropeller Tool ประกอบด้วย 4ส่วนหลักๆคือ

1. The object view pane แสดงส่วนของOBJECTที่มีอยู่ในโปรแกรมที่เรากำลังใช้งานอยู่ เนื่องจากว่าภาษาสปินสามารถสร้างOBJECTได้หลายๆอันในหนึ่งโปรแกรม ซึ่งเป็นประโยชน์คือเราเปิดเพียงโปรแกรมหลัก และหากเราต้องการแก้ไขส่วนOBJECT (ส่วนย่อยของโปรแกรม) ก็เพียงแค่คลิ๊กเลือกOBJECTที่ต้องการได้จากส่วนนี้ได้

2. The recent folders field and the folder listแสดงโฟลเดอร์ลิส เพื่อง่ายต่อการเรียกงานจากโฟลเดอร์ที่เราเก็บไว้ออกมาใช้

3. The file list and the filter field แสดงไฟล์ลิส ที่อยู่ในโฟลเดอร์ที่เลือกจากโฟลเดอร์ลิส

4. Editor pane สำหรับเขียนโปรแกรม



รูปที่ 2.5 หน้าต่างหลักของโปรแกรมPropeller Tool

2.3.3 ภาษาสปิน

ในการเขียนโปรแกรมเพื่อพัฒนาโพรเพลเลอร์นั้น จะใช้ภาษาสปิน ซึ่งมีลักษณะการเขียนโปรแกรมเป็นแบบออบเจ็กต์ ประกอบด้วยบล๊อกต่างๆ คือ CON, VAR, OBJ, PUB, PRI, และ DAT โดยแต่ละบล๊อกมีรายละเอียดดังนี้

1. CON

เป็นที่สำหรับเก็บค่าคงที่แบบครอบคลุมทั้งโปรแกรม ซึ่งมีค่าคงที่ที่กำหนดเป็นมาตรฐานคือ

1.1 _CLKFREQ

ใช้กำหนดความถี่สัญญาณนาฬิกา โดยกำหนดเพียงครั้งเดียวที่ส่วนหัวของโปรแกรม

รูปแบบการใช้งานคำสั่ง

CON

_CLKFREQ = Expression

พารามิเตอร์

Expression หมายถึง ค่าตัวเลขจำนวนเต็มเพื่อกำหนดความถี่สัญญาณนาฬิกา

ตัวอย่างที่ 1

```
CON
_CLKMODE = XTAL1 + PLL8X
_CLKFREQ = 32_000_000
```

โปรแกรมในบรรทัดแรกเป็นการกำหนดค่าสัญญาณนาฬิกาจากภายนอกให้เป็นแบบคริสตอลความถี่ต่ำและความถี่สัญญาณนาฬิกาคูณด้วย 8 ส่วนบรรทัดที่ 2 เป็นการกำหนดความถี่ออสซิลเลเตอร์เท่ากับ 32MHz ดังนั้นหมายความว่า คริสตอลต้องมีความถี่เท่ากับ 4MHz จาก 4MHz x 8 เท่ากับ 32MHz ค่าของ _XINFREQ จะถูกกำหนดให้มีค่าเท่ากับ 4MHz โดยอัตโนมัติ

ตัวอย่างที่ 2

```
_CLKMODE = XTAL2
_CLKFREQ = 10_000_000
```

บรรทัดแรกกำหนดค่าสัญญาณนาฬิกาจากภายนอกเป็นคริสตอลความถี่กลางโดย ไม่มีการเพิ่มความถี่จากตัวคูณ บรรทัดที่ 2 กำหนดความถี่สัญญาณนาฬิกาที่ 10MHz ดังนั้นค่าของ _XINFREQ จะถูกกำหนดให้มีค่าเท่ากับ 10MHz โดยอัตโนมัติ

ตารางที่2.2 ตารางกำหนดโหมดออสซิลเลเตอร์

ตารางกำหนดโหมดของความถี่ออสซิลเลเตอร์	
RCFAST	ออสซิลเลเตอร์ภายในความถี่สูง ประมาณ 12MHz(เป็นค่าเริ่มต้น)
RCSLOW	ออสซิลเลเตอร์ภายในความถี่ต่ำ ประมาณ 20MHz
XINPUT	สัญญาณนาฬิกาจากภายนอกต่อเข้าขา XI ช่วง DC ถึง 80MHz
XTAL1	ต่อคริสตอลภายนอกความถี่ต่ำ (4MHz ถึง 16MHz)
XTAL2	ต่อคริสตอลภายนอกความถี่ปานกลาง (8MHz ถึง 32MHz)
XTAL3	ต่อคริสตอลภายนอกความถี่สูง (20MHz ถึง 80MHz)
PLL1X	คูณค่าความถี่ออสซิลเลเตอร์ 1 เท่า
PLL2X	คูณค่าความถี่ออสซิลเลเตอร์ 2 เท่า

PLL4X	คูณค่าความถี่ออสซิลเลเตอร์ 4 เท่า
PLL8X	คูณค่าความถี่ออสซิลเลเตอร์ 8 เท่า
PLL16X	คูณค่าความถี่ออสซิลเลเตอร์ 16 เท่า

1.2_CLKMODE

ใช้กำหนดโหมดของออสซิลเลเตอร์ที่ใช้

รูปแบบการใช้งานคำสั่ง

CON

_CLKMODE = Expression

พารามิเตอร์

Expression หมายถึง ค่าตัวเลขจำนวนเต็มเพื่อกำหนดโหมดเพียงโหมดเดียวหรือสองโหมดของสัญญาณนาฬิกา โดยกำหนดที่ส่วนหัวของโปรแกรม ดังแสดงในตารางที่ 3.1 แต่ในการกำหนดสองโหมดไม่ใช่ทุกโหมดที่สามารถกำหนดค่าได้ โดยในตารางที่ 3.2 แสดงโหมดต่างๆ ที่สามารถกำหนดค่าได้

ตัวอย่างที่ 3

CON

_CLKMODE = RCFAST

เป็นการกำหนดค่าสัญญาณนาฬิกาจากภายในโหมดความถี่สูง ประมาณ 12MHz ซึ่งเป็นค่าเริ่มต้นเมื่อจ่ายไฟให้ไมโครคอนโทรลเลอร์ครั้งแรก สำหรับในโหมดนี้ไม่สามารถใช้การคูณความถี่ได้

ตารางที่ 2.3 โหมดของสัญญาณนาฬิกาที่สามารถกำหนดค่าได้

โหมดของสัญญาณนาฬิกาที่สามารถกำหนดค่าได้	
ค่าที่กำหนดได้	ค่าในรีจิสเตอร์ CLK
RCFAST	0_0_0_00_000

RCSLOW	0_0_0_00_001
XINPUT	0_0_0_00_010
XTAL1	0_0_1_01_010
XTAL2	0_0_1_10_010
XTAL3	0_0_1_11_010
XINPUT+PLL1X	0_1_1_00_011
XINPUT+PLL2X	0_1_1_00_100
XINPUT+PLL4X	0_1_1_00_101
XINPUT+PLL8X	0_1_1_00_110
XINPUT+PLL16X	0_1_1_00_111
XTAL1+PLL1X	0_1_1_01_011
XTAL1+PLL2X	0_1_1_01_100
XTAL1+PLL4X	0_1_1_01_101
XTAL1+PLL8X	0_1_1_01_110
XTAL1+PLL16X	0_1_1_01_111
XTAL2+PLL1X	0_1_1_10_011
XTAL2+PLL2X	0_1_1_10_100
XTAL2+PLL4X	0_1_1_10_101
XTAL2+PLL8X	0_1_1_10_110
XTAL2+PLL16X	0_1_1_10_111
XTAL3+PLL1X	0_1_1_11_011
XTAL3+PLL2X	0_1_1_11_100
XTAL3+PLL4X	0_1_1_11_101
XTAL3+PLL8X	0_1_1_11_110
XTAL3+PLL16X	0_1_1_11_111

ตัวอย่างที่ 4

```
CON
        _CLKMODE = XTAL1 + PLL8X
```

กำหนดสัญญาณนาฬิกาจากภายนอกเป็นโหมดคริสตอลความถี่ต่ำ และกำหนดตัวคูณให้กับสัญญาณนาฬิกา 8 เท่า ถ้าคริสตอลความถี่ 4MHz ความถี่สัญญาณนาฬิกาเท่ากับ 4MHz x 8 = 32MHz

การกำหนด_CLKMODE จะต้องใช้ร่วมกับ _XINFREQ เพื่อแสดงค่าความถี่ของคริสตอลที่ต่อภายนอก เพื่อให้ไมโครคอนโทรลเลอร์สามารถปรับค่าคาบเวลาได้อย่างเหมาะสม ดังนี้

```
CON
  _CLKMODE = XTAL1 + PLL8
  _XINFREQ = 4_000_000
```

นอกจากนี้ยังมีคำสั่งอื่นๆ ที่ใช้อยู่ในบล๊อก CON ประกอบไปด้วย

_FREE	ใช้สำหรับกำหนดพื้นที่ว่างของหน่วยความจำ
_STACK	ใช้จองพื้นที่ของสแตก
TRUE	สถานะลอจิกเป็นจริง “1” ($FFFFFFFF)
FALSE	สถานะลอจิกเป็นจริง “0” ($00000000)
POSX	ค่าสูงสุดด้านบวก (32 บิต) 2,147,483.647 ($FFFFFFFF)
NEGX	ค่าสูงสุดด้านลบ (32 บิต) -2,147,483.647 ($00000000)
PI	ค่าตัวเลขทศนิยมของPI ประมาณ 3.141593 ($40490FDB)
RCFAST	สัญญาณนาฬิกาจากภายในความถี่สูง
RCSLOW	สัญญาณนาฬิกาจากภายในความถี่ต่ำ
XINPUT	สัญญาณนาฬิกาจากภายนอก (เข้าที่ขา XI)
XTAL1	สัญญาณนาฬิกาจากภายนอกความถี่ต่ำ
XTAL2	สัญญาณนาฬิกาจากภายนอกความถี่ปานกลาง
XTAL3	สัญญาณนาฬิกาจากภายนอกความถี่สูง

PLL1X คูณค่าสัญญาณนาฬิกา 1 เท่า

PLL2X คูณค่าสัญญาณนาฬิกา 2 เท่า

PLL4X คูณค่าสัญญาณนาฬิกา 4 เท่า

PLL8X คูณค่าสัญญาณนาฬิกา 8 เท่า

PLL16X คูณค่าสัญญาณนาฬิกา 16 เท่า

2. VAR

เป็นการกำหนดค่าตัวแปรแบบครอบคลุมทั้งโปรแกรม โดยชนิดของตัวแปรที่ใช้กับ Spin ประกอบไปด้วย

Byte: กำหนดตัวแปรเป็นตัวเลขจำนวนเต็ม 8 บิต

Word: กำหนดตัวแปรเป็นตัวเลขจำนวนเต็ม 16 บิต

Long: กำหนดตัวแปรเป็นตัวเลขจำนวนเต็ม 32 บิต

Float: กำหนดตัวแปรเป็นตัวเลขทศนิยม

ตัวอย่างที่ 5

```
VAR
    Byte Temp(10)
    Word Code
    Long LargeNumber
```

3. OBJ

เป็นบล็อกในส่วนที่ใช้ประกาศไลบรารี่ที่ต้องการเรียกใช้ภายในเครื่องหมายคำพูด

ตัวอย่างที่ 6

```
OBJ
    Num : "Numbers"
    Term : "TV_Terminal"
```

4. PUB

เป็นบล็อกสำหรับการสร้างส่วนของโปรแกรมย่อย หรือฟังก์ชัน แบบเรียกใช้งานได้จากภายในโปรเจ็กต์หรือเรียกผ่านจากโปรเจ็กต์อื่นๆก็ได้ PUB ตัวแรกที่พบจะถือว่าเป็นส่วนที่เรียกใช้เป็นอันดับแรก ถ้าเทียบกับภาษา C ก็จะหมายถึงส่วนของ main นั่นเอง

5. PRI

เป็นบล็อกสำหรับสร้างส่วนของโปรแกรมย่อย หรือฟังก์ชันแบบเรียกใช้งานเฉพาะภายในตัวโปรเจ็กต์ที่กำลังทำงานอยู่เท่านั้น การทำงานส่วนอื่นๆ จะเหมือนกับการทำงานของบล็อก PUB

6. DAT

เป็นบล็อกสำหรับกำหนดเก็บตารางของข้อมูลหรือเขียนโค้ดโปรแกรมภาษาแอสเซมบลี บล็อกที่กล่าวถึงทั้งหมดนี้ สามารถเลือกประกาศเฉพาะตัวที่ต้องการเรียกใช้เท่านั้น การประกาศสามารถประกาศบล็อกชนิดเดียวกันได้หลายๆ บล็อก โดยแต่ละบล็อกจะคั่นด้วยสีที่เป็นสีอ่อน และสีเข้มโดย

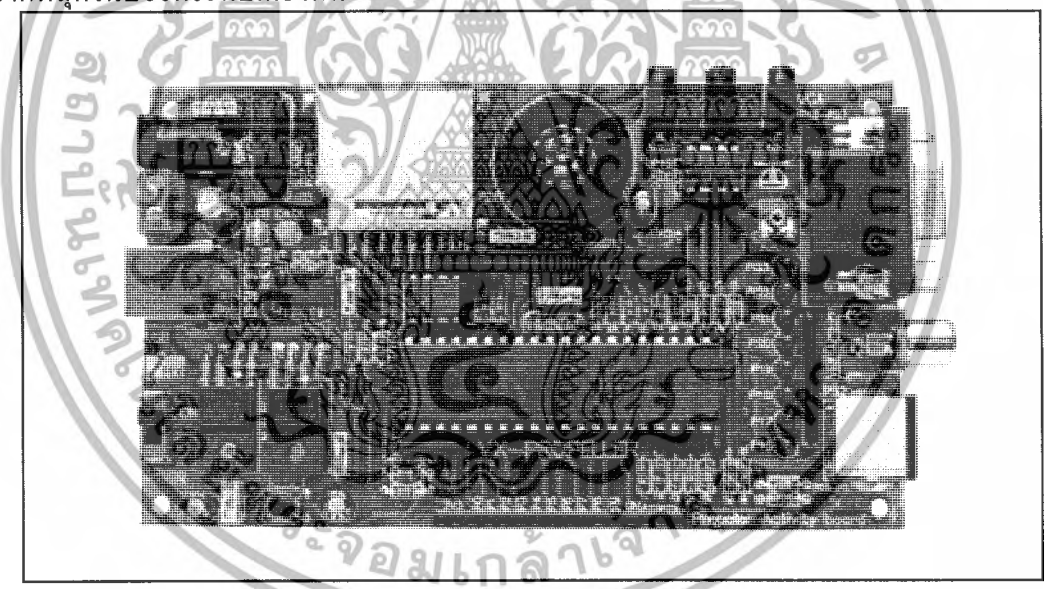
CON	แสดงด้วยสีเหลือง
VAR	แสดงด้วยสีส้ม
OBJ	แสดงด้วยสีชมพู
PUB	แสดงด้วยสีน้ำเงิน
PRI	แสดงด้วยสีฟ้า
DAT	แสดงด้วยสีเขียว

บทที่ 3

การออกแบบ

3.1 บอร์ดที่ใช้ในการทดลอง

บอร์ดที่ใช้ในการทดลอง คือบอร์ดของโพรเพลเลอร์โดยใช้ชิปโพรเพลเลอร์เบอร์ P8X32A-D40 ต่อร่วมกับชิป FT232RL เพื่อรองรับการดาวน์โหลดโปรแกรมผ่านพอร์ต USB มีการต่อชิปโพรเพลเลอร์เข้ากับอุปกรณ์อินพุตเอาท์พุต ตั้งแต่ระดับพื้นฐานอย่าง LED และลำโพง จนถึงขั้นก้าวหน้าอย่าง SD การ์ด, จอภาพ VGA, จอโทรทัศน์ผ่านช่อง AV, คีย์บอร์ดหรือเมาส์แบบ PS2 และวงจรแปลงสัญญาณอะนาลอกเป็นดิจิตอลแบบอนุกรม นอกจากนี้ยังมีการจัดสรรขาพอร์ตอิสระมากถึง 14 ขาให้สามารถนำไปใช้เชื่อมต่อกับอุปกรณ์ภายนอกได้อย่างมากเพียงพอ ซึ่งนับว่ามากที่สุดในบอร์ดระดับเดียวกัน



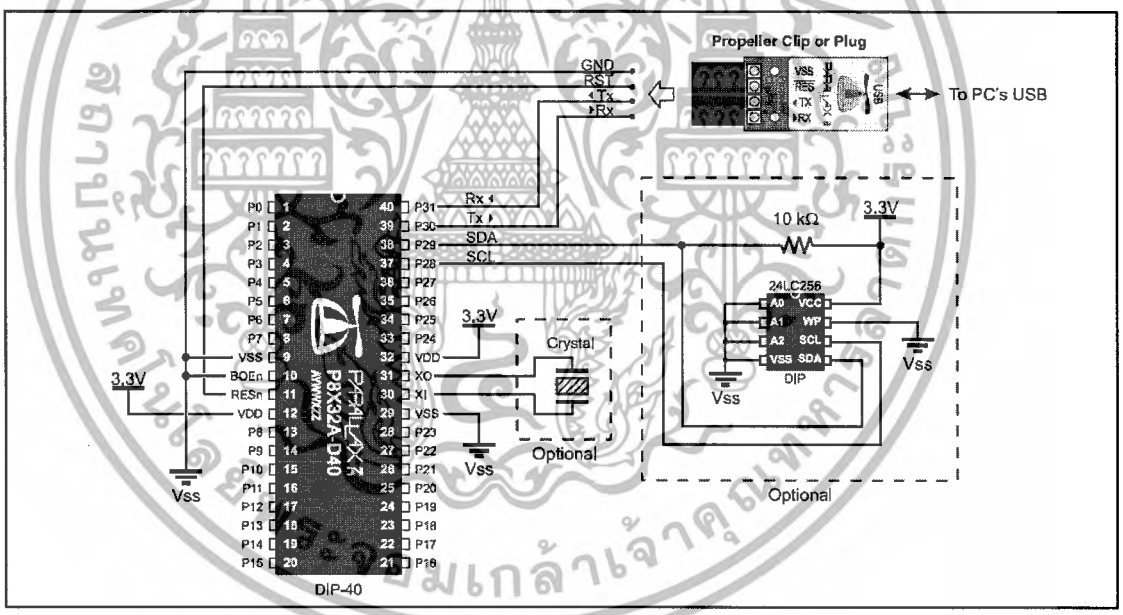
รูปที่ 3.1 บอร์ด VX-Propeller

การทำงานของวงจรบอร์ดVX-Propeller

ภาคจ่ายไฟจะรับแรงดันอินพุทจากแหล่งจ่ายไฟตรงสองแหล่งด้วยกันคือ จากจุดต่อแจ็กอะแดปเตอร์และจุดต่อเทอร์มินอลบล็อก สำหรับจุดต่อแจ็กอะแดปเตอร์จะมีไดโอดบริดจ์ต่อไว้เพื่อป้องกันการกลับขั้วของวงจร ก่อนส่งให้กับไอซี 78R05 เพื่อเรกูเลตแรงดันให้เหลือ 5 โวลต์ใช้เลี้ยงวงจรในส่วนแรงดัน 5 โวลต์เช่นจุดต่อ VGA และจุดต่อคีย์บอร์ด จากนั้นส่งเข้าไปยังเรกูเลเตอร์

เบอร์ 78R33 เพื่อเรกูเลตให้เหลือ 3.3 โวลต์ เป็นไฟเลี้ยงให้กับไมโครคอนโทรลเลอร์และอุปกรณ์ต่อพ่วงอื่นๆ

ภาคสื่อสารข้อมูลกับคอมพิวเตอร์ และดาวน์โหลดโปรแกรมเชื่อมต่อกับคอมพิวเตอร์ผ่านพอร์ต USB โดยมีไอซี FT232RL ทำหน้าที่แปลงรูปแบบการสื่อสารข้อมูลจาก USB ให้เป็นการสื่อสารแบบอนุกรม ซึ่ง Propeller ต้องการสายสัญญาณในการเชื่อมต่อ 3 สาย ประกอบด้วย Tx, Rx และ DTR โดยขาสัญญาณ DTR จะต้องต่อเข้ากับขา Reset ของ Propeller เพื่อรีเซตตัวมันและเข้าสู่โหมดโปรแกรม แต่เนื่องจากจะต้องป้อนสัญญาณรีเซตด้วยลอจิก"0" จึงจำเป็นต้องต่อทรานซิสเตอร์ ไว้เพื่อกลับสถานะลอจิก และจะรีเซตเมื่อขา DTR มีลอจิก "1" ไอซี FT232RL เมื่อติดต่อกับคอมพิวเตอร์เป็นที่เรียบร้อยจะแสดงสถานะที่ LED สีน้ำเงินจะติดสว่าง และเมื่อมีการส่งข้อมูลLEDสีเหลืองจะติดสว่าง



รูปที่ 3.2 วงจงที่ใช้โหลดโปรแกรมจากคอมพิวเตอร์ไปยังชิปโพรเพลเลอร์ผ่านทางUSB

ส่วนหัวใจหลักของวงจรเป็นชิปโพรเพลเลอร์ทำงานด้วยคริสตอลภายนอก 5 MHz ซึ่งสามารถใช้วงจรคูณสัญญาณภายในให้ได้ความถี่สูงถึง 80 MHz ตัวมันจะมีขาอินพุตเอาท์พุตทั้งหมด 32 ขา ถูกใช้งานเพื่อสื่อสารข้อมูลกับคอมพิวเตอร์ 2 ขา คือขา Rx และ Tx ใช้เชื่อมต่อกับหน่วยความจำอีอีพรอมเพื่อเก็บข้อมูลโปรแกรมอีก 2 ขา นอกนั้นใช้งานได้อเนกประสงค์

บอร์ด VX-Propeller ได้จัดการเชื่อมต่อขาพอร์ตกับอุปกรณ์ภายนอกไว้ดังนี้

P0 ถึง P3 ต่อกับซ็อกเก็ต SD การ์ด

P10 ต่อกับไอซีแปลงสัญญาณอะนาลอกเป็นดิจิตอลเบอร์ QP410

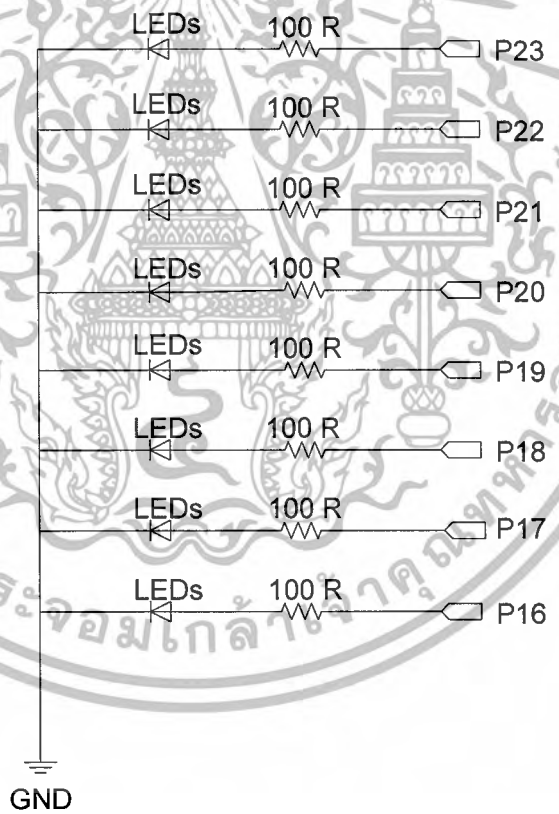
P11 ต่อกับลำโพง

P12 ถึง P15 ต่อกับวงจรจัดสัญญาณวีดีโอ

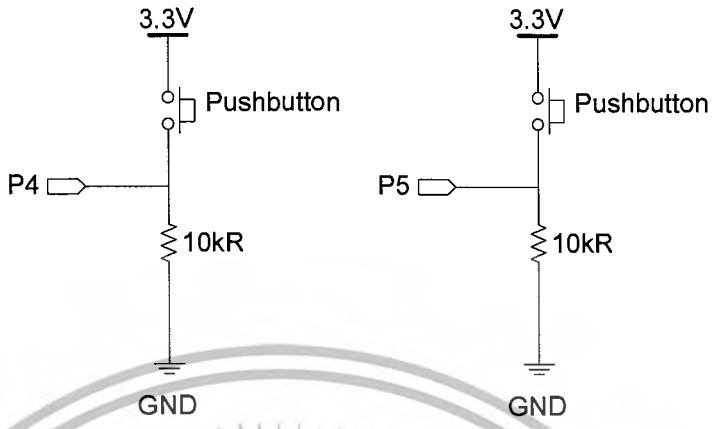
P16 ถึง P23 ต่อกับวงจรจัดสัญญาณเพื่อเชื่อมต่อกับมอนิเตอร์ VGA และต่อพ่วงเข้ากับ LED จำนวน 8 ดวง โดยมีสวิตช์จั๊มเปอร์เลือกการทำงาน

P26 ถึง P27 ต่อกับจุดต่อ PS2 สำหรับต่อคีย์บอร์ดภายนอก

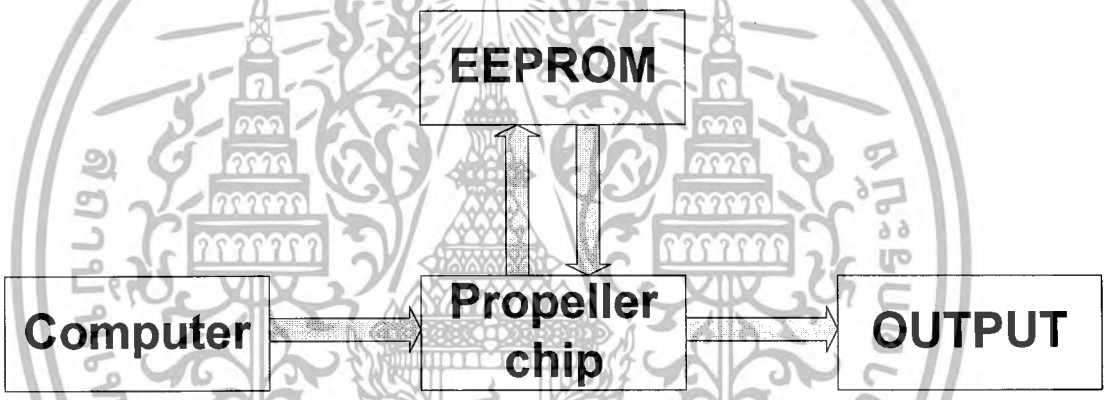
ซึ่งในการศึกษาจะใช้เอาท์พุตที่เป็น LED (P16 ถึง P23) และ อินพุตเป็นสวิตส์แบบกดติดปล่อยดับ (P4 และ P5) ซึ่งมีการต่อวงจรดังรูปที่ 3.3 และ 3.4 ตามลำดับ



รูปที่ 3.3 วงจรอินพุตที่ P16 ถึง P23



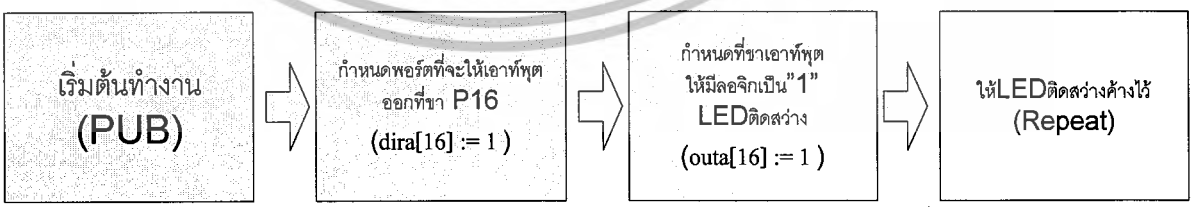
รูปที่ 3.4 วงจรอินพุตที่ P4 และ P5



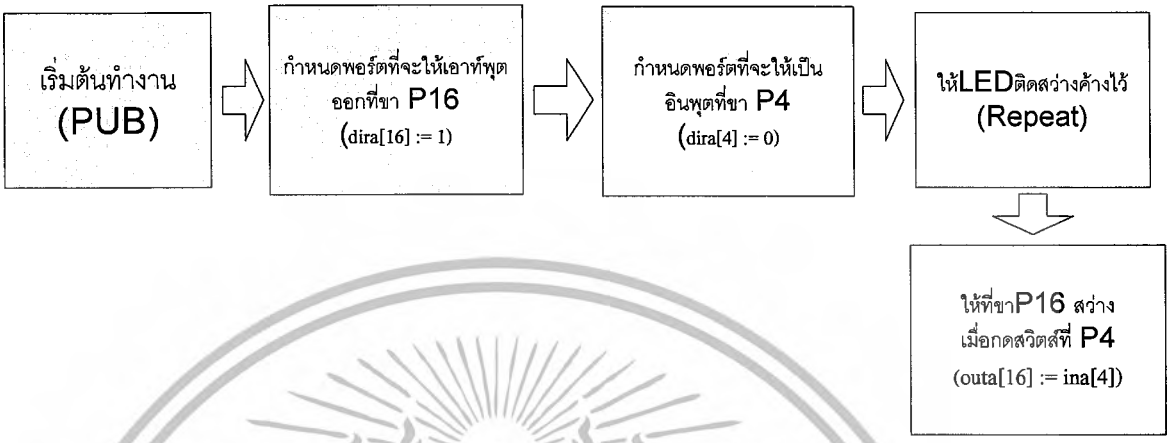
รูปที่ 3.5 บล๊อกไดอะแกรมการทำงานของชิปโพรเพลเลอร์เบอร์ P8X32A-D40 โดยใช้อีอีพรอม

ตอนที่ 1 ศึกษาการทำงานพื้นฐานของชิปโพรเพลเลอร์เบอร์ P8X32A-D40 และภาษาสปิน

1.1 คำสั่งให้LEDที่ P16 สว่าง

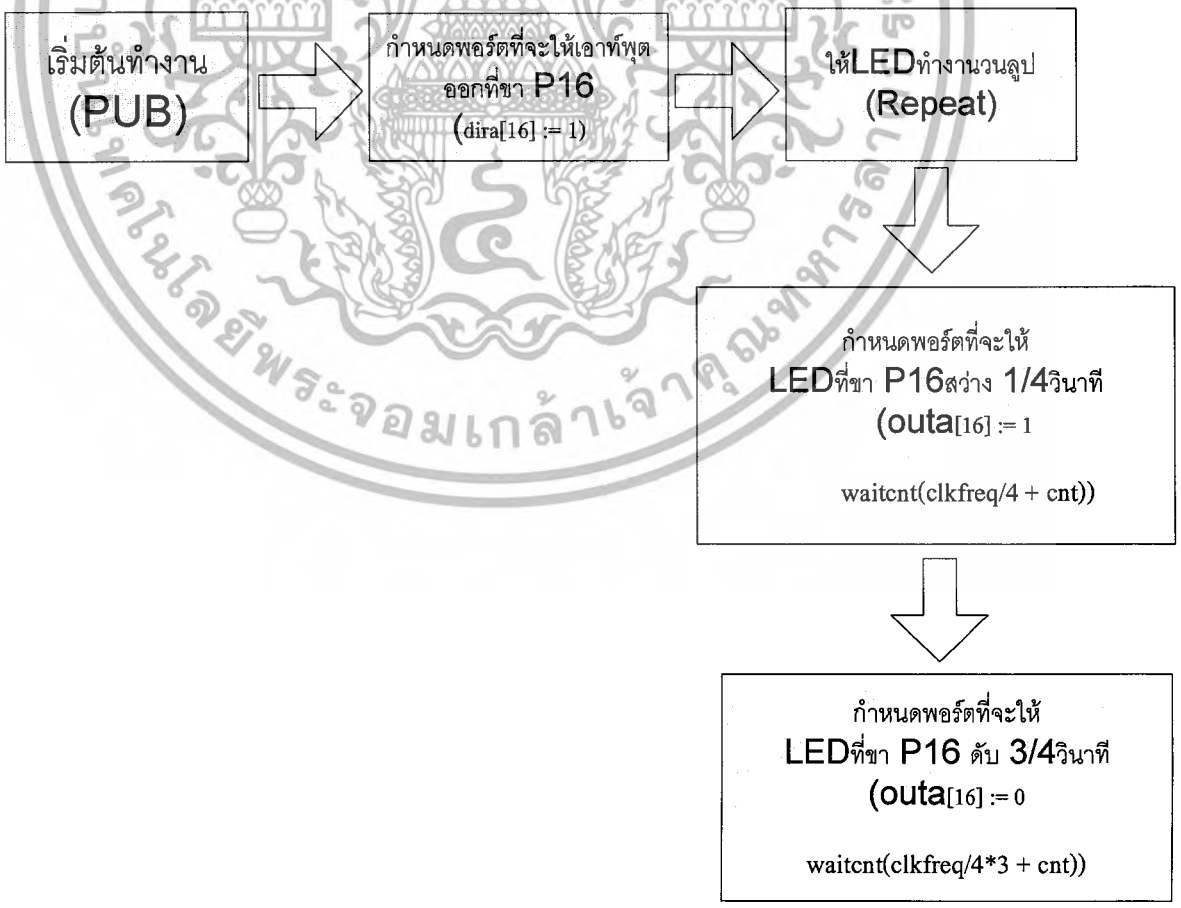


1.2 กดปุ่มที่ P4 แล้ว LED ที่ P16 สว่าง

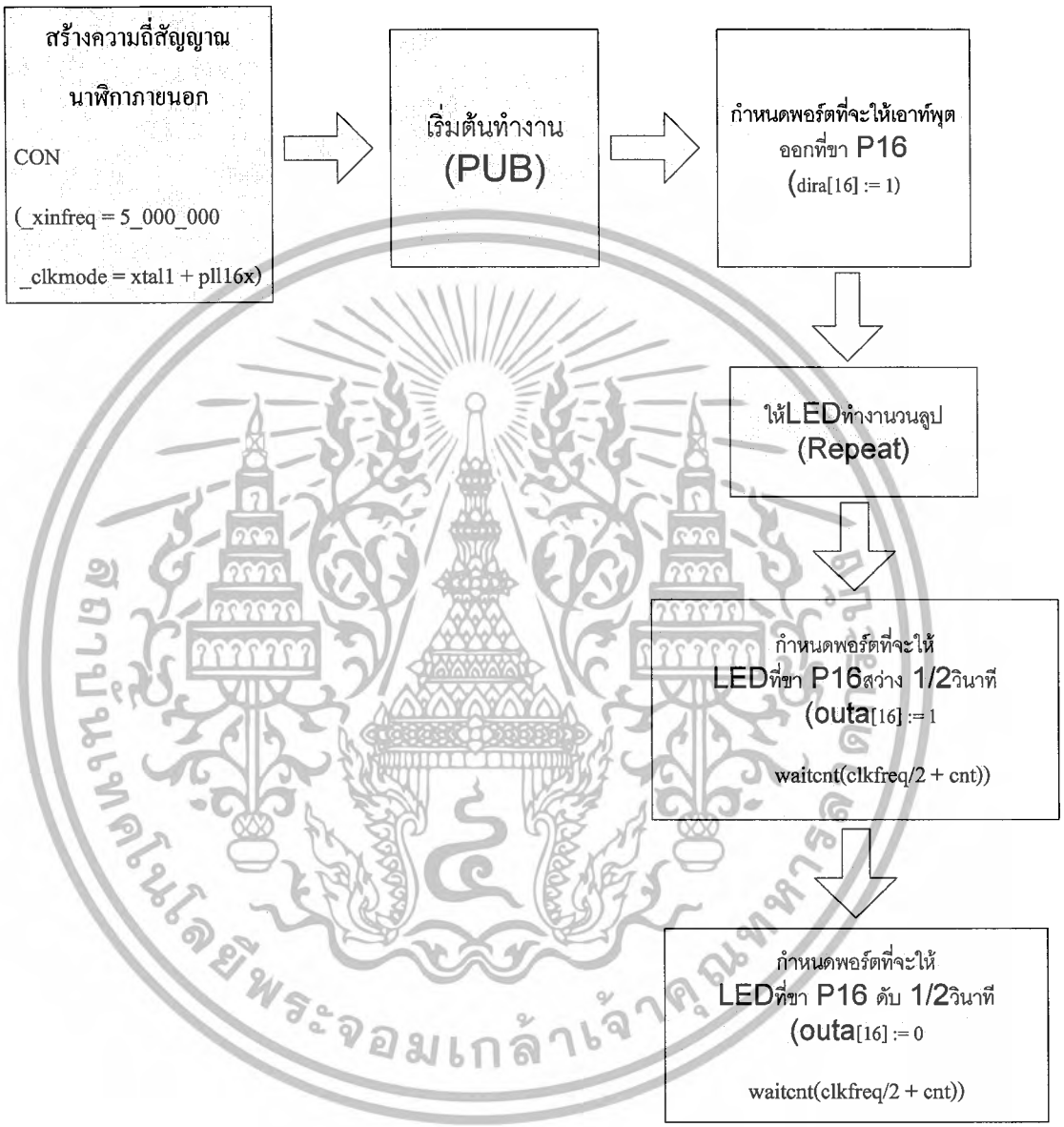


1.3 ให้LEDกระพริบ โดยใช้คำสั่ง waitcnt และใช้ความถี่ภายใน

โดยมีระยะการติด 1/4 s และ ดับ 3/4 s



1.4 ให้ LED กระพริบด้วยอัตราคงที่ โดยใช้คริสตอลภายนอกความถี่ต่ำ (XTAL1) โดยมีระยะการติด 1/2 วินาที และ ดับ 1/2 วินาที



1.5 กำหนดให้มีการทำงานพร้อมกันหลายค๊อก

ใช้คำสั่ง Cognew ในการสร้างค๊อกเพิ่ม

1.6 การเขียนโปรแกรมโดยใช้หลายออบเจ๊ก

โดยให้มีส่วนรับค่า และแสดงผล แยกจากกัน แต่นำมารวมกันด้วยเป็นโปรแกรมเดียวกัน โดยการดึงมาใช้ด้วยคำสั่ง obj

ตอนที่ 2 นำมาประยุกต์ใช้เพื่อออกแบบระบบเตือนรถไฟอัตโนมัติ โดยได้ทำเป็นแบบจำลอง

การออกแบบระบบเตือนรถไฟ

ในการออกแบบประกอบด้วยสองส่วนดังต่อไปนี้

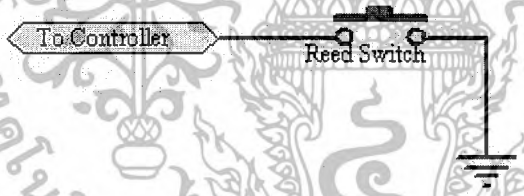
1. ส่วนของวงจร

2.ส่วนของโปรแกรม

3.1 ส่วนของวงจร ประกอบด้วย อินพุต มัลติคอร์ไมโครคอนโทรลเลอร์ และเอาท์พุต

รูปที่ 3.6 บล็อกไดอะแกรมแสดงส่วนประกอบของวงจร

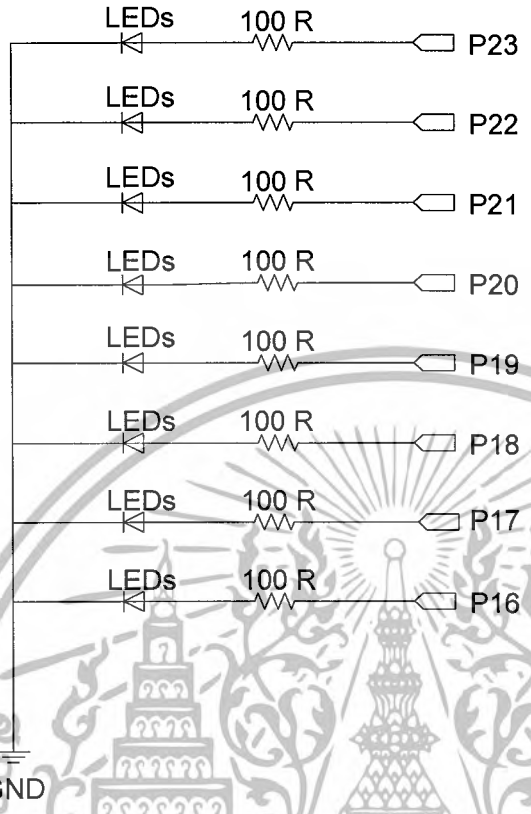
3.1.1 ส่วนอินพุต ได้แก่ ส่วนเซนเซอร์เป็นรีดสวิตช์



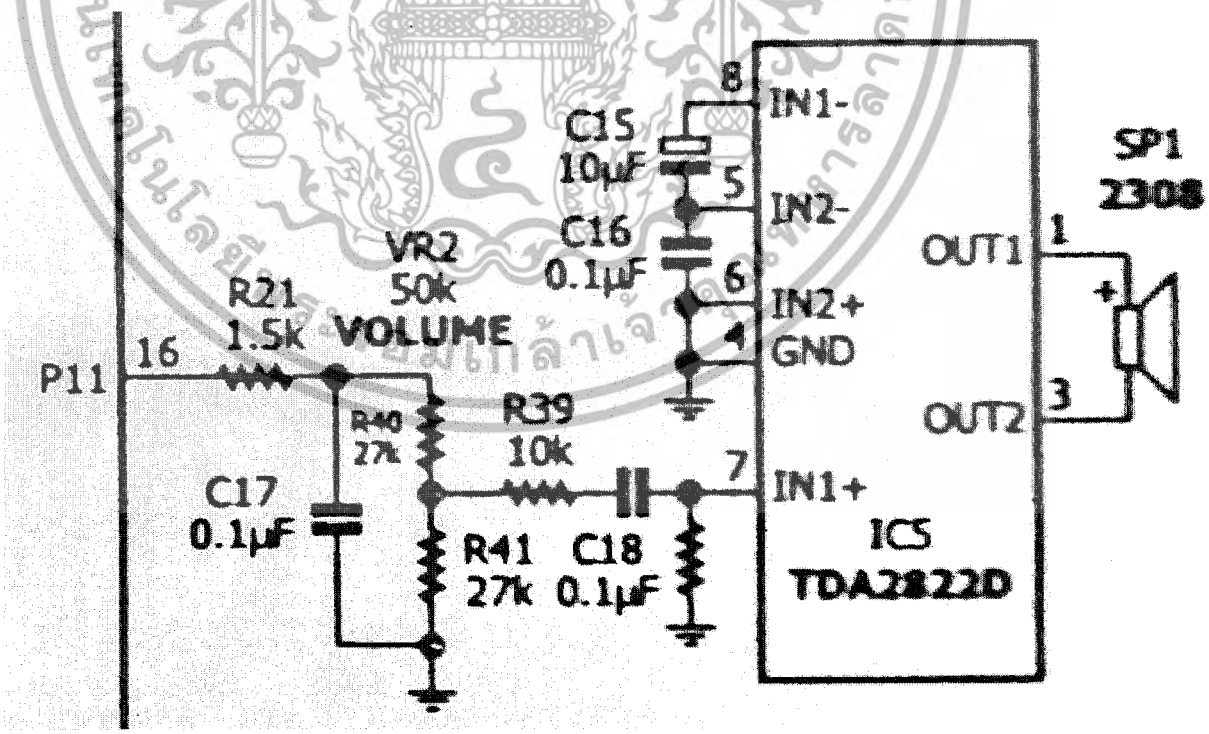
รูปที่ 3.7 รีดสวิตช์

3.1.2 ส่วนมัลติคอร์ไมโครคอนโทรลเลอร์ เป็นส่วนฮาร์ดแวร์สำเร็จรูปซึ่งมีคุณสมบัติดังที่ได้กล่าวมาแล้ว

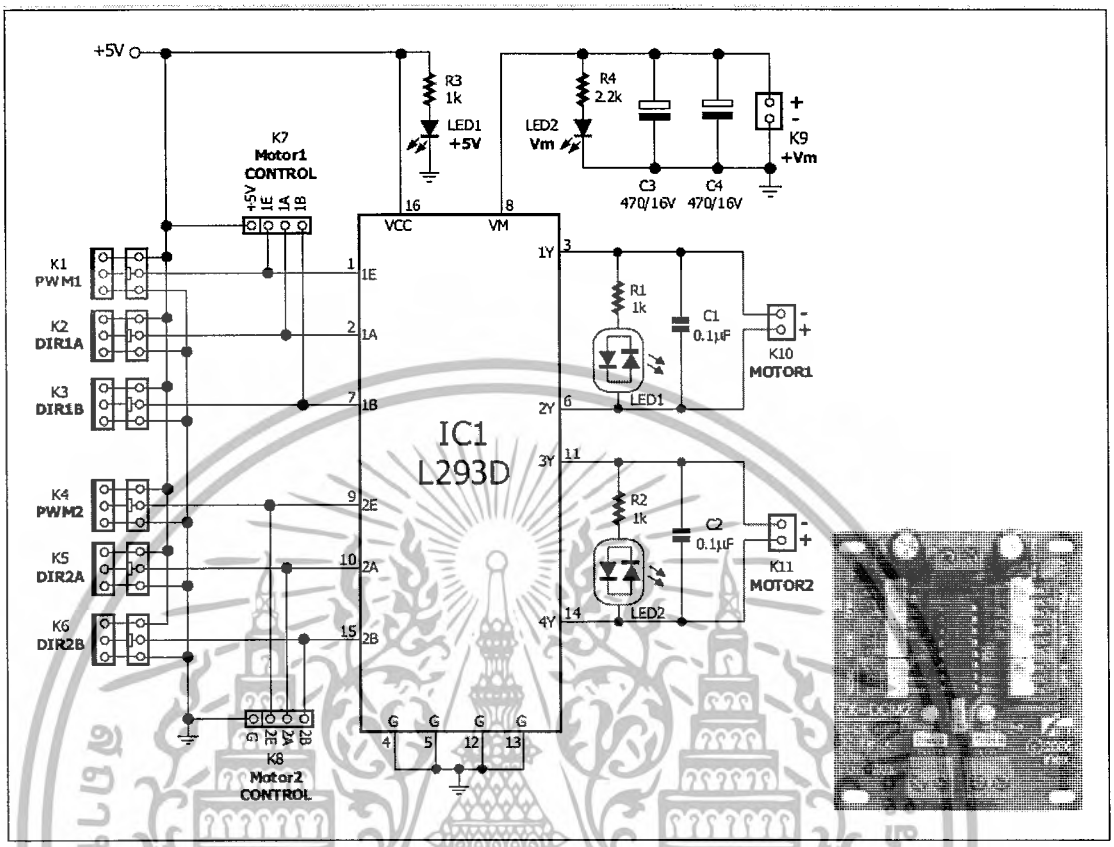
3.1.3ส่วนเอาท์พุต ได้แก่ สัญญาณไฟเตือน สัญญาณเสียงเตือน และมอเตอร์ปิด-เปิดแผงที่กั้นรถไฟ



รูปที่ 3.8 วงจรสัญญาณไฟ



รูปที่ 3.9 วงจรสัญญาณเสียง



รูปที่ 3.10 วงจรขับมอเตอร์

วงจรขับมอเตอร์ใช้ไอซีขับมอเตอร์เบอร์ L293D สามารถขับมอเตอร์ได้ 2 ช่อง ใช้ไฟเลี้ยงระหว่าง+6 ถึง+12V แยกจากกัน และต่อกับมอเตอร์ได้ 2 ช่องทาง

รูปแบบการขับมอเตอร์ไฟตรงให้ทำงานมี 4 รูปแบบ คือ

1.ขับให้มอเตอร์หมุนตามเข็มนาฬิกา

2.ขับให้มอเตอร์หมุนทวนเข็มนาฬิกา

3.ปล่อยแกนมอเตอร์อิสระ

4.ล็อกแกนมอเตอร์

การควบคุมมอเตอร์ทั้งหมดของไอซี L293D ได้รับการกำหนดจากสัญญาณลอจิก ดังแสดงรายละเอียดดังตารางที่ 3.1

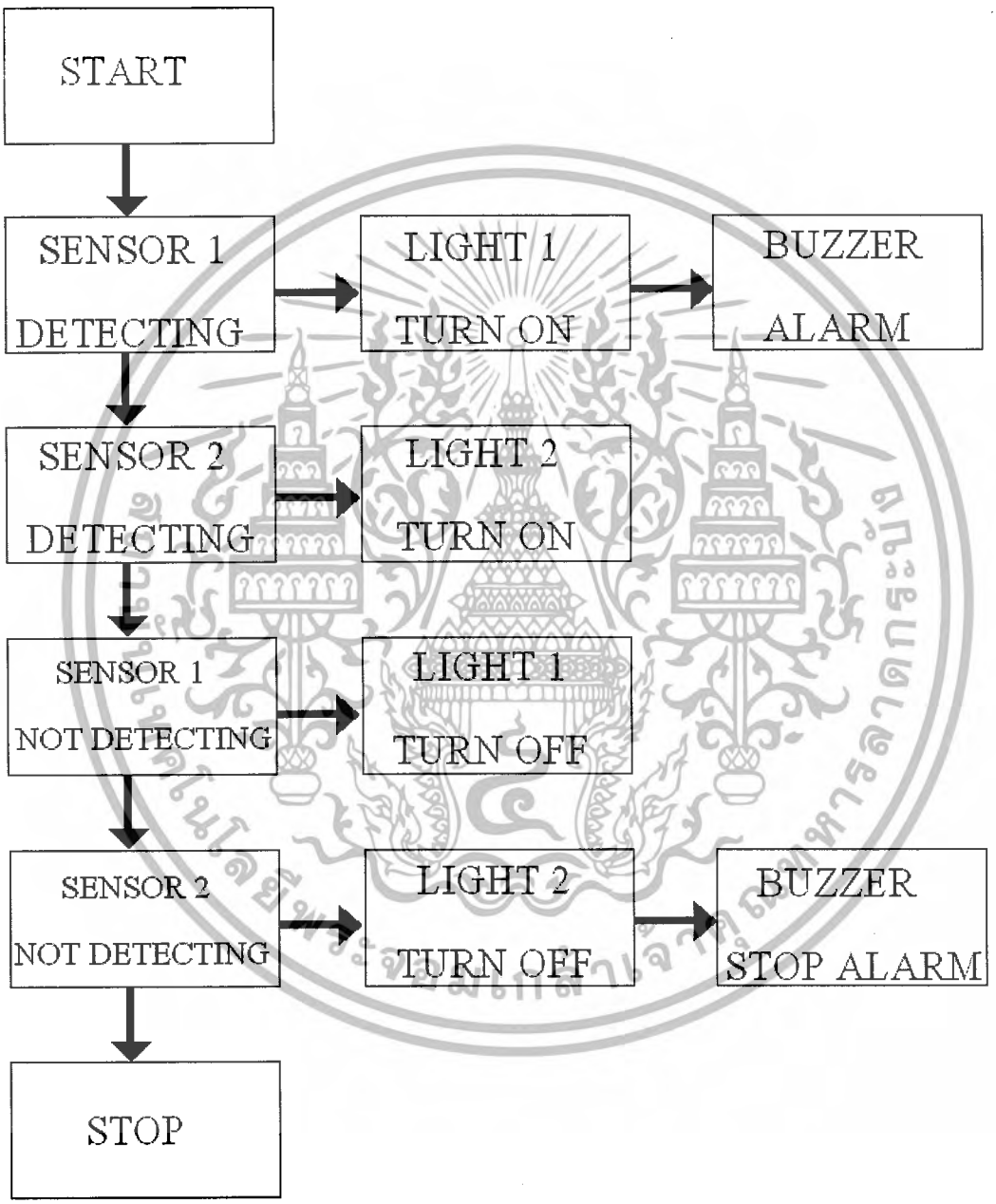
ตารางที่ 3.1 แสดงการป้อนลอจิกเพื่อกำหนดทิศทางของมอเตอร์

ขา 1E/2E	ขา1A/2A	ขา1B/2B	การทำงานของมอเตอร์
0	X	X	แกนมอเตอร์เป็นอิสระ
1	0	0	แกนมอเตอร์ถูกล็อก
1	0	1	มอเตอร์หมุนตามเข็มนาฬิกา
1	1	0	มอเตอร์หมุนทวนเข็มนาฬิกา
1	1	1	แกนมอเตอร์ถูกล็อก

X หมายถึง เป็นลอจิก “0” หรือ “1” ก็ได้

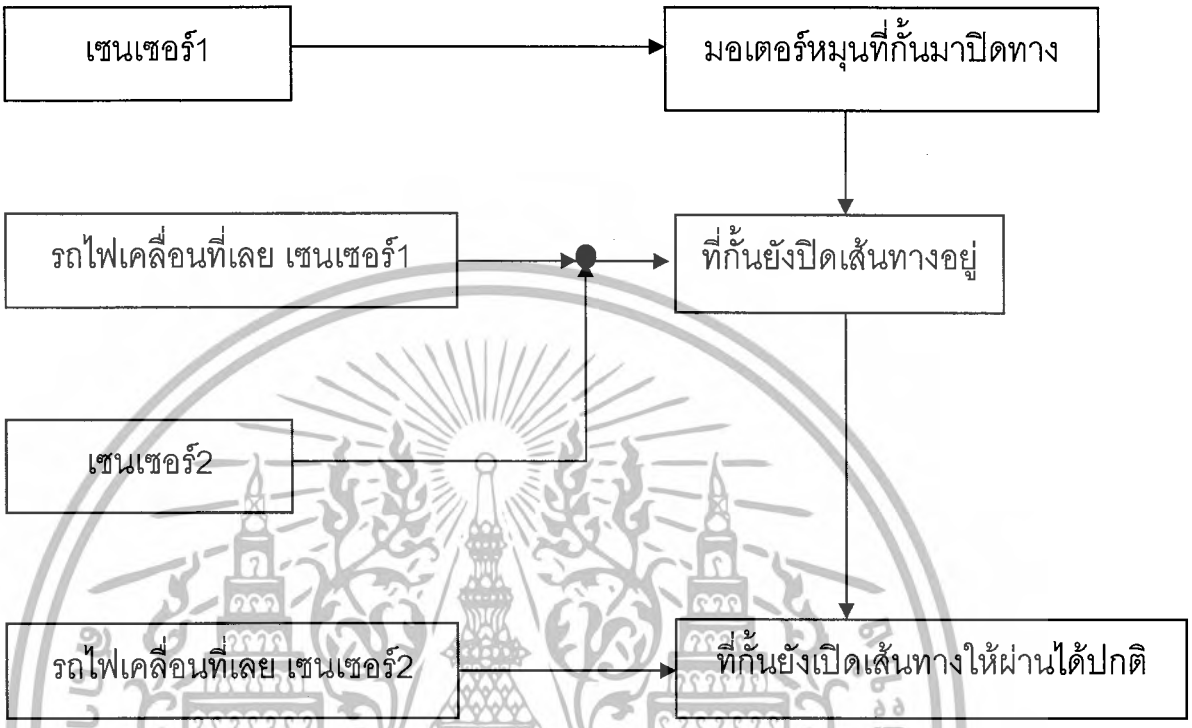
3.2 ส่วนโปรแกรมสำหรับควบคุมระบบ

3.2.1 โปรแกรมสำหรับควบคุมระบบสัญญาณเตือน



รูปที่ 3.11 บล็อกไดอะแกรมแสดงการทำงานของโปรแกรมสำหรับควบคุมระบบสัญญาณ

2. โปรแกรมสำหรับควบคุมมอเตอร์ปิด-เปิดแผงที่กั้นรถไฟ

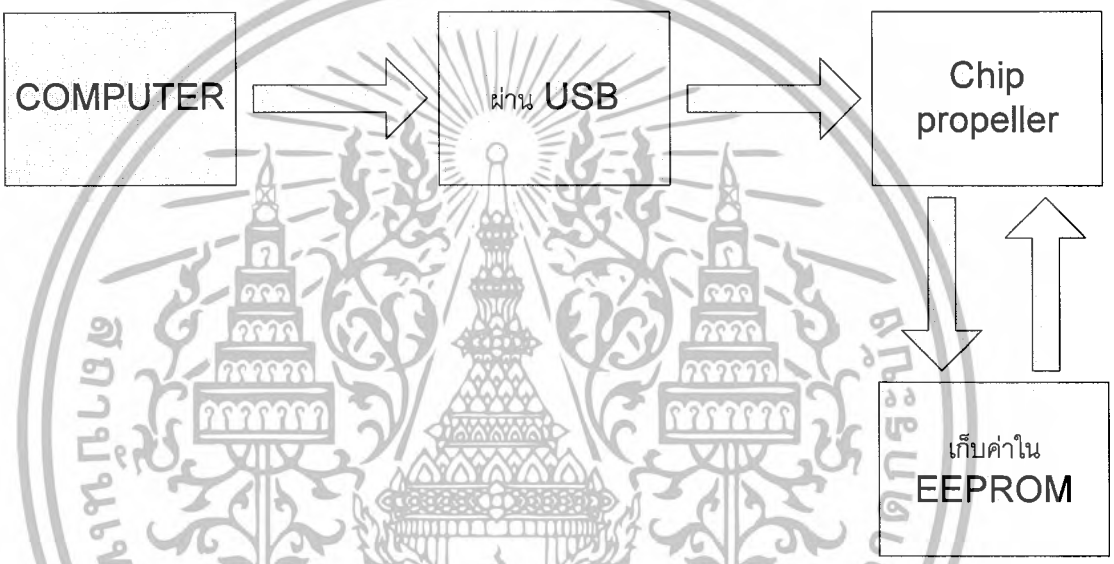


รูปที่ 3.12 บล็อกไดอะแกรมแสดงการทำงานของโปรแกรมสำหรับควบคุมการปิด - เปิดแผงที่กั้นรถไฟ

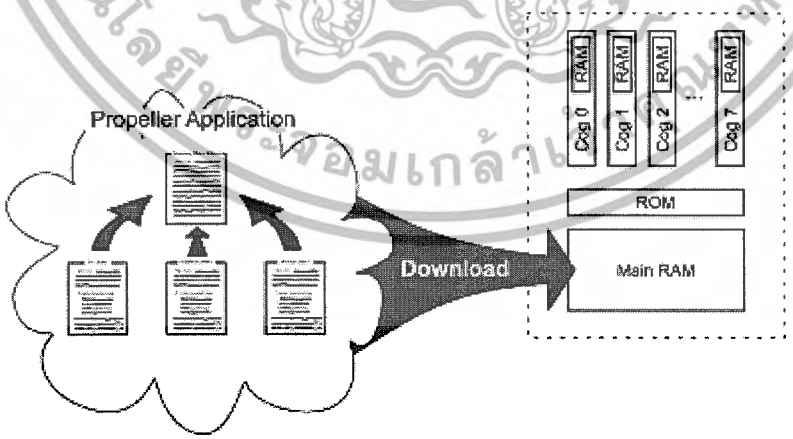
บทที่ 4

การทดลอง และผลการทดลอง

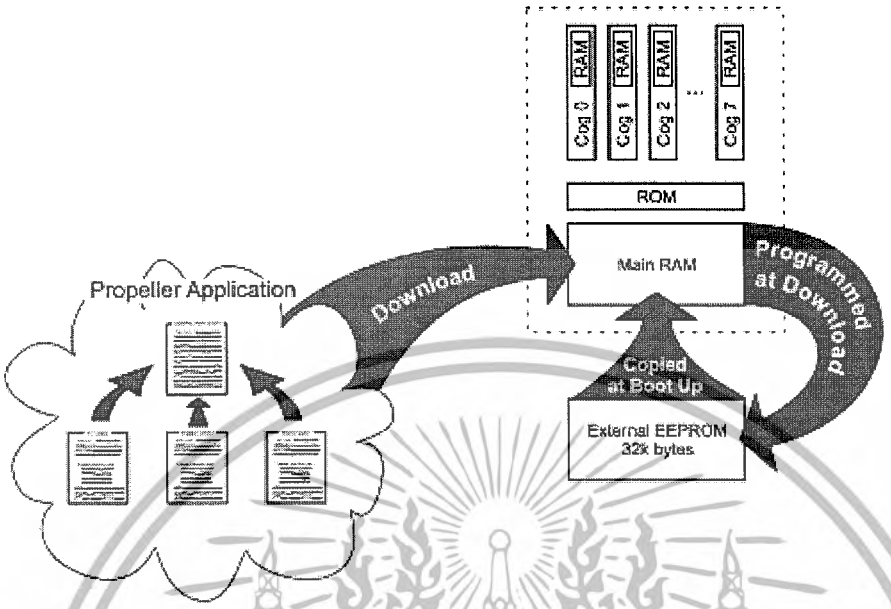
ในบทนี้เป็นการศึกษาการทำงานของชิปโพรเพลเลอร์ ภาษาสปิน และนำมาประยุกต์ใช้เพื่อออกแบบระบบเตือนรถไฟอัตโนมัติ โดยได้ทำออกมาเป็นแบบจำลอง



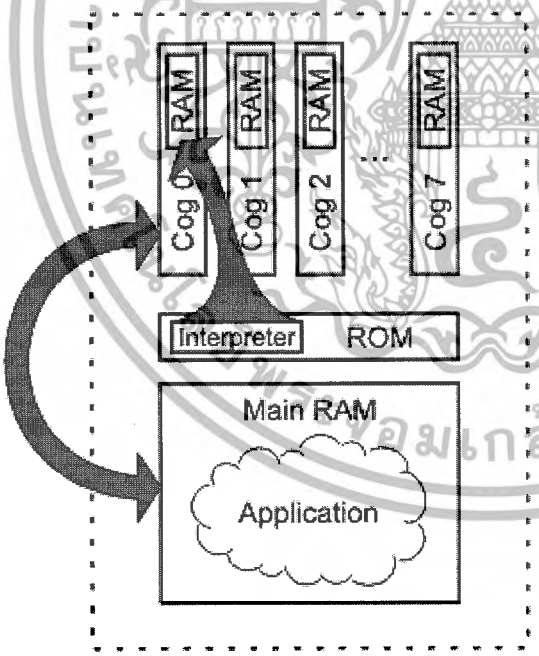
รูปที่ 4.1 บล็อกไดอะแกรมแสดงการดาวน์โหลดโปรแกรมลงชิปโพรเพลเลอร์



รูปที่ 4.2 การดาวน์โหลดโปรแกรมลงในแรมภายในชิป

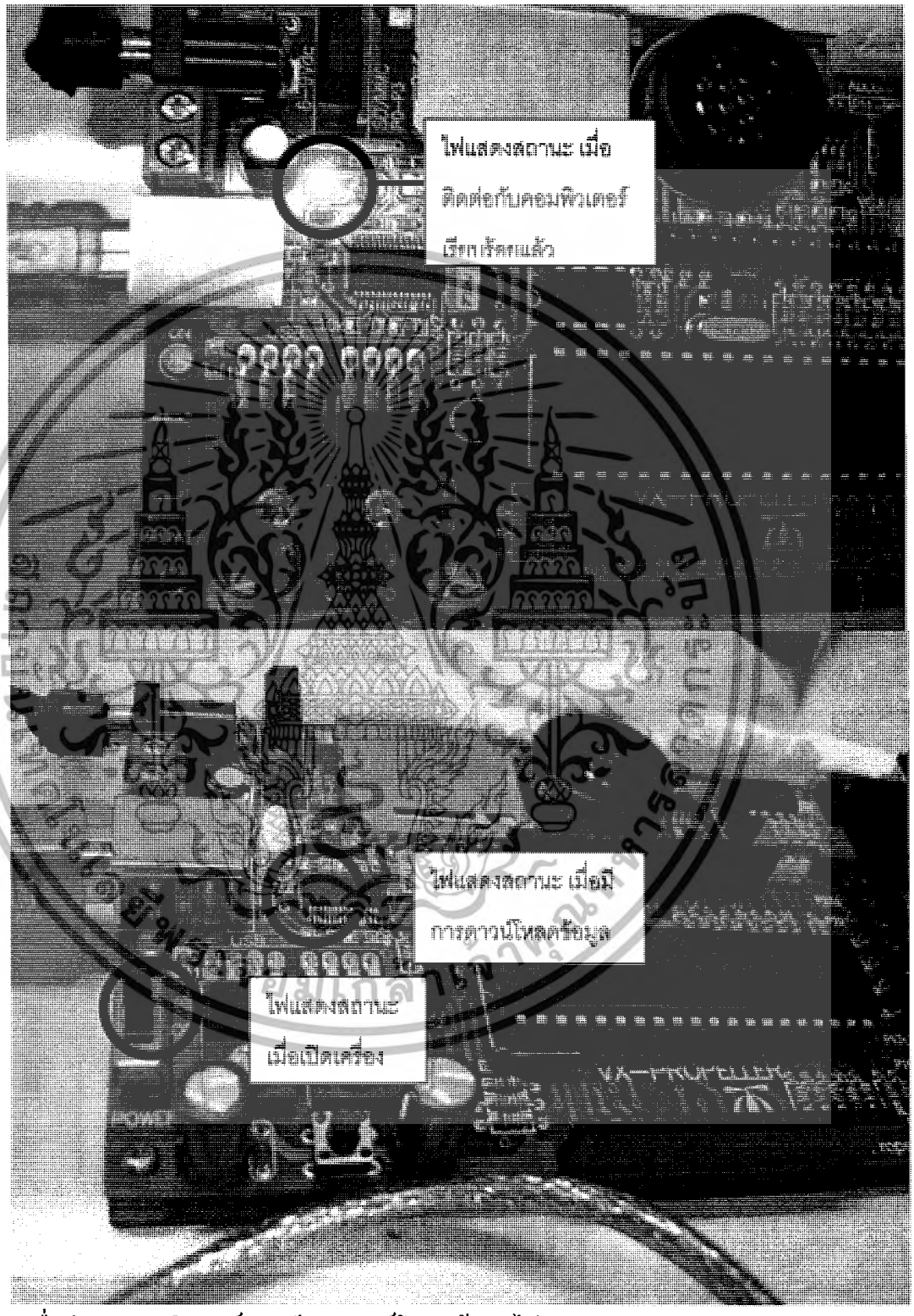


รูปที่ 4.3 การดาวน์โหลดโปรแกรมลงในEEPROM



รูปที่ 4.4 การทำงานของโปรแกรมการส่งข้อมูลไปยังค็อก

เมื่อต่อสายUSBจากคอมพิวเตอร์



รูปที่ 4.5 เมื่อต่อกับคอมพิวเตอร์ และมีการดาวน์โหลดข้อมูลไปยังPropeller

ตอนที่ 1 ศึกษาการทำงานพื้นฐานของชิปโพรเพลเลอร์เบอร์ P8X32A-D40 และภาษาสปิน ได้ทำการทดลอง และมีผลการทดลองดังต่อไปนี้

1.1 คำสั่งให้LEDที่ขา P16 สว่าง

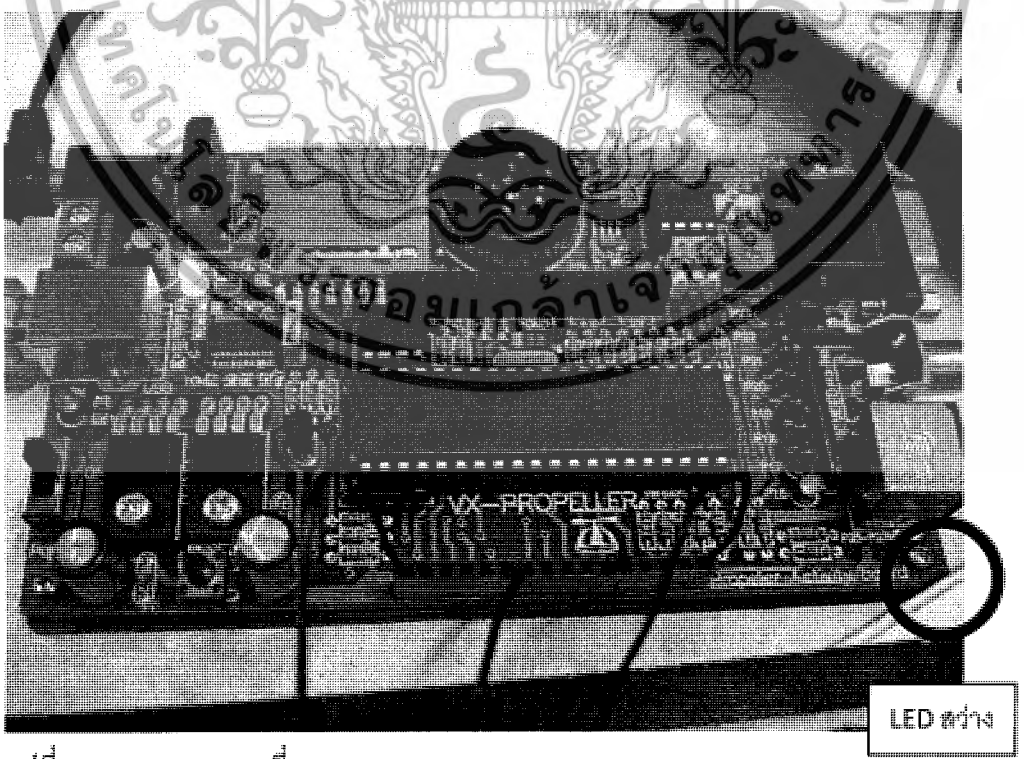
```
PUB LedOn
 dira[16] := 1
 outa[16] := 1
 repeat
```

ในการเริ่มต้นโปรแกรมจะใช้คำสั่งPUBเพื่อสร้างโปรแกรมย่อย และPUBส่วนแรกที่พบจะถือว่าเป็นส่วนที่เรียกใช้เป็นอันดับแรก ถ้าจะเปรียบเทียบกับในภาษาซี ก็จะหมายถึงส่วนของmain

คำสั่งDira[16]:=1 เป็นการกำหนดให้เป็นขาเอาท์พุต ซึ่งคือขา16

คำสั่งOuta[16]:= 1 เป็นคำสั่งเพื่อส่งค่าลอจิก “0” หรือ “1” ออกไปยังพอร์ตที่ต้องการ ซึ่งในที่นี้ เป็นการส่งค่าลอจิก “1” ไปยังพอร์ตที่ 16

คำสั่งRepeat เป็นคำสั่งทำการวนลูปโดยกำหนดจำนวนครั้งที่ตัวแปรcount แต่ถ้าไม่ได้กำหนดจะวนลูปแบบไม่มีสิ้นสุด

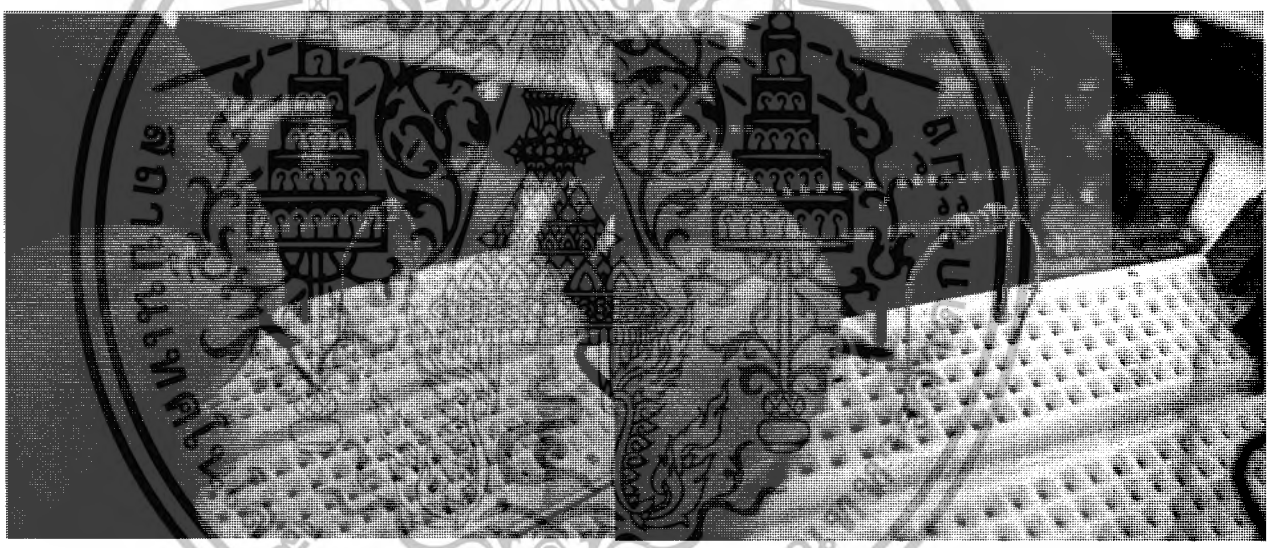


รูปที่ 4.6 ผลการทดลองที่ 1.1

1.2 กดปุ่มที่P4 แล้ว LEDที่ P16 สว่าง

```
PUB ButtonLed
  dira[16]  := 1
  dira[4] := 0
  repeat
    outa[16] := ina[4]
```

ในการเริ่มต้นโปรแกรมจะเหมือนกับตอนที่1.1 แต่ในตอนที่ 1.2 นี้ ได้เพิ่มส่วนอินพุตโดยใช้คำสั่ง dira[4]:=0 ซึ่งหมายถึง การกำหนดขาอินพุตที่ขา4 และ outa[16] := ina[4] ซึ่งเป็นการบอกว่าค่าลอจิกจากตำแหน่งขา4 จะส่งค่าไปยังตำแหน่งขา 16



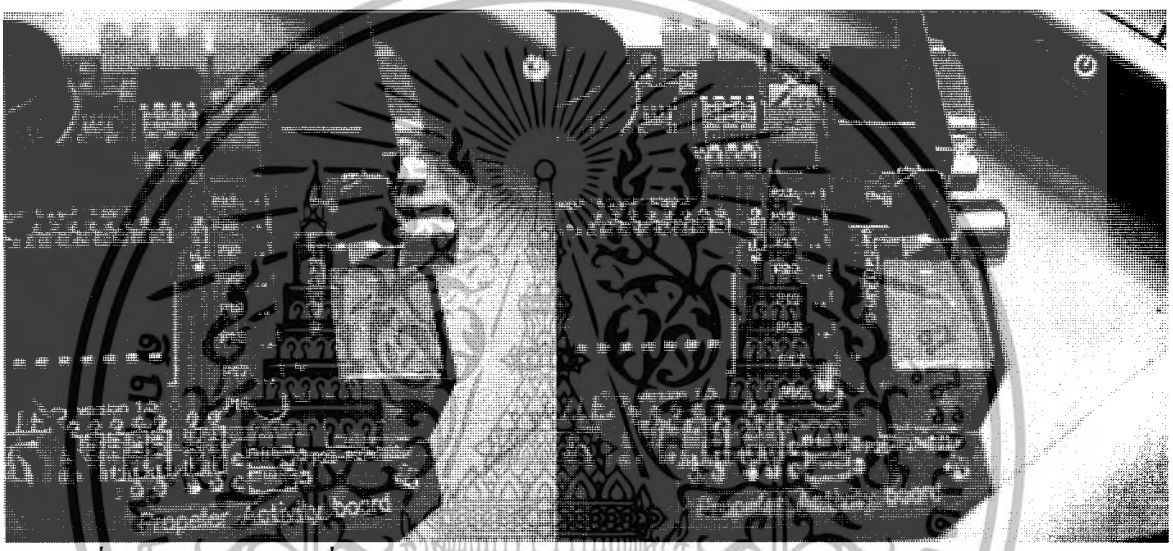
รูปที่ 4.7 ผลการทดลองที่ 1.2

1.3 ให้LEDกระพริบ โดยใช้คำสั่ง waitcnt และใช้ความถี่ภายใน โดยมีระยะการติด 1/4 วินาที และดับ 3/4 วินาที

```
PUB LedOnOff
  dira[16] := 1
    repeat
   outa[16] := 1
    waitcnt(clkfreq/4 + cnt)
```

```
outa[16] := 0
waitcnt(clkfreq/4*3 + cnt)
```

ในตอนที่ 1.3 นี้ ทำการทดลองเพื่อสังเกตการณ์ทำงานของคำสั่ง waitcnt( ) เพื่อเป็นการหน่วงเวลา โดยตรวจค่าเวลาของระบบ แล้วบวกเข้าไปด้วยค่าเวลาที่ต้องการหน่วง จากโปรแกรม waitcnt(clkfreq/4 + cnt) หมายถึง ค่าเวลาบวกไปจากค่าปัจจุบัน clkfreq/4 โดยที่หนึ่งclkfreqเท่ากับ 1 วินาที

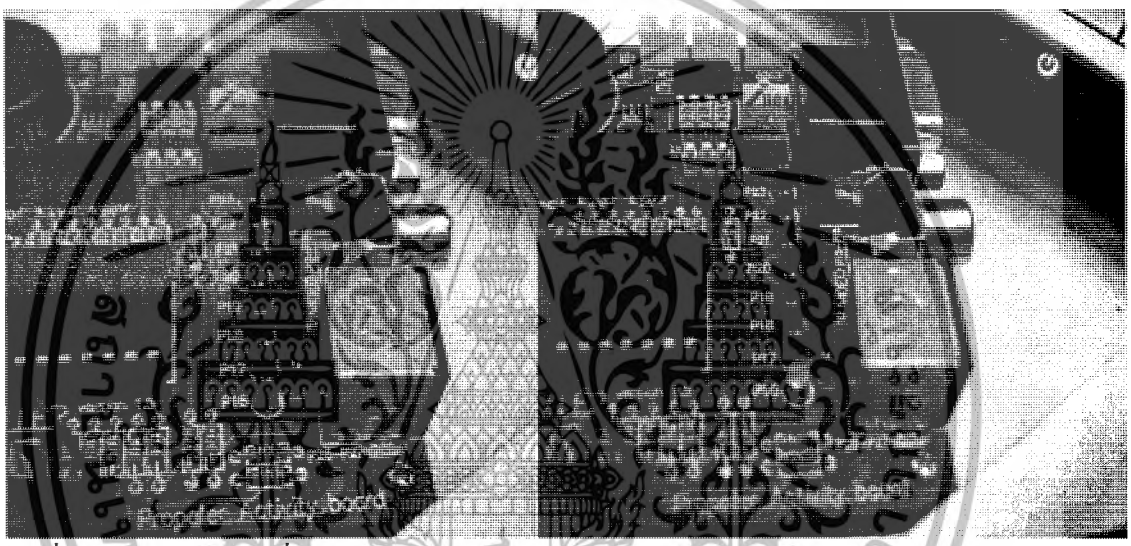


รูปที่ 4.8 ผลการทดลองที่ 1.3

1.4 ให้ LED กระพริบด้วยอัตราคงที่ โดยใช้คริสตอลภายนอกความถี่ต่ำ (XTAL1)

```
CON
  _clkmode = xtal1 + pll16x
   _xinfreq = 5_000_000
    PUB LedOnOff
      dira[16] := 1
        repeat
       outa[16] := 1
        waitcnt(clkfreq/2 + cnt)
       outa[16] := 0
      waitcnt(clkfreq/2 + cnt)
```

ในตอนที่ 1.4 นี้ ทำการทดลองเพื่อสังเกตการณ์ทำงานของคำสั่ง con ที่สำหรับเก็บค่าคงที่แบบครอลคลุมทั้งโปรแกรม จากโปรแกรม _clkmode = xtal1 + pll16x หมายถึง กำหนดสัญญาณนาฬิกาจากภายนอกเป็นโหมดตริสตอลความถี่ต่ำ และกำหนดตัวคูณให้กับสัญญาณนาฬิกา 16 เท่า โดยกำหนดให้ความถี่คริสตอลเท่ากับ 5MHz (_xinfreq = 5_000_000) ดังนั้นความถี่ของสัญญาณนาฬิกาเท่ากับ 5MHz x 16 = 80 MHz

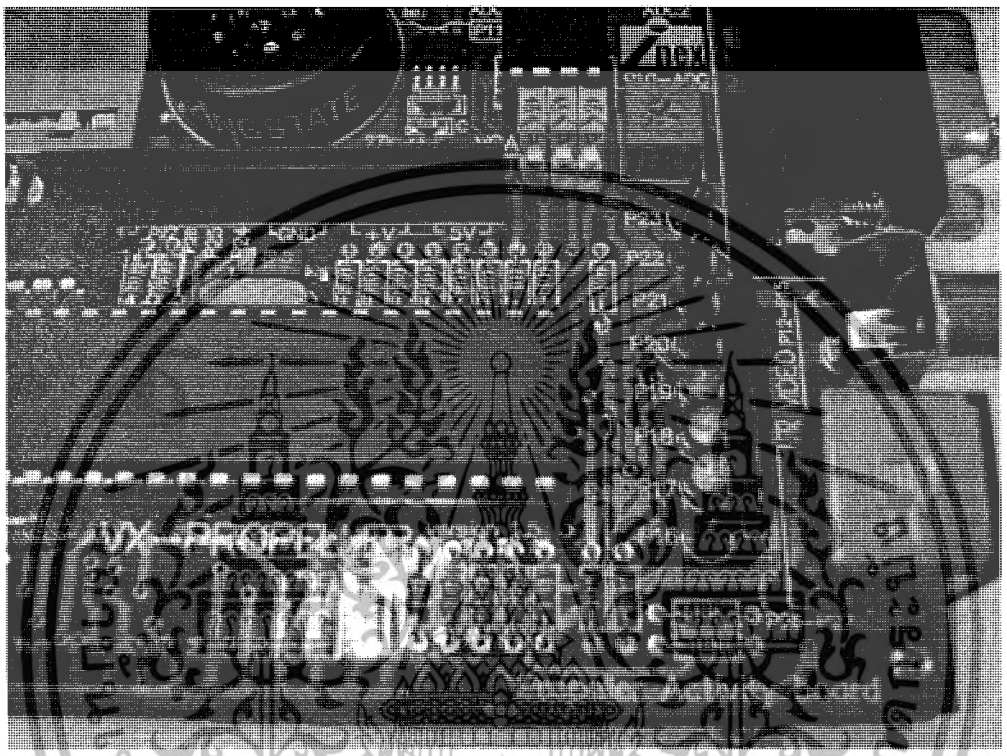


รูปที่ 4.9 ผลการทดลองที่ 1.4

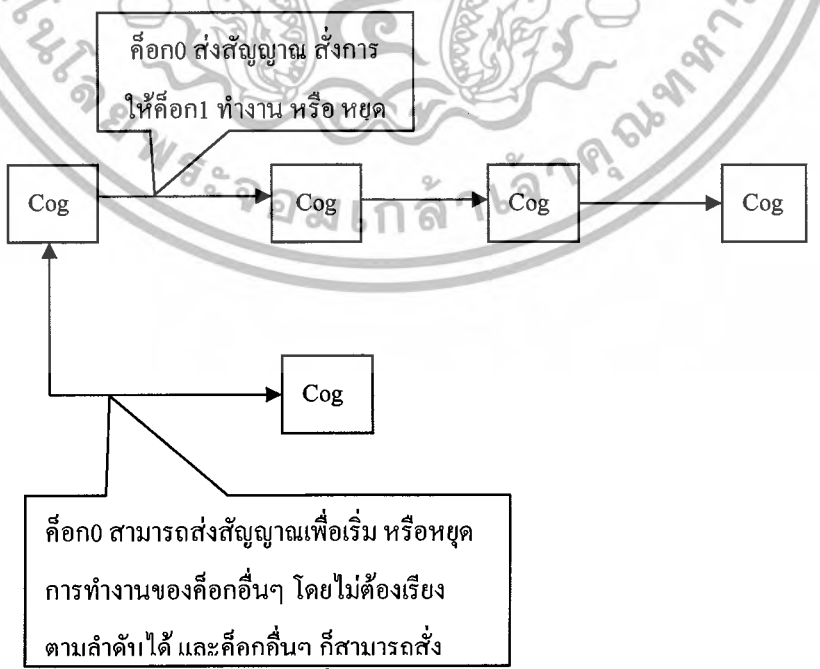
1.5 กำหนดให้มีการทำงานพร้อมกันหลายค๊อก

```
VAR
   long stack[30]
PUB LaunchBlinkCogs
   cognew(Blink(17, clkfreq/3, 9), @stack[0])
   cognew(Blink(18, clkfreq/7, 21), @stack[10])
   cognew(Blink(19, clkfreq/11, 39), @stack[20])
PUB Blink( pin, rate, reps)
   dira[pin]~~
   outa[pin]~
```

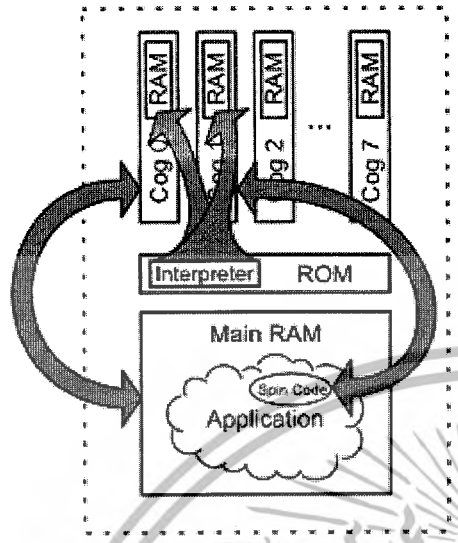
```
repeat reps * 2
  waitcnt(rate/2 + cnt)
  !outa[pin]
```



รูปที่ 4.10 ผลการทดลองที่ 1.5



รูปที่ 4.11 บล็อกไดอะแกรมแสดงการสื่อสารระหว่างค็อก



รูปที่ 4.12 การทำงานของโปรแกรมการส่งข้อมูลไปยังหลายค๊อก

1.6 การเขียนโปรแกรมโดยใช้หลายออบเจ็ก

```
OBJ
   Blinker : "Blinker"
   Button  : "Button"
PUB ButtonBlinkTime | time
   repeat
     time := Button.Time(4)
     Blinker.Start(16, time, 20)
```

คำสั่งส่วนที่สั่งให้ไฟกระพริบ

คำสั่งส่วนรับอินพุต

CogObjectExample*

Full Source Condensed Summary Documentation

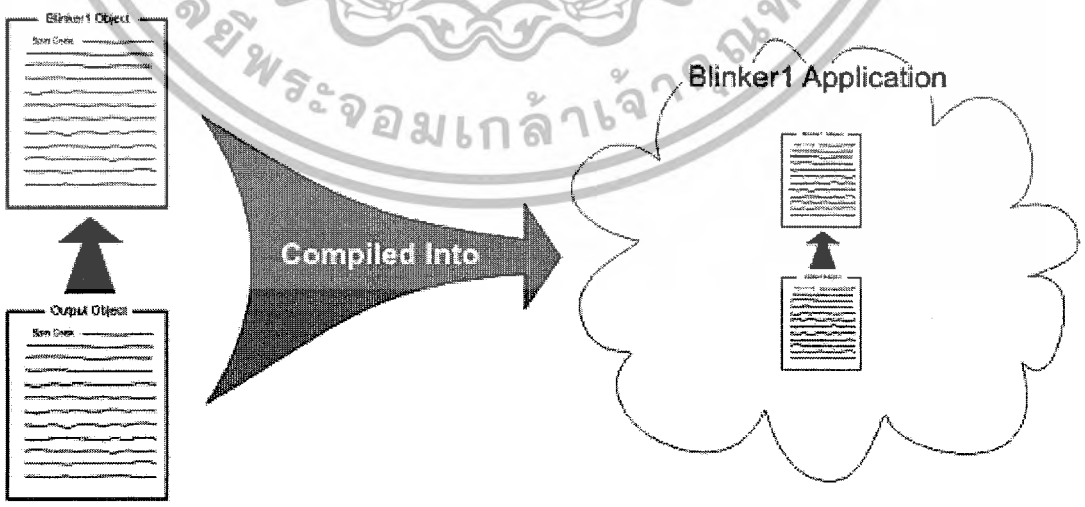
```
OBJ

    Blinker : "Blinker"
    Button  : "Button"


PUB ButtonBlinkTime | time

    repeat

        time := Button.Time(23)
        Blinker.Start(4, time, 20)
```

Blinker* Button*

Full Source Condensed Summary Documentation

```
VAR
  long  stack[10]                       'Cog stack space
  byte  cog                             'Cog ID

PUB Start(pin, rate, reps) : success
  Stop
  success := (cog := cognew(Blink(pin, rate, reps), @stack) + 1)

PUB Stop

  if cog
    cogstop(cog~ - 1)

PUB Blink(pin, rate, reps)

    dira[pin]~~
    outa[pin]~

    repeat reps * 2
      waitcnt(rate/2 + cnt)
      !outa[pin]
```

รูปที่ 4.13 ผลการทดลองที่ 1.6



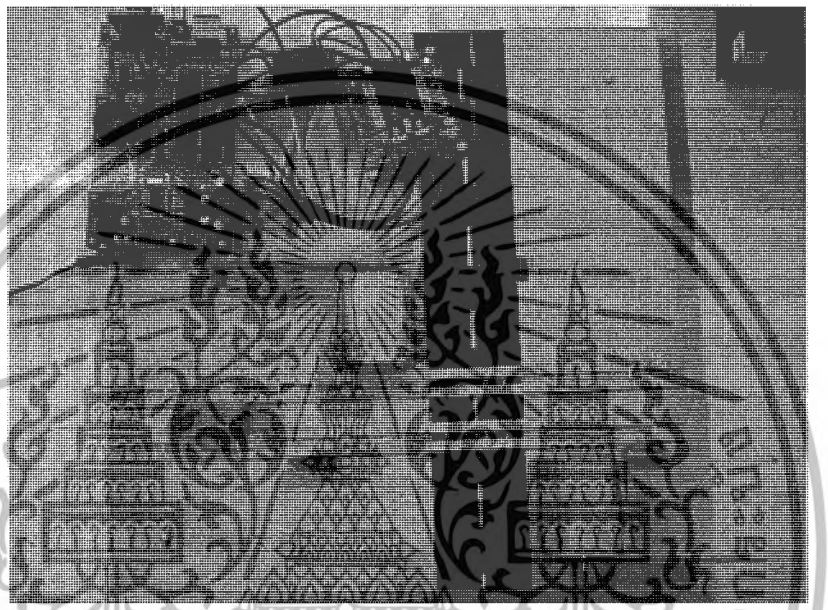
รูปที่ 4.14 การใส่objectลงในโปรแกรมหลัก

ตอนที่ 2 นำมาประยุกต์ใช้เพื่อออกแบบระบบเตือนรถไฟอัตโนมัติ

ได้ผลการทดลองดังนี้

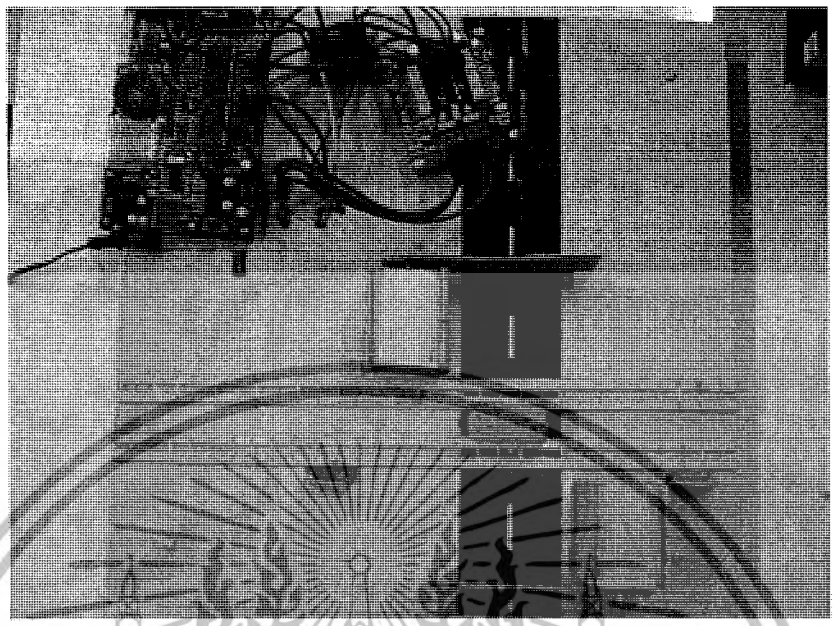
2.1 โปรแกรมสำหรับควบคุมระบบสัญญาณเตือน

รูปการทดลอง



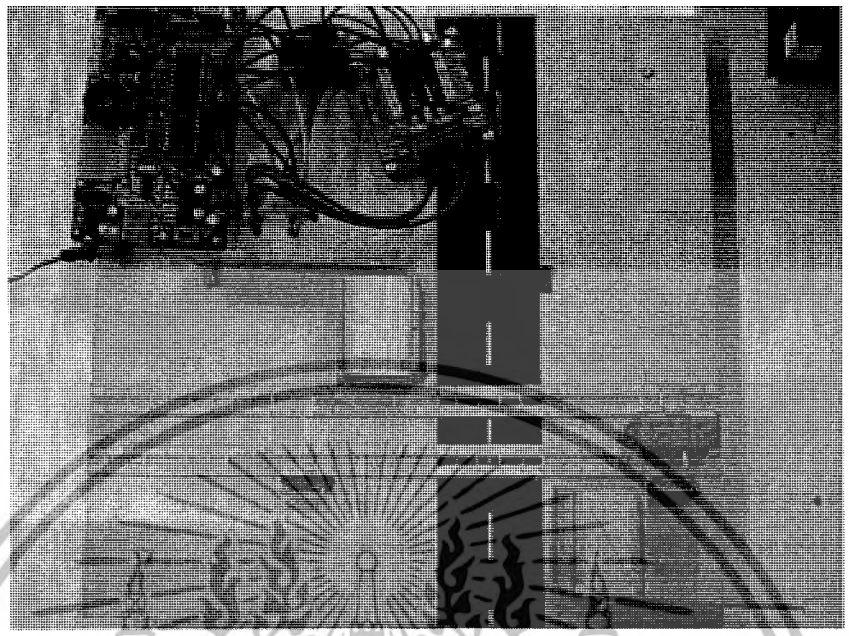
รูปที่ 4.15 รถไฟกำลังแล่นมา

รูปที่ 4.16 เซ็นเซอร์ตรวจเจอรถไฟ ทำให้สัญญาณไฟเตือนติดสว่างที่ P16 มีเสียงเตือน และ ที่กั้นปิดลง



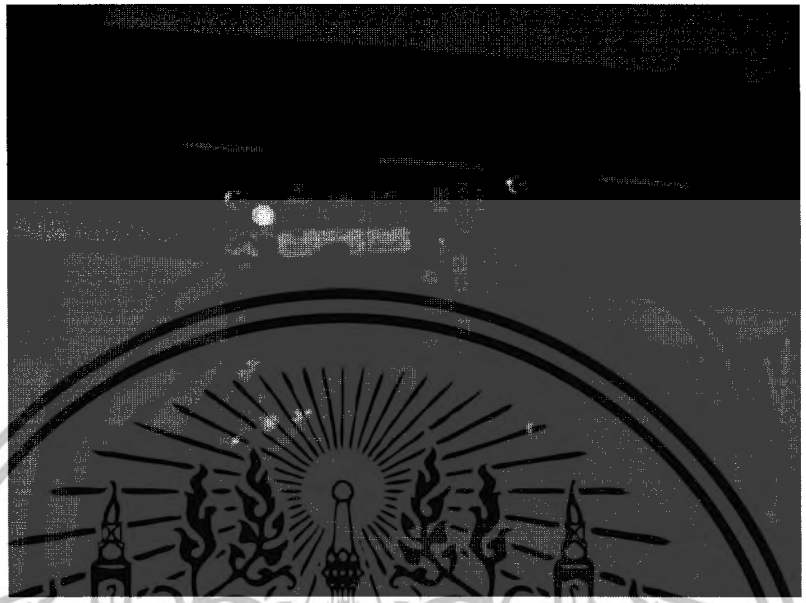
รูปที่ 4.17 ไฟสัญญาณ เสียงเตือนยังคงติด และที่กั้นยังคงปิดอยู่

รูปที่ 4.18 เซ็นเซอร์ตัวที่สองตรวจจับรถไฟ ที่กั้นยังปิดอยู่ เสียงเตือน และ ไฟเตือนติดอยู่



รูปที่ 4.19 เมื่อเลยเซ็นเซอร์ตัวที่สอง ที่กั้นจึงเปิดออก เสียงเตือน และไฟเตือนจึงดับลง

การทำงานของมอเตอร์สำหรับควบคุมการเปิดปิดที่กั้น



รูปที่ 4.20 แสดงสถานะของมอเตอร์หมุนทวนเข็ม (ที่กั้นปิด)

รูปที่ 4.21 แสดงสถานะของมอเตอร์หมุนตามเข็ม (ที่กั้นเปิด)

บทที่ 5

บทสรุป และข้อเสนอแนะ

ในปริญญานิพนธ์ฉบับนี้ ได้นำเสนอการศึกษาชิปโพรเพลเลอร์ ภาษาสปิน และนำมาออกแบบเป็นระบบเตือนรถไฟอัตโนมัติ

5.1 สรุปผล

จากการทดลองตอนที่ 1 ศึกษาการทำงานของชิปโพรเพลเลอร์ และ ภาษาสปิน ทำให้ทราบถึงความสามารถของชิปโพรเพลเลอร์ ในการเขียนโปรแกรมที่ไม่ยากต่อการศึกษา และง่ายต่อการนำไปพัฒนา ทั้งยังมีจำนวนค็อกมากถึงแปดค็อก โดยใช้สัญญาณนาฬิกาตัวเดียวกันทำให้การทำงานหลายค็กพร้อมกันนั้น จะได้งานออกมาพร้อมกันด้วย

จากการทดลองตอนที่ 2 เป็นการนำชิปโพรเพลเลอร์มาออกแบบเป็นระบบเตือนรถไฟอัตโนมัติ โดยในการทดลองทำการจำลองสถานการณ์ ขึ้นเพื่อทำการออกแบบระบบเพื่อให้ได้ระบบที่และพร้อมนำไปใช้ได้จริง ซึ่งในระบบสามารถรับค่าจากเซ็นเซอร์ เพื่อไปทำงานต่อยังส่วนควบคุมสัญญาณเตือน ที่มีทั้งเสียง และแสง และยังมีส่วนเปิด-ปิดแผงกั้น เพื่อกั้นรถให้หยุดเมื่อรถไฟมาได้

5.2 แนวทางการพัฒนาในอนาคต

ในการนำไปพัฒนานั้น จากอุปกรณ์ที่ทำขึ้นเป็นการจำลองระบบขึ้น หากต้องการนำไปใช้สามารถปรับปรุงและนำไปใช้ได้จริง โดยอุปกรณ์ที่จะใช้เป็นเซ็นเซอร์ สำหรับตรวจจับรถไฟอาจจะใช้เป็นเซ็นเซอร์วัดความสั่นสะเทือน หรือเซ็นเซอร์ตรวจจับโลหะได้ นอกจากนี้หากผู้ใช้ต้องการดัดแปลงเพิ่มเติมก็สามารถทำได้ เนื่องจากพื้นที่ให้ใช้งานเหลืออยู่

บรรณานุกรม

"ก้าวแรกกับ Propeller มัลติคอร์ไมโครคอนโทรลเลอร์",กฤษดา ใจเย็น และ ชัยวัฒน์ ลิ้มพรจิตร วิไล,inex innovative experiment

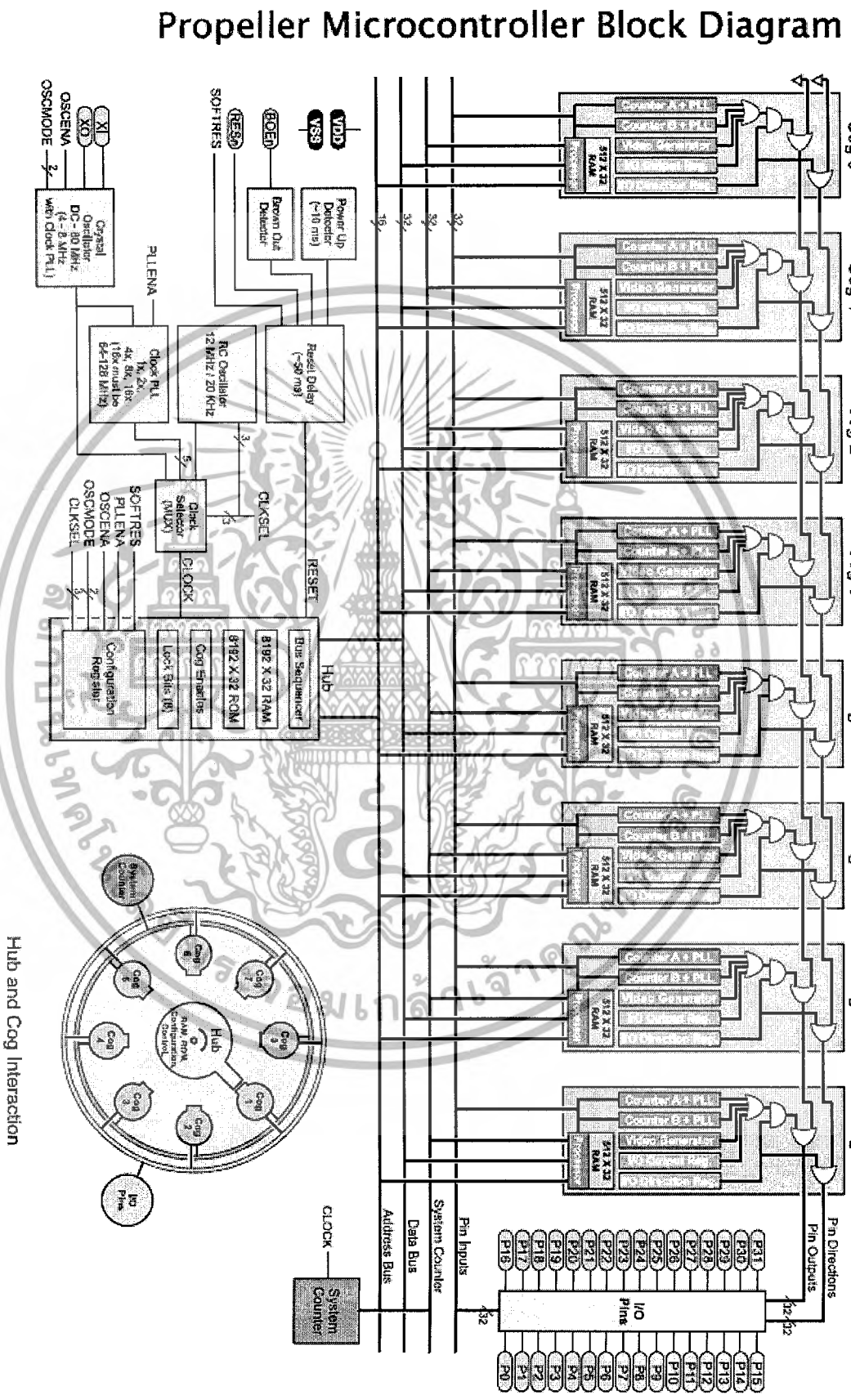
http://forums.parallax.com/

http://obex.parallax.com/

http://www.propellercode.com/

http://www.parallax.com/propeller

ภาคผนวก



Propeller™ P8X32A Datasheet

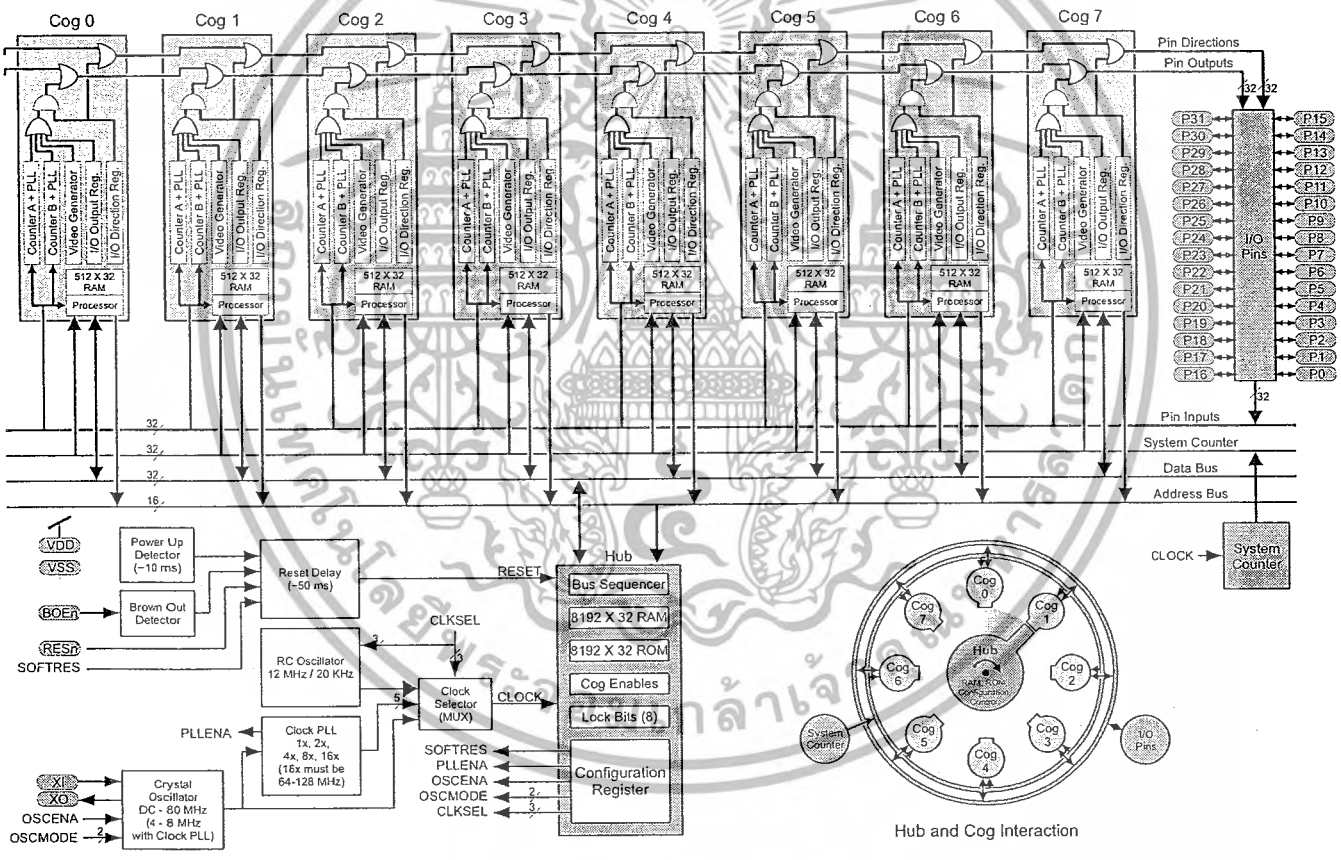
8-Cog Multiprocessor Microcontroller

1.0 PRODUCT OVERVIEW

1.1. Introduction

The Propeller chip is designed to provide high-speed processing for embedded systems while maintaining low current consumption and a small physical footprint. In addition to being fast, the Propeller chip provides flexibility and power through its eight processors, called cogs, that can perform simultaneous tasks independently or cooperatively, all while maintaining a relatively simple architecture that is easy to learn and utilize. Two programming languages are available: Spin (a high-level object-based language) and Propeller Assembly. Both include custom commands to easily manage the Propeller chip's unique features.

Figure 1: Propeller P8X32A Block Diagram



Hub and Cog Interaction

1.2. Stock Codes

Table 1: Propeller Chip Stock Codes

Device Stock #	Package Type	I/O Pins	Power Requirements	External Clock Speed	Internal RC Oscillator	Internal Execution Speed	Global ROM/RAM	Cog RAM
P8X32A-D40	40-pin DIP	32 CMOS	3.3 volts DC	DC to 80 MHz	12 MHz or 20 kHz*	0 to 160 MIPS (20 MIPS/cog)	64 K bytes; 32768 bytes ROM / 32768 bytes RAM	512 x 32 bits per cog
P8X32A-Q44	44-pin LQFP							
P8X32A-M44	44-pin QFN							

***Approximate; may range from 8 MHz – 20 MHz, or 13 kHz – 33 kHz, respectively.**

Table of Contents

1.3. Key Features

The design of the Propeller chip frees application developers from common complexities of embedded systems programming. For example:

- Eight processors (cogs) perform simultaneous processes independently or cooperatively, sharing common resources through a central hub. The Propeller application designer has full control over how and when each cog is employed; there is no compiler-driven or operating system-driven splitting of tasks among multiple cogs. This method empowers the developer to deliver absolutely deterministic timing, power consumption, and response to the embedded application.
- Asynchronous events are easier to handle than with devices that use interrupts. The Propeller has no need for interrupts; just assign some cogs to individual, high-bandwidth tasks and keep other cogs free and unencumbered. The result is a more responsive application that is easier to maintain.
- A shared System Clock allows each cog to maintain the same time reference, allowing true synchronous execution.

1.4. Programming Advantages

- The object-based high-level Spin language is easy to learn, with special commands that allow developers to quickly exploit the Propeller chip's unique and powerful features.
- Propeller Assembly instructions provide conditional execution and optional flag and result writing for each individual instruction. This makes critical, multi-decision blocks of code more consistently timed; event handlers are less prone to jitter and developers spend less time padding, or squeezing, cycles.

1.5. Applications

The Propeller chip is particularly useful in projects that can be vastly simplified with simultaneous processing, including:

- Industrial control systems
- Sensor integration, signal processing, and data acquisition
- Handheld portable human-interface terminals
- Motor and actuator control
- User interfaces requiring NTSC, PAL, or VGA output, with PS/2 keyboard and mouse input
- Low-cost video game systems
- Industrial, educational or personal-use robotics
- Wireless video transmission (NTSC or PAL)

1.6. Programming Platform Support

Parallax Inc. supports the Propeller chip with a variety of hardware tools and boards:

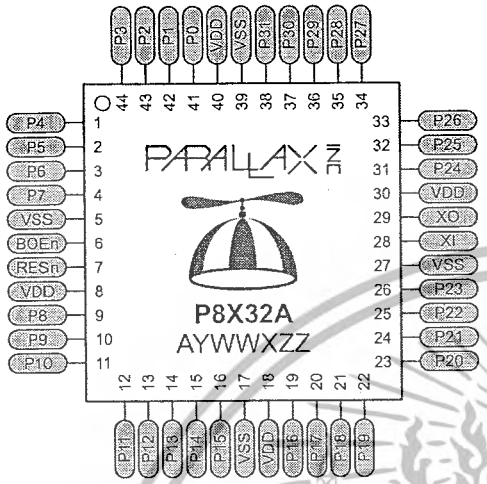
- Propeller Clip (#32200) and Propeller Plug (#32201). These boards provide convenient programming port connections, see the Typical Connection Diagrams on Page 5.
- The Propeller Demo Board (#32100) provides a convenient means to test-drive the Propeller chip's varied capabilities through a host of device interfaces on one compact board. The schematic is provided on page 24. Main features:
  - P8X32A-Q44 Propeller Chip
  - 24LC256-I/ST EEPROM for program storage
  - Replaceable 5.000 MHz crystal
  - 3.3 V and 5 V regulators with on/off switch
  - USB-to-serial interface for programming and communication
  - VGA and TV output
  - Stereo output with 16 Ω headphone amplifier
  - Electret microphone input
  - Two PS/2 mouse and keyboard I/O connectors
  - 8 LEDs (share VGA pins)
  - Pushbutton for reset
  - Big ground post for scope hookup
  - I/O pins P0-P7 are free and brought out to header
  - Breadboard for custom circuits
- The Propeller Proto Board (#32212) features a surface-mount Propeller chip with the necessary components to achieve a programming interface, with pads ready for a variety of I/O connectors and DIP/SIP chips, and a generous through-hole prototyping area.
- The PropStick USB (#32210) features a Propeller chip, EEPROM, 3.3 VDC and 5 VDC regulators, reset button, crystal and USB connection on a 0.6 wide DIP package for easy prototyping on perfboard and breadboard.

1.7. Corporate and Community Support

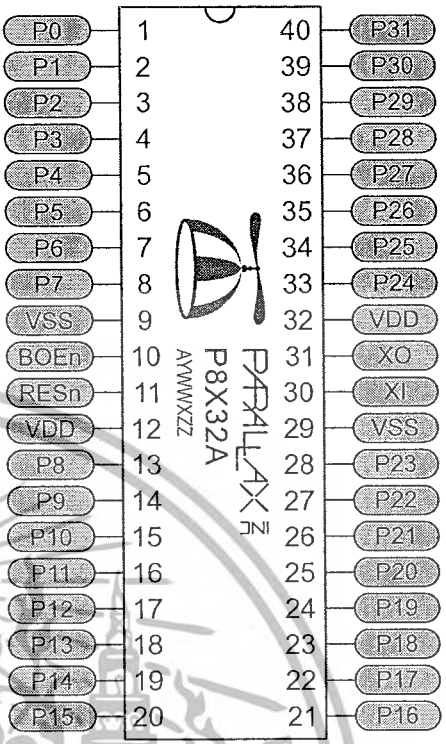
- Parallax provides technical support free of charge. In the Continental US, call toll free (888) 512-1024; from outside please call (916) 624-8333. Or, email: support@parallax.com.
- Parallax hosts a moderated public user's forum just for the Propeller: http://forums.parallax.com/forums.
- Browse through community-created Propeller objects and share yours with others via Parallax-hosted Propeller Object Exchange Library: http://obex.parallax.com.

2.0 CONNECTION DIAGRAMS

2.1. Pin Assignments



LQFP and QFN Packages



DIP Package

2.2. Pin Descriptions

Table 2: Pin Descriptions

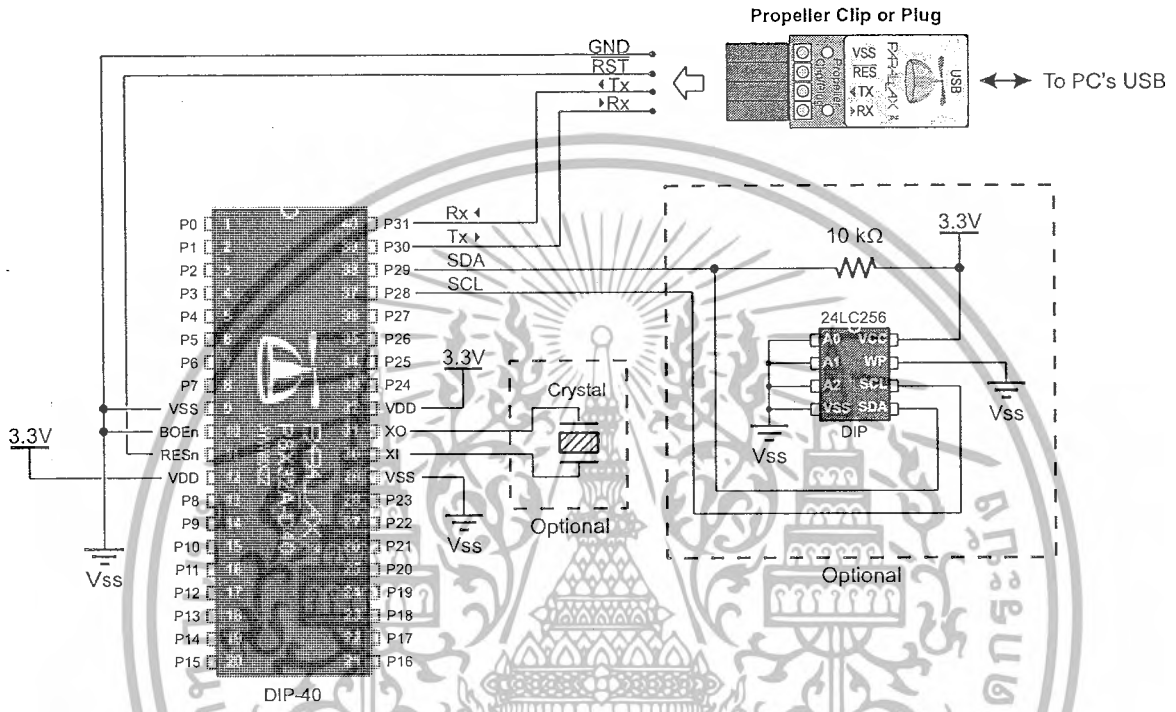
Pin Name	Direction	Description
P0 – P31	I/O	General purpose I/O Port A. Can source/sink 40 mA each at 3.3 VDC. CMOS level logic with threshold of ≈ ½ VDD or 1.6 VDC @ 3.3 VDC. The pins shown below have a special purpose upon power-up/reset but are general purpose I/O afterwards. P28 - I2C SCL connection to optional, external EEPROM. P29 - I2C SDA connection to optional, external EEPROM. P30 - Serial Tx to host. P31 - Serial Rx from host.
VDD	---	3.3 volt power (2.7 – 3.6 VDC)
VSS	---	Ground
BOEn	I	Brown Out Enable (active low). Must be connected to either VDD or VSS. If low, RESn becomes a weak output (delivering VDD through 5 kΩ) for monitoring purposes but can still be driven low to cause reset. If high, RESn is CMOS input with Schmitt Trigger.
RESn	I/O	Reset (active low). When low, resets the Propeller chip: all cogs disabled and I/O pins floating. Propeller restarts 50 ms after RESn transitions from low to high.
XI	I	Crystal Input. Can be connected to output of crystal/oscillator pack (with XO left disconnected), or to one leg of crystal (with XO connected to other leg of crystal or resonator) depending on CLK Register settings. No external resistors or capacitors are required.
XO	O	Crystal Output. Provides feedback for an external crystal, or may be left disconnected depending on CLK Register settings. No external resistors or capacitors are required.

2.3. Typical Connection Diagrams

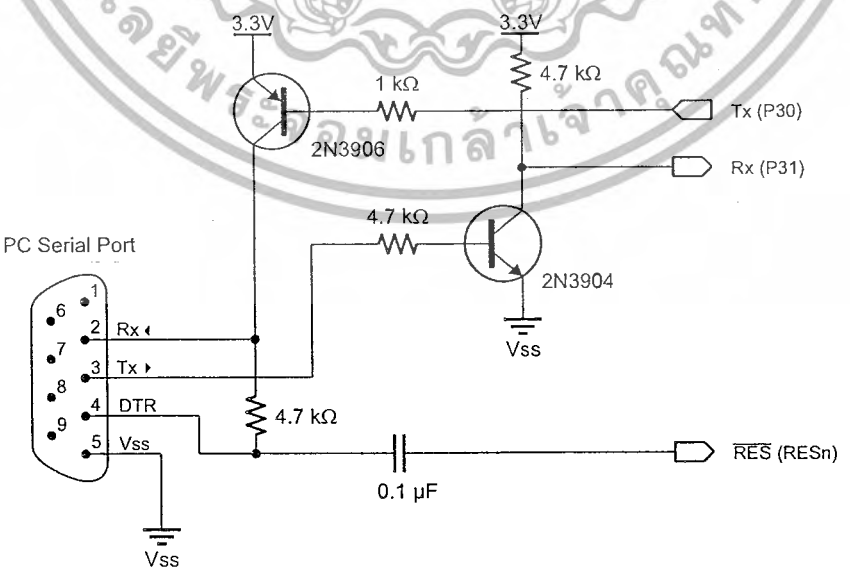
2.3.1. Propeller Clip or Propeller Plug Connection - Recommended

Note that the connections to the external oscillator and EEPROM, which are enclosed in dashed lines, are optional.

Propeller Clip, Stock #32200; Propeller Plug, Stock #32201. The Propeller Clip/Plug schematic is available for download from www.parallax.com.



2.3.2. Alternative Serial Port Connection



3.0 OPERATING PROCEDURES

3.1. Boot-Up Procedure

Upon power-up, or reset:

1. The Propeller chip's internal RC oscillator begins running at 20 kHz, then after a 50 ms reset delay, switches to 12 MHz. Then the first processor (Cog 0) loads and runs the built-in Boot Loader program.
2. The Boot Loader performs one or more of the following tasks, in order:
   a. Detects communication from a host, such as a PC, on pins P30 and P31. If communication from a host is detected, the Boot Loader converses with the host to identify the Propeller chip and possibly download a program into global RAM and optionally into an external 32 KB EEPROM.
   b. If no host communication was detected, the Boot Loader looks for an external 32 KB EEPROM on pins P28 and P29. If an EEPROM is detected, the entire 32 KB data image is loaded into the Propeller chip's global RAM.
   c. If no EEPROM was detected, the boot loader stops, Cog 0 is terminated, the Propeller chip goes into shutdown mode, and all I/O pins are set to inputs.
3. If either step 2a or 2b was successful in loading a program into the global RAM, and a suspend command was not given by the host, then Cog 0 is reloaded with the built-in Spin Interpreter and the user code is run from global RAM.

3.2. Run-Time Procedure

A Propeller Application is a user program compiled into its binary form and downloaded to the Propeller chip's RAM or external EEPROM. The application consists of code written in the Propeller chip's Spin language (high-level code) with optional Propeller Assembly language components (low-level code). Code written in the Spin language is interpreted during run time by a cog running the Spin Interpreter while code written in Propeller Assembly is run in its pure form directly by a cog. Every Propeller Application consists of at least a little Spin code and may actually be written entirely in Spin or with various amounts of Spin and assembly. The Propeller chip's Spin Interpreter is started in Step 3 of the Boot Up Procedure, above, to get the application running.

Once the boot-up procedure is complete and an application is running in Cog 0, all further activity is defined by the application itself. The application has complete control over things like the internal clock speed, I/O pin usage, configuration registers, and when, what and how many cogs are running at any given time. All of this is variable at run time, as controlled by the application.

3.3. Shutdown Procedure

When the Propeller goes into shutdown mode, the internal clock is stopped causing all cogs to halt and all I/O pins are set to input direction (high impedance). Shutdown mode is triggered by one of the three following events:

1. VDD falling below the brown-out threshold (~2.7 VDC), when the brown out circuit is enabled,
2. the RESn pin going low, or
3. the application requests a reboot (see the REBOOT command in the Propeller Manual).

Shutdown mode is discontinued when the voltage level rises above the brown-out threshold and the RESn pin is high.

4.0 SYSTEM ORGANIZATION

4.1. Shared Resources

There are two types of shared resources in the Propeller: 1) common, and 2) mutually-exclusive. Common resources can be accessed at any time by any number of cogs. Mutually-exclusive resources can also be accessed by any number of cogs, but only by one cog at a time. The common resources are the I/O pins and the System Counter. All other shared resources are mutually-exclusive by nature and access to them is controlled by the Hub. See Section 4.4 on page 7.

4.2. System Clock

The System Clock (shown as "CLOCK" in Figure 1, page 1) is the central clock source for nearly every component of the Propeller chip. The System Clock's signal comes from one of three possible sources:

- The internal RC oscillator (~12 MHz or ~20 kHz)
- The XI input pin (either functioning as a high-impedance input or a crystal oscillator in conjunction with the XO pin)
- The Clock PLL (phase-locked loop) fed by the XI input

The source is determined by the CLK register's settings, which is selectable at compile time and reselectable at run time. The Hub and internal Bus operate at half the System Clock speed.

4.3. Cogs (processors)

The Propeller contains eight (8) identical, independent processors, called cogs, numbered 0 to 7. Each cog contains a Processor block, local 2 KB RAM configured as 512 longs (512 x 32 bits), two advanced counter modules with PLLs, a Video Generator, I/O Output Register, I/O Direction Register, and other registers not shown in the Block Diagram.

All eight cogs are driven from the System Clock; they each maintain the same time reference and all active cogs execute instructions simultaneously. They also all have access to the same shared resources.

Cogs can be started and stopped at run time and can be programmed to perform tasks simultaneously, either independently or with coordination from other cogs through Main RAM. Each cog has its own RAM, called Cog RAM, which contains 512 registers of 32 bits each. The Cog RAM is all general purpose RAM except for the last 16 registers, which are special purpose registers, as described in Table 15 on page 15.

4.4. Hub

To maintain system integrity, mutually-exclusive resources must not be accessed by more than one cog at a time. The Hub controls access to mutually-exclusive resources by giving each cog a turn in a "round robin" fashion from Cog 0 through Cog 7 and back to Cog 0 again. The Hub and its bus run at half the System Clock rate, giving a cog access to mutually-exclusive resources once every 16 System Clock cycles. Hub instructions, the Propeller Assembly instructions that access mutually-exclusive resources, require 7 cycles to execute but they first need to be synchronized to the start of the Hub Access Window.

It takes up to 15 cycles (16 minus 1, if we just missed it) to synchronize to the Hub Access Window plus 7 cycles to execute the hub instruction, so hub instructions take from 7 to 22 cycles to complete.

Figure 2 and Figure 3 show examples where Cog 0 has a hub instruction to execute. Figure 2 shows the best-case scenario; the hub instruction was ready right at the start of that cog's access window. The hub instruction executes immediately (7 cycles) leaving an additional 9 cycles for other instructions before the next Hub Access Window arrives.

Figure 3 shows the worst-case scenario; the hub instruction was ready on the cycle right after the start of Cog 0 s access window; it just barely missed it. The cog waits until the next Hub Access Window (15 cycles later) then the hub instruction executes (7 cycles) for a total of 22 cycles for that hub instruction. Again, there are 9 additional cycles after the hub instruction for other instructions to execute before the next Hub Access Window arrives. To get the most efficiency out of Propeller Assembly routines that have to frequently access mutually-exclusive resources, it can be beneficial to interleave non-hub instructions with hub instructions to lessen the number of cycles waiting for the next Hub Access Window. Since most Propeller Assembly instructions take 4 clock cycles, two such instructions can be executed in between otherwise contiguous hub instructions.

Keep in mind that a particular cog's hub instructions do not, in any way, interfere with other cogs' instructions because of the Hub mechanism. Cog 1, for example, may start a hub instruction during System Clock cycle 2, in both of these examples, possibly overlapping its execution with that of Cog 0 without any ill effects. Meanwhile, all other cogs can continue executing non-hub instructions, or awaiting their individual hub access windows regardless of what the others are doing.

Figure 2: Cog-Hub Interaction – Best Case Scenario

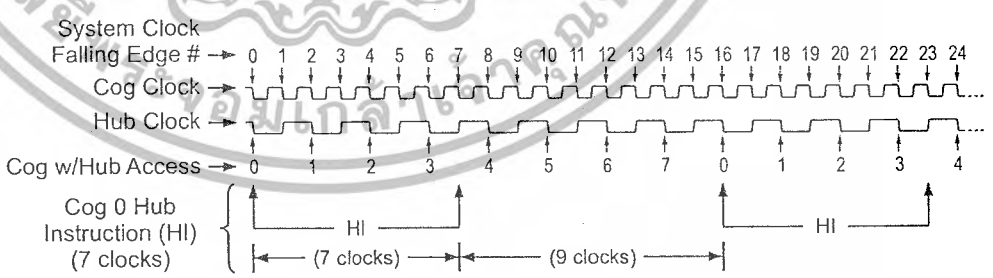
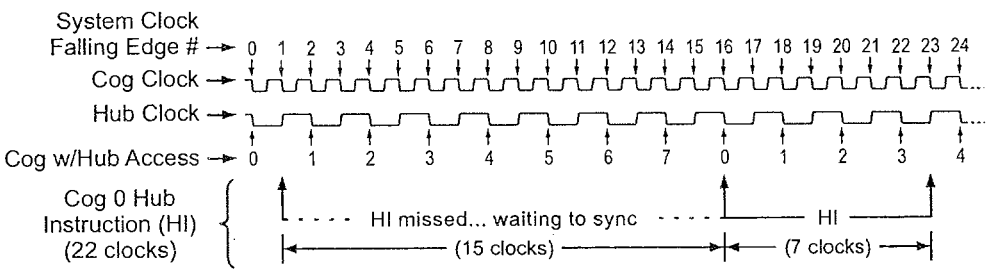


Figure 3: Cog-Hub Interaction – Worst Case Scenario



4.5. I/O Pins

The Propeller has 32 I/O pins, 28 of which are general purpose. I/O Pins 28 - 31 have a special purpose at boot up and are available for general purpose use afterwards; see section 2.2, page 4. After boot up, any I/O pins can be used by any cogs at any time. It is up to the application developer to ensure that no two cogs try to use the same I/O pin for different purposes during run time.

Each cog has its own 32-bit I/O Direction Register and 32-bit I/O Output Register. The state of each cog's Direction Register is OR'd with that of the previous cogs' Direction Registers, and each cog's output states is OR'd with that of the previous cogs' output states. Note that each cog's output states are made up of the OR'd states of its internal I/O hardware and that is all AND'd with its Direction Register's states. The result is that each I/O pin's direction and output state is the "wired-OR" of the entire cog collective. No electrical contention between cogs is possible, yet they can all still access the I/O pins simultaneously. The result of this I/O pin wiring configuration can be described in the following rules:

A. A pin is an input only if no active cog sets it to an output.
B. A pin outputs low only if all active cogs that set it to output also set it to low.
C. A pin outputs high if any active cog sets it to an output and also sets it high.

Table 3 demonstrates a few possible combinations of the collective cogs' influence on a particular I/O pin, P12 in this example. For simplification, these examples assume that bit 12 of each cog's I/O hardware, other than its I/O Output Register, is cleared to zero (0).

Any cog that is shut down has its Direction Register and output states cleared to zero, effectively removing it from influencing the final state of the I/O pins that the remaining active cogs are controlling.

Each cog also has its own 32-bit Input Register. This input register is really a pseudo-register; every time it is read, the actual states of the I/O pins are read, regardless of their input or output direction.

Table 3: I/O Sharing Examples

	Bit 12 of Cogs' I/O Direction Register								Bit 12 of Cogs' I/O Output Register								State of I/O Pin P12	Rule Followed
Cog ID	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7		
Example 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Input	A
Example 2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Output Low	B
Example 3	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	Output High	C
Example 4	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	Output Low	B
Example 5	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	Output High	C
Example 6	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0	Output High	C
Example 7	1	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	Output High	C
Example 8	1	1	1	0	1	1	1	1	0	0	0	1	0	0	0	0	Output Low	B

Note: For the I/O Direction Register, a 1 in a bit location sets the corresponding I/O pin to the output direction; a 0 sets it to an input direction.

4.6. System Counter

The System Counter is a global, read-only, 32-bit counter that increments once every System Clock cycle. Cogs can read the System Counter (via their CNT registers, see Table 15 on page 15) to perform timing calculations and can use the WAITCNT command (see section 6.3 on page 18 and section 6.4 on page 21) to create effective delays within their processes. The System Counter is a common resource which every cog can read simultaneously. The System Counter is not cleared upon startup since its practical use is for differential timing. If a cog needs to keep track of time from a specific, fixed moment in time, it simply needs to read and save the initial counter value at that moment in time, and compare subsequent counter values against that initial value.

4.7. Locks

There are eight lock bits (semaphores) available to facilitate exclusive access to user-defined resources among multiple cogs. If a block of memory is to be used by two or more cogs at once and that block consists of more than one long (four bytes), the cogs will each have to perform multiple reads and writes to retrieve or update that memory block. This leads to the likely possibility of read/write contention on that memory block where one cog may be writing while another is reading, resulting in misreads and/or miswrites.

The locks are global bits accessed through the Hub via LOCKNEW, LOCKRET, LOCKSET, and LOCKCLR. Because locks are accessed only through the Hub, only one cog at a time can affect them, making this an effective control mechanism. The Hub maintains an inventory of which locks are in use and their current states; cogs can check out, return, set, and clear locks as needed during run time.

4.8. Cog Counters

Each cog has two counter modules: CTRA and CTRB. Each counter module can control or monitor up to two I/O pins and perform conditional 32-bit accumulation of its FRQ register into its PHS register on every clock cycle.

Each counter module also has its own phase-locked loop (PLL) which can be used to synthesize frequencies up to 128 MHz.

With a little setup or oversight from the cog, a counter can be used for:

- frequency synthesis
- frequency measurement
- pulse counting
- pulse measurement
- multi-pin state measurement
- pulse-width modulation
- duty-cycle measurement
- digital-to-analog conversion
- analog-to-digital conversion

For some of these operations, the cog can be set up and left in a free-running mode. For others, it may use WAITCNT to time-align counter reads and writes within a loop, creating the effect of a more complex state machine.

Note that for a cog clock frequency of 80 MHz, the counter update period is a mere 12.5 ns. This high speed, combined with 32-bit precision, allows for very dynamic signal generation and measurement.

The design goal for the counter was to create a simple and flexible subsystem which could perform some repetitive task on every clock cycle, thereby freeing the cog to perform some computationally richer super-task. While the counters have only 32 basic operating modes, there is no limit to how they might be used dynamically through software. Integral to this concept is the use of the WAITPEQ, WAITPNE, and WAITCNT instructions, which can event-align or time-align a cog with its counters.

Each counter has three registers:

4.8.1. CTRA / CTRB – Control register

The CTR (CTRA and CTRB) register selects the counter's operating mode. As soon as this register is written, the new operating mode goes into effect. Writing a zero to CTR will immediately disable the counter, stopping all pin output and PHS accumulation.

Table 4: CTRA and CTRB Registers

31	30..26	25..23	22..15	14..9	8..6	5..0
-	CTRMODE	PLLDIV	-	BPIN	-	APIN

The CTRMODE field selects one of 32 operating modes for the counter, conveniently written (along with PLLDIV) using the MOVI instruction. These modes of operation are listed in Table 6 on page 10.

Table 5: PLLDIV Field

PLLDIV	%000	%001	%010	%011	%100	%101	%110	%111
Output	VCO / 128	VCO / 64	VCO / 32	VCO / 16	VCO / 8	VCO / 4	VCO / 2	VCO / 1

PLLDIV selects a PLL output tap and may be ignored if not used.

The PLL modes (%00001 to %00011) cause FRQ-to-PHS accumulation to occur every clock cycle. This creates a numerically-controlled oscillator (NCO) in PHS[31], which feeds the counter PLL's reference input. The PLL will multiply this frequency by 16 using its voltage-controlled oscillator (VCO). For stable operation, it is recommended that the VCO frequency be kept within 64 MHz to 128 MHz. This translates to an NCO frequency of 4 MHz to 8 MHz.

The PLLDIV field of the CTR register selects which power-of-two division of the VCO frequency will be used as the final PLL output. This affords a PLL range of 500 kHz to 128 MHz.

BPIN selects a pin to be the secondary I/O. It may be ignored if not used and may be written using the MOVD instruction.

APIN selects a pin to be the primary I/O. It may be ignored if not used and may be written using the MOVS instruction.

4.8.2. FRQA / FRQB – Frequency register

FRQ (FRQA and FRQB) holds the value that will be accumulated into the PHS register. For some applications, FRQ may be written once, and then ignored. For others, it may be rapidly modulated.

4.8.3. PHSA / PHSB – Phase register

The PHS (PHSA and PHSB) register can be written and read via cog instructions, but it also functions as a free-running accumulator, summing the FRQ register into itself on potentially every clock cycle. Any instruction writing to PHS will override any accumulation for that clock cycle. PHS can only be read through the source operand (same as PAR, CNT, INA, and INB). Beware that doing a read-modify-write instruction on PHS, like "ADD PHSA, #1", will cause the last-written value to be used as the destination operand input, rather than the current accumulation.

Table 6: Counter Modes (CTRMODE Field Values)				
CTRMODE	Description	Accumulate FRQx to PHSx	APIN Output*	BPIN Output*
%00000	Counter disabled (off)	0 (never)	0 (none)	0 (none)
%00001 %00010 %00011	PLL internal (video mode) PLL single-ended PLL differential	1 (always) 1 1	0 PLLx PLLx	0 0 !PLLx
%00100 %00101	NCO single-ended NCO differential	1 1	PHSx[31] PHSx[31]	0 !PHSx[31]
%00110 %00111	DUTY single-ended DUTY differential	1 1	PHSx-Carry PHSx-Carry	0 !PHSx-Carry
%01000 %01001 %01010 %01011	POS detector POS detector with feedback POSEDGE detector POSEDGE detector w/ feedback	A^1 A^1 A^1 & $!A^2$ A^1 & $!A^2$	0 0 0 0	0 !A1 0 !A1
%01100 %01101 %01110 %01111	NEG detector NEG detector with feedback NEGEDGE detector NEGEDGE detector w/ feedback	$!A^1$ $!A^1$ $!A^1$ & A^2 $!A^1$ & A^2	0 0 0 0	0 !A1 0 !A1
%10000 %10001 %10010 %10011 %10100 %10101 %10110 %10111 %11000 %11001 %11010 %11011 %11100 %11101 %11110 %11111	LOGIC never LOGIC !A & !B LOGIC A & !B LOGIC !B LOGIC !A & B LOGIC !A LOGIC A <> B LOGIC !A | !B LOGIC A & B LOGIC A == B LOGIC A LOGIC A | !B LOGIC B LOGIC !A | B LOGIC A | B LOGIC always	0 $!A^1$ & $!B^1$ A^1 & $!B^1$ $!B^1$ $!A^1$ & B^1 $!A^1$ A^1 <> B^1 $!A^1$ | $!B^1$ A^1 & B^1 A^1 == B^1 A^1 A^1 | $!B^1$ B^1 $!A^1$ | B^1 A^1 | B^1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

*Must set corresponding DIR bit to affect pin. A1 = APIN input delayed by 1 clock. A2 = APIN input delayed by 2 clocks. B1 = BPIN input delayed by 1 clock.

4.9. Video Generator

Each cog has a video generator module that facilitates transmitting video image data at a constant rate. There are two registers and one instruction which provide control and access to the video generator. Counter A of the cog must be running in a PLL mode and is used to generate the timing signal for the Video Generator. The Video Scale Register specifies the number of Counter A PLL (PLLA) clock cycles for each pixel and number of clock cycles before fetching another frame of data provided by the WAITVID instruction which is executed within the cog. The Video Configuration Register establishes the mode the Video Generator should operate, and can generate VGA or composite video (NTSC or PAL).

The Video Generator should be initialized by first starting Counter A, setting the Video Scale Register, setting the Video Configuration Register, then finally providing data via the WAITVID instruction. Failure to properly initialize the Video Generator by first starting PLLA will cause the cog to indefinitely hang when the WAITVID instruction is executed.

4.9.1. VCFG – Video Configuration Register

The Video Configuration Register contains the configuration settings of the video generator and is shown in Table 7.

In Propeller Assembly, the VMode through AuralSub fields can conveniently be written using the MOVI instruction, the VGroup field can be written with the MOVD instruction, and the VPins field can be written with the MOVS instruction.

Table 7: VCFG Register									
31	30..29	28	27	26	25..23	22..12	11..9	8	7..0
-	VMode	CMode	Chroma1	Chroma0	AuralSub	-	VGroup	-	VPins

The 2-bit VMode (video mode) field selects the type and orientation of video output, if any, according to Table 8.

Table 8: The Video Mode Field

VMode	Video Mode
00	Disabled, no video generated.
01	VGA mode; 8-bit parallel output on VPins 7:0
10	Composite Mode 1; broadcast on VPins 7:4, baseband on VPins 3:0
11	Composite Mode 2; baseband on VPins 7:4, broadcast on VPins 3:0

The CMode (color mode) field selects two or four color mode. 0 = two-color mode; pixel data is 32 bits by 1 bit and only colors 0 or 1 are used. 1 = four-color mode; pixel data is 16 bits by 2 bits, and colors 0 through 3 are used.

The Chroma1 (broadcast chroma) bit enables or disables chroma (color) on the broadcast signal. 0 = disabled, 1 = enabled.

The Chroma0 (baseband chroma) bit enables or disables chroma (color) on the baseband signal. 0 = disabled, 1 = enabled.

The AuralSub (aural sub-carrier) field selects the source of the FM aural (audio) sub-carrier frequency to be modulated on. The source is the PLLA of one of the cogs, identified by AuralSub's value. This audio must already be modulated onto the 4.5 MHz sub-carrier by the source PLLA.

Table 9: The AuralSub Field

AuralSub	Sub-Carrier Frequency Source
000	Cog 0 s PLLA
001	Cog 1 s PLLA
010	Cog 2 s PLLA
011	Cog 3 s PLLA
100	Cog 4 s PLLA
101	Cog 5 s PLLA
110	Cog 6 s PLLA
111	Cog 7 s PLLA

The VGroup (video output pin group) field selects which group of 8 I/O pins to output video on.

Table 10: The VGroup Field

VGroup	Pin Group
000	Group 0: P7..P0
001	Group 1: P15..P8
010	Group 2: P23..P16
011	Group 3: P31..P24
100-111	<reserved for future use>

The VPins (video output pins) field is a mask applied to the pins of VGroup that indicates which pins to output video signals on.

Table 11: The VPins Field

VPins	Effect
00001111	Drive Video on lower 4 pins only; composite
11110000	Drive Video on upper 4 pins only; composite
11111111	Drive video on all 8 pins; VGA
XXXXXXXX	Any value is valid for this field; the above values are the most common.

4.9.2. VSCL – Video Scale Register

The Video Scale Register sets the rate at which video data is generated, and is shown in Table 12.

Table 12: VSCL Register

VSCL Bits		
31..20	19..12	11..0
–	PixelClocks	FrameClocks

The 8-bit PixelClocks field indicates the number of clocks per pixel; the number of clocks that should elapse before each pixel is shifted out by the video generator module. These clocks are the PLLA clocks, not the System Clock. A value of 0 for this field is interpreted as 256.

The 12-bit FrameClocks field indicates the number of clocks per frame; the number of clocks that will elapse before each frame is shifted out by the video generator module. These clocks are the PLLA clocks, not the System Clock. A frame is one long of pixel data (delivered via the WAITVID command). Since the pixel data is either 16 bits by 2 bits, or 32 bits by 1 bit (meaning 16 pixels wide with 4 colors, or 32 pixels wide with 2 colors, respectively), the FrameClocks is typically 16 or 32 times that of the PixelClocks value. A value of 0 for this field is interpreted as 4096.

4.9.3. WAITVID Command/Instruction

The WAITVID instruction is the delivery mechanism for data to the cog's Video Generator hardware. Since the Video Generator works independently from the cog itself, the two must synchronize each time data is needed for the display device. The frequency at which this occurs is dictated by the frequency of PLLA and the Video Scale Register. The cog must have new data available before the moment the Video Generator needs it. The cog uses WAITVID to wait for the right time and then "hand off" this data to the Video Generator.

Two longs of data are passed to the Video Generator by with the syntax WAITVID *Colors, Pixels*.

The *Colors* parameter is a 32-bit value containing either four 8-bit color values (for 4 color mode) or two 8-bit color values in the lower 16 bits (for 2 color mode). For

VGA mode, each 8-bit color value is written to the pins specified by the VGroup and VPins field. For VGA typically the 8 bits are grouped into 2 bits per primary color and Horizontal and Vertical Sync control lines, but this is up to the software and application of how these bits are used. For composite video each 8-bit color value is composed of 3 fields. Bits 0-2 are the luminance value of the generated signal. Bit 3 is the modulation bit which dictates whether the chroma information will be generated and bits 4-7 indicate the phase angle of the chroma value. When the modulation bit is set to 0, the chroma information is ignored and only the luminance value is output to pins. When the modulation bit is set to 1 the luminance value is modulated ± 1 with a phase angle set by bits 4-7. In order to achieve the full resolution of the chroma value, PLLA should be set to 16 times the modulation frequency (in composite video this is called the color-burst frequency). The PLLB of the cog is used to generate the broadcast frequency; whether this is generated depends on if PLLB is running and the values of VMode and VPins.

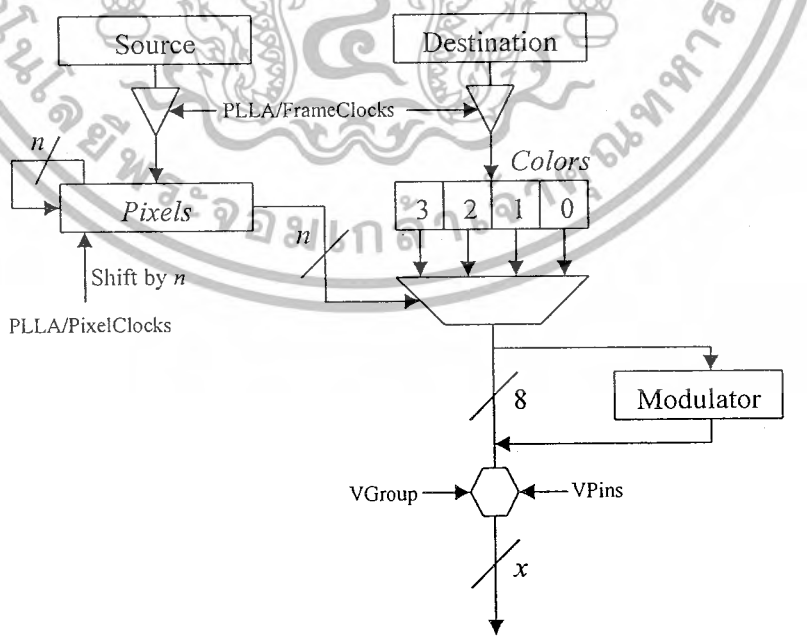
The *Pixels* parameter describes the pixel pattern to display, either 16 pixels or 32 pixels depending on the color depth configuration of the Video Generator. When four-color mode is specified, *Pixels* is a 16x2 bit pattern where each 2-bit pixel is an index into *Colors* on which data pattern should be presented to the pins. When two-color mode is specified, *Pixels* is a 32x1 bit pattern where each bit specifies which of the two color patterns in the lower 16 bits of *Colors* should be output to the pins. The Pixel data is shifted out least significant bits (LSB) first.

When the FrameClocks value is greater than 16 times the PixelClocks value and 4-color mode is specified, the two most significant bits are repeated until FrameClocks PLLA cycles have occurred. When FrameClocks value is greater than 32 times PixelClocks value and 2-color mode is specified, the most significant bit is repeated until FrameClocks PLLA cycles have occurred. When FrameClocks cycles occur and the cog is not in a `WAITVID` instruction, whatever data is on the source and destination busses at the time will be fetched and used. So it is important to be in a `WAITVID` instruction before this occurs.

While the Video Generator was created to display video signals, its potential applications are much more diverse. The Composite Video mode can be used to generate phase-shift keying communications of a granularity of 16 or less and the VGA mode can be used to generate any bit pattern with a fully settable and predictable rate.

Figure 4 is a block diagram of how the VGA mode is organized. The two inverted triangles are the load mechanism for *Pixels* and *Colors; n* is 1 or 2 bits depending on the value of CMode. The inverted trapezoid is a 4-way 8-bit multiplexer that chooses which byte of *Colors* to output. When in composite video mode the Modulator transforms the byte into the luminance and chroma signal and outputs the broadcast signal. VGroup steers the 8 bits to a block of output pins and outputs to those pins which are set to 1 in VPins; this combined functionality is represented by the hexagon.

Figure 4: Video Generator



4.10. CLK Register

The CLK register is the System Clock configuration control; it determines the source and characteristics of the System Clock. It configures the RC Oscillator, Clock PLL, Crystal Oscillator, and Clock Selector circuits (See the Block Diagram, page 1). It is configured at compile time by the _CLKMODE declaration and is writable at run time through the CLKSET command. Whenever the CLK register is written, a global delay of ~75 µs occurs as the clock source transitions.

Whenever this register is changed, a copy of the value written should be placed in the Clock Mode value location (which is BYTE[4] in Main RAM) and the resulting master clock frequency should be written to the Clock Frequency value location (which is LONG[0] in Main RAM) so that objects which reference this data will have current information for their timing calculations.

Use Spin's CLKSET command when possible (see sections 6.3 and 6.4) since it automatically updates all the above-mentioned locations with the proper information.

Table 13: Valid Clock Modes

Valid Expression	CLK Reg. Value	Valid Expression	CLK Reg. Value
RCFAST	0_0_0_00_000	XTAL1 + PLL1X	0_1_1_01_011
RCSLOW	0_0_0_00_001	XTAL1 + PLL2X	0_1_1_01_100
XINPUT	0_0_1_00_010	XTAL1 + PLL4X	0_1_1_01_101
		XTAL1 + PLL8X	0_1_1_01_110
		XTAL1 + PLL16X	0_1_1_01_111
XTAL1	0_0_1_01_010	XTAL2 + PLL1X	0_1_1_10_011
XTAL2	0_0_1_10_010	XTAL2 + PLL2X	0_1_1_10_100
XTAL3	0_0_1_11_010	XTAL2 + PLL4X	0_1_1_10_101
		XTAL2 + PLL8X	0_1_1_10_110
		XTAL2 + PLL16X	0_1_1_10_111
XINPUT + PLL1X	0_1_1_00_011	XTAL3 + PLL1X	0_1_1_11_011
XINPUT + PLL2X	0_1_1_00_100	XTAL3 + PLL2X	0_1_1_11_100
XINPUT + PLL4X	0_1_1_00_101	XTAL3 + PLL4X	0_1_1_11_101
XINPUT + PLL8X	0_1_1_00_110	XTAL3 + PLL8X	0_1_1_11_110
XINPUT + PLL16X	0_1_1_00_111	XTAL3 + PLL16X	0_1_1_11_111

Table 14: CLK Register Fields

Bit	7	6	5	4	3	2	1	0
Name	RESET	PLLENA	OSCENA	OSCM1	OSCM2	CLKSEL2	CLKSEL1	CLKSEL0

RESET	Effect
0	Always write '0 here unless you intend to reset the chip.
1	Same as a hardware reset – reboots the chip.

PLLENA	Effect
0	Disables the PLL circuit.
1	Enables the PLL circuit. The PLL internally multiplies the XIN pin frequency by 16. OSCENA must be '1 to propagate the XIN signal to the PLL. The PLL's internal frequency must be kept within 64 MHz to 128 MHz – this translates to an XIN frequency range of 4 MHz to 8 MHz. Allow 100 µs for the PLL to stabilize before switching to one of its outputs via the CLKSEL bits. Once the OSC and PLL circuits are enabled and stabilized, you can switch freely among all clock sources by changing the CLKSEL bits.

OSCENA	Effect
0	Disables the OSC circuit
1	Enables the OSC circuit so that a clock signal can be input to XIN, or so that XIN and XOUT can function together as a feedback oscillator. The OSCM bits select the operating mode of the OSC circuit. Note that no external resistors or capacitors are required for crystals and resonators. Allow a crystal or resonator 10 ms to stabilize before switching to an OSC or PLL output via the CLKSEL bits. When enabling the OSC circuit, the PLL may be enabled at the same time so that they can share the stabilization period.

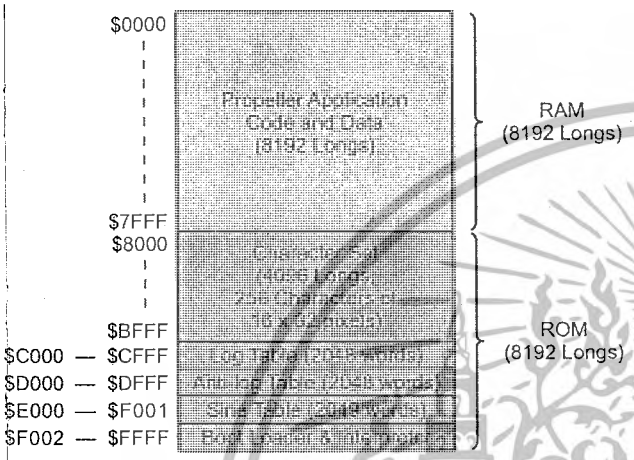
OSCM1	OSCM2	XOUT Resistance	XIN and XOUT Capacitance	Frequency Range
0	0	Infinite	6 pF (pad only)	DC to 80 MHz Input
0	1	2000 Ω	36 pF	4 MHz to 16 MHz Crystal/Resonator
1	0	1000 Ω	26 pF	8 MHz to 32 MHz Crystal/Resonator
1	1	500 Ω	16 pF	20 MHz to 60 MHz Crystal/Resonator

CLKSEL2	CLKSEL1	CLKSEL0	Master Clock	Source	Notes
0	0	0	~12 MHz	Internal	No external parts (8 to 20 MHz)
0	0	1	~20 kHz	Internal	No external parts, very low power (13-33 kHz)
0	1	0	XIN	OSC	OSCENA must be '1
0	1	1	XIN × 1	OSC+PLL	OSCENA and PLLENA must be '1
1	0	0	XIN × 2	OSC+PLL	OSCENA and PLLENA must be '1
1	0	1	XIN × 4	OSC+PLL	OSCENA and PLLENA must be '1
1	1	0	XIN × 8	OSC+PLL	OSCENA and PLLENA must be '1
1	1	1	XIN × 16	OSC+PLL	OSCENA and PLLENA must be '1

5.0 MEMORY ORGANIZATION

5.1. Main Memory

The Main Memory is a block of 64 K bytes (16 K longs) that is accessible by all cogs as a mutually-exclusive resource through the Hub. It consists of 32 KB of RAM and 32 KB of ROM. Main memory is byte, word and long addressable.



5.1.1. Main RAM

The 32 KB of Main RAM is general purpose and is the destination of a Propeller Application either downloaded from a host or from the external 32 KB EEPROM.

5.1.2. Main ROM

The 32 KB of Main ROM contains all the code and data resources vital to the Propeller chip's function: character definitions, log, anti-log and sine tables, and the Boot Loader and Spin Interpreter.

5.1.3. Character Definitions

The first half of ROM is dedicated to a set of 256 character definitions. Each character definition is 16 pixels wide by 32 pixels tall. These character definitions can be used for video generation, graphical LCD's, printing, etc.

The character set is based on a North American / Western European layout, with many specialized characters added and inserted. There are connecting waveform and schematic building-block characters, Greek characters commonly used in electronics, and several arrows and bullets. (A corresponding Parallax True-Type Font is installed with and used by the Propeller Tool software, and is available to other Windows applications.)

The character definitions are numbered 0 to 255 from left-to-right, then top-to-bottom, per Figure 5 below. They are arranged as follows: Each pair of adjacent even-odd characters is merged together to form 32 longs. The first character pair is located in $8000-$807F. The second pair occupies $8080-$80FF, and so on, until the last pair fills $BF80-$BFFF.

Figure 5: Propeller Font Character Set

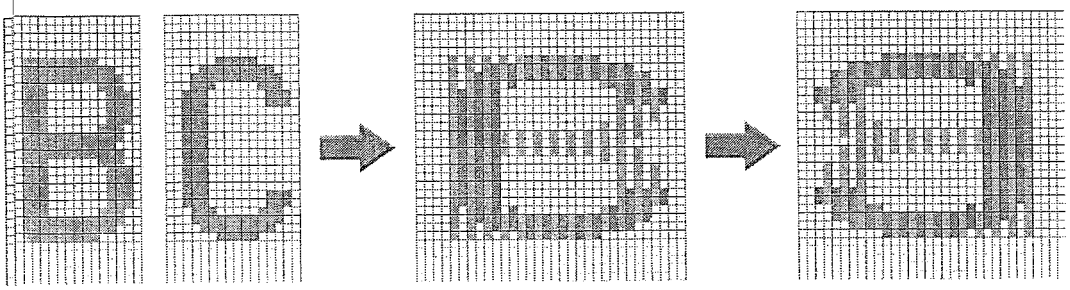


Figure 6

Propeller Character Interleaving

As shown in Figure 6, The character pairs are merged row-by-row such that each character's 16 horizontal pixels are spaced apart and interleaved with their neighbors' so that the even character takes bits 0, 2, 4, ...30, and the odd character takes bits 1, 3, 5, ...31. The leftmost pixels are in the lowest bits, while the rightmost pixels are in the highest bits. This forms a long for each row of pixels in the character pair. 32 such longs, building from top row down to bottom, make up the complete merged-pair definition. The definitions are encoded in this manner so that a cog's video hardware can handle the merged longs directly, using color selection to display either the even or the odd character.

Some character codes have inescapable meanings, such as 9 for Tab, 10 for Line Feed, and 13 for Carriage Return. These character codes invoke actions and do not equate to static character definitions. For this reason, their character definitions have been used for special four-color characters. These four-color characters are used for drawing 3-D box edges at run-time and are implemented as 16 x 16 pixel cells, as opposed to the normal 16 x 32 pixel cells. They occupy even-odd character pairs 0-1, 8-9, 10-11, and 12-13.

5.1.4. Math Function Tables

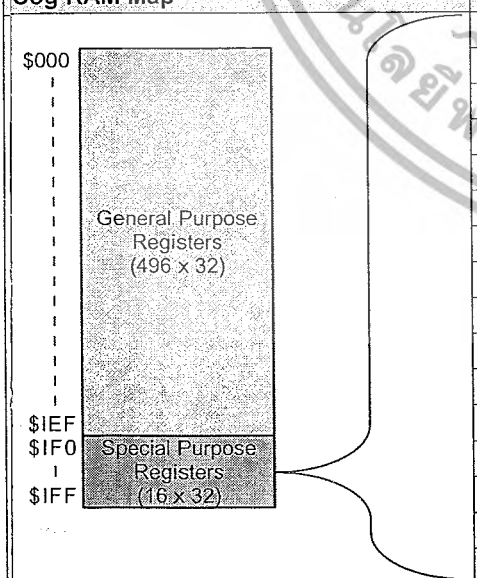
Base-2 Log and Anti-Log tables, each with 2048 unsigned words, facilitate converting values to and from exponent form to facilitate some operations; see the Propeller Manual for access instructions. Also, a sine table provides 2049 unsigned 16-bit sine samples spanning 0° to 90° inclusively (0.0439° resolution).

5.2. Cog RAM

As stated in Section 4.3, the Cog RAM is used for executable code, data, and variables, and the last 16 locations serve as interfaces to the System Counter, I/O pins, and local cog peripherals (see Table 15).

When a cog is booted up, locations 0 ($000) through 495 ($1EF) are loaded sequentially from Main RAM / ROM and its special purpose locations, 496 ($1F0) through 511 ($1FF), are cleared to zero. Each Special Purpose register may be accessed via its physical address, its predefined name, or indirectly in Spin via a register array variable SPR with an index of 0 to 15, the last four bits of the register's address.

Table 15: Cog RAM Special Purpose Registers



Address	Name	Type	Description
$1F0	PAR	Read-Only[1]	Boot Parameter
$1F1	CNT	Read-Only[1]	System Counter
$1F2	INA	Read-Only[1]	Input States for P31 - P0
$1F3	INB	Read-Only[1]	Input States for P63- P32[3]
$1F4	OUTA	Read/Write	Output States for P31 - P0
$1F5	OUTB	Read/Write	Output States for P63 – P32[3]
$1F6	DIRA	Read/Write	Direction States for P31 - P0
$1F7	DIRB	Read/Write	Direction States for P63 - P32[3]
$1F8	CTRA	Read/Write	Counter A Control
$1F9	CTRB	Read/Write	Counter B Control
$1FA	FRQA	Read/Write	Counter A Frequency
$1FB	FRQB	Read/Write	Counter B Frequency
$1FC	PHSA	Read/Write[2]	Counter A Phase:
$1FD	PHSB	Read/Write[2]	Counter B Phase
$1FE	VCFG	Read/Write	Video Configuration
$1FF	VSCL	Read/Write	Video Scale

Note 1: Only accessible as a source register (i.e. MOV *Dest, Source*).
Note 2: Only readable as a Source Register (i.e. MOV *Dest, Source*); read-modify-write not possible as a Destination Register.
Note 3: Reserved for future use.

6.0 PROGRAMMING LANGUAGES

The Propeller chip is programmed using two languages designed specifically for it: 1) Spin, a high-level object-based language, and 2) Propeller Assembly, a low-level, highly-optimized assembly language. There are many hardware-based commands in Propeller Assembly that have direct equivalents in the Spin language.

The Spin language is compiled by the Propeller Tool software into tokens that are interpreted at run time by the Propeller chip's built-in Spin Interpreter. The Propeller Assembly language is assembled into pure machine code by the Propeller Tool and is executed in its pure form at run time.

Propeller Objects can be written entirely in Spin or can use various combinations of Spin and Propeller Assembly. It is often advantageous to write objects almost entirely in Propeller Assembly, but at least two lines of Spin code are required to launch the final application.

6.1. Reserved Word List

All words listed are always reserved, whether programming in Spin or in Propeller Assembly. **As of Propeller Tool v1.05:**

Table 16: Reserved Word List

_CLKFREQ[s]	COGINIT[d]	IF_C_AND_NZ[a]	LOCKNEW[d]	NOP[a]	REPEAT[s]	TRUE[d]
_CLKMODE[s]	COGNEW[s]	IF_C_AND_Z[a]	LOCKRET[d]	NOT[s]	RES[a]	TRUNC[s]
_FREE[s]	COGSTOP[d]	IF_C_EQ_Z[a]	LOCKSET[d]	NR[a]	RESULT[s]	UNTIL[s]
_STACK[s]	CON[s]	IF_C_NE_Z[a]	LONG[s]	OBJ[s]	RET[a]	VAR[s]
_XINFREQ[s]	CONSTANT[s]	IF_C_OR_NZ[a]	LONGFILL[s]	ONES[a#]	RETURN[s]	VCFG[d]
ABORT[s]	CTRA[d]	IF_C_OR_Z[a]	LONGMOVE[s]	OR[d]	REV[a]	VSCL[d]
ABS[a]	CTRB[d]	IF_E[a]	LOOKDOWN[s]	ORG[a]	ROL[a]	WAITCNT[d]
ABSNEG[a]	DAT[s]	IF_NC[a]	LOOKDOWNZ[s]	OTHER[s]	ROR[a]	WAITPEQ[d]
ADD[a]	DIRA[d]	IF_NC_AND_NZ[a]	LOOKUP[s]	OUTA[d]	ROUND[s]	WAITPNE[d]
ADDABS[a]	DIRB[d#]	IF_NC_AND_Z[a]	LOOKUPZ[s]	OUTB[d#]	SAR[a]	WAITVID[d]
ADDS[a]	DJNZ[a]	IF_NC_OR_NZ[a]	MAX[a]	PAR[d]	SHL[a]	WC[a]
ADDSX[a]	ELSE[s]	IF_NC_OR_Z[a]	MAXS[a]	PHSA[d]	SHR[a]	WHILE[s]
ADDX[a]	ELSEIF[s]	IF_NE[a]	MIN[a]	PHSB[d]	SPR[s]	WORD[s]
AND[d]	ELSEIFNOT[s]	IF_NEVER[a]	MINS[a]	PI[d]	STEP[s]	WORDFILL[s]
ANDN[a]	ENC[a#]	IF_NZ[a]	MOV[a]	PLL1X[s]	STRCOMP[s]	WORDMOVE[s]
BYTE[s]	FALSE[d]	IF_NZ_AND_C[a]	MOVD[a]	PLL2X[s]	STRING[s]	WR[a]
BYTEFILL[s]	FILE[s]	IF_NZ_AND_NC[a]	MOVI[a]	PLL4X[s]	STRSIZE[s]	WRBYTE[a]
BYTEMOVE[s]	FIT[a]	IF_NZ_OR_C[a]	MOVS[a]	PLL8X[s]	SUB[a]	WRLONG[a]
CALL[a]	FLOAT[s]	IF_NZ_OR_NC[a]	MUL[a#]	PLL16X[s]	SUBABS[a]	WRWORD[a]
CASE[s]	FROM[s]	IF_Z[a]	MULS[a#]	POSX[d]	SUBS[a]	WZ[a]
CHIPVER[s]	FRQA[d]	IF_Z_AND_C[a]	MUXC[a]	PRI[s]	SUBSX[a]	XINPUT[s]
CLKFREQ[s]	FRQB[d]	IF_Z_AND_NC[a]	MUXNC[a]	PUB[s]	SUBX[a]	XOR[a]
CLKMODE[s]	HUBOP[a]	IF_Z_EQ_C[a]	MUXNZ[a]	QUIT[s]	SUMC[a]	XTAL1[s]
CLKSET[d]	IF[s]	IF_Z_NE_C[a]	MUXZ[a]	RCFAST[s]	SUMNC[a]	XTAL2[s]
CMP[a]	IFNOT[s]	IF_Z_OR_C[a]	NEG[a]	RCL[a]	SUMNZ[a]	XTAL3[s]
CMPS[a]	IF_A[a]	IF_Z_OR_NC[a]	NEGC[a]	RCR[a]	SUMZ[a]	
CMPSUB[a]	IF_AE[a]	INA[d]	NEGNC[a]	RCSLOW[s]	TEST[a]	
CMPSX[a]	IF_ALWAYS[a]	INB[d#]	NEGNZ[a]	RDBYTE[a]	TESTN[a]	
CMPX[a]	IF_B[a]	JMP[a]	NEGX[d]	RDLONG[a]	TJNZ[a]	
CNT[d]	IF_BE[a]	JMPRET[a]	NEGZ[a]	RDWORD[a]	TJZ[a]	
COGID[d]	IF_C[a]	LOCKCLR[d]	NEXT[s]	REBOOT[s]	TO[s]	

a = Assembly element; s = Spin element; d = dual (available in both languages); # = reserved for future use

6.1.1. Words Reserved for Future Use

- DIRB, INB, and OUTB: Reserved for future use with a possible 64 I/O pin model. When used with the P8X32A, these labels can be used to access Cog RAM at those locations for general-purpose use.
- ENC, MUL, MULS, ONES: Use with the current P8X32A architecture yields indeterminate results.

6.2. Math and Logic Operators

Table 17: Math and Logic Operators

Level[1]	Operator		Constant Expressions[3]		Is Unary	Description
	Normal	Assign[2]	Integer	Float		
Highest (0)	--	always			✓	Pre-decrement (--X) or post-decrement (X--).
	++	always			✓	Pre-increment (++X) or post-increment (X++).
	~	always			✓	Sign-extend bit 7 (~X) or post-clear to 0 (X~).
	~~	always			✓	Sign-extend bit 15 (~~X) or post-set to -1 (X~~).
	?	always			✓	Random number forward (?X) or reverse (X?).
	@	never	✓		✓	Symbol address.
	@@	never			✓	Object address plus symbol.
1	+	never	✓	✓	✓	Positive (+X); unary form of Add.
	-	if solo	✓	✓	✓	Negate (-X); unary form of Subtract.
	^^	if solo	✓	✓	✓	Square root.
	||	if solo	✓	✓	✓	Absolute value.
	|<	if solo	✓		✓	Bitwise: Decode 0 -- 31 to long w/single-high-bit.
	>|	if solo	✓		✓	Bitwise: Encode long to 0 -- 32; high-bit priority.
	!	if solo	✓		✓	Bitwise: NOT.
2	<-	<-=	✓			Bitwise: Rotate left.
	->	->=	✓			Bitwise: Rotate right.
	<<	<<=	✓			Bitwise: Shift left.
	>>	>>=	✓			Bitwise: Shift right.
	~>	~>=	✓			Shift arithmetic right.
	><	><=	✓			Bitwise: Reverse.
3	&	&=	✓			Bitwise: AND.
4	|	|=	✓			Bitwise: OR.
	^	^=	✓			Bitwise: XOR.
5	*	*=	✓	✓		Multiply and return lower 32 bits (signed).
	**	**=	✓			Multiply and return upper 32 bits (signed).
	/	/=	✓	✓		Divide (signed).
	//	//=	✓			Modulus (signed).
6	+	+=	✓	✓		Add.
	-	-=	✓	✓		Subtract.
7	#>	#>=	✓	✓		Limit minimum (signed).
	<#	<#=	✓	✓		Limit maximum (signed).
8	<	<=	✓	✓		Boolean: Is less than (signed).
	>	>=	✓	✓		Boolean: Is greater than (signed).
	<>	<>=	✓	✓		Boolean: Is not equal.
	==	===	✓	✓		Boolean: Is equal.
	=<	=<=	✓	✓		Boolean: Is equal or less (signed).
	=>	=>=	✓	✓		Boolean: Is equal or greater (signed).
9	NOT	if solo	✓	✓	✓	Boolean: NOT (promotes non-0 to -1).
10	AND	AND=	✓	✓		Boolean: AND (promotes non-0 to -1).
11	OR	OR=	✓	✓		Boolean: OR (promotes non-0 to -1).
Lowest (12)	=	always	n/a[3]	n/a[3]		Constant assignment (CON blocks).
	:=	always	n/a[3]	n/a[3]		Variable assignment (PUB/PRI blocks).

[1] Precedence level: higher-level operators evaluate before lower-level operators. Operators in same level are commutable; evaluation order does not matter.
[2] Assignment forms of binary (non-unary) operators are in the lowest precedence (level 12).
[3] Assignment forms of operators are not allowed in constant expressions.

6.3. Spin Language Summary Table

Spin Command	Returns Value	Description
ABORT ⟨*Value*⟩	✓	Exit from PUB/PRI method using abort status with optional return value.
BYTE *Symbol* ⟨[*Count*]⟩		Declare byte-sized symbol in VAR block.
⟨*Symbol*⟩ BYTE *Data* ⟨[*Count*]⟩		Declare byte-aligned and/or byte-sized data in DAT block.
BYTE [*BaseAddress*] ⟨[*Offset*]⟩	✓	Read/write byte of main memory.
Symbol.BYTE ⟨[*Offset*]⟩	✓	Read/write byte-sized component of word/long-sized variable.
BYTEFILL (*StartAddress*, *Value*, *Count*)		Fill bytes of main memory with a value.
BYTEMOVE (*DestAddress*, *SrcAddress*, *Count*)		Copy bytes from one region to another in main memory.
CASE *CaseExpression* →I *MatchExpression* : →I *Statement(s)* ⟨→I *MatchExpression* : →I *Statement(s)*⟩ ⟨→I OTHER : →I *Statement(s)*⟩		Compare expression against matching expression(s), execute code block if match found. *MatchExpression* can contain a single expression or multiple comma-delimited expressions. Expressions can be a single value (ex: 10) or a range of values (ex: 10..15).
CHIPVER	✓	Version number of the Propeller chip (Byte at $FFFF)
CLKFREQ	✓	Current System Clock frequency, in Hz (Long at $0000)
CLKMODE	✓	Current clock mode setting (Byte at $0004)
CLKSET (*Mode*, *Frequency*)		Set both clock mode and System Clock frequency at run time.
CNT	✓	Current 32-bit System Counter value.
COGID	✓	Current cog's ID number; 0-7.
COGINIT (*CogID*, *SpinMethod* ⟨(*ParameterList*)⟩, *StackPointer*)		Start or restart cog by ID to run Spin code.
COGINIT (*CogID*, *AsmAddress*, *Parameter*)		Start or restart cog by ID to run Propeller Assembly code.
COGNEW (*SpinMethod* ⟨(*ParameterList*)⟩, *StackPointer*)	✓	Start new cog for Spin code and get cog ID; 0-7 = succeeded, -1 = failed.
COGNEW (*AsmAddress*, *Parameter*)	✓	Start new cog for Propeller Assembly code and get cog ID; 0-7 = succeeded, -1 = failed.
COGSTOP (*CogID*)		Stop cog by its ID.
CON *Symbol* = *Expr* ⟨((, ¦ →)) *Symbol* = *Expr*⟩		Declare symbolic, global constants.
CON #*Expr* ((, ¦ →)) *Symbol* ⟨[*Offset*]⟩ ⟨((, ¦ →)) *Symbol* ⟨[*Offset*]⟩		Declare global enumerations (incrementing symbolic constants).
CON *Symbol* ⟨[*Offset*]⟩ ⟨((, ¦ →)) *Symbol* ⟨[*Offset*]⟩ ⟩		Declare global enumerations (incrementing symbolic constants).
CONSTANT (*ConstantExpression*)	✓	Declare in-line constant expression to be completely resolved at compile time.
CTRA	✓	Counter A Control register.
CTRB	✓	Counter B Control register.
DAT ⟨*Symbol*⟩ *Alignment* ⟨*Size*⟩ ⟨*Data*⟩ ⟨[*Count*]⟩ ⟨, ⟨*Size*⟩ *Data* ⟨[*Count*]⟩⟩		Declare table of data, aligned and sized as specified.
DAT ⟨*Symbol*⟩ ⟨*Condition*⟩ *Instruction* ⟨*Effect(s)*⟩		Denote Propeller Assembly instruction.
DIRA ⟨[*Pin(s)*]⟩	✓	Direction register for 32-bit port A. Default is 0 (input) upon cog startup.
FILE "*FileName*"		Import external file as data in DAT block.
FLOAT (*IntegerConstant*)	✓	Convert integer constant expression to compile-time floating-point value in any block.
FRQA	✓	Counter A Frequency register.
FRQB	✓	Counter B Frequency register.

Spin Command	Returns Value	Description
((IF ¦ IFNOT)) *Condition(s)* → *IfStatement(s)* ⟨ELSEIF *Condition(s)* → *ElseIfStatement(s)*⟩ ⟨ELSEIFNOT *Condition(s)* → *ElseIfStatement(s)*⟩ ⟨ELSE → *ElseStatement(s)*⟩		Test condition(s) and execute block of code if valid. IF and ELSEIF each test for TRUE. IFNOT and ELSEIFNOT each test for FALSE.
INA ⟨[*Pin(s)*]⟩	✓	Input register for 32-bit ports A.
LOCKCLR (*ID*)	✓	Clear semaphore to false and get its previous state; TRUE or FALSE.
LOCKNEW	✓	Check out new semaphore and get its ID; 0-7, or -1 if none were available.
LOCKRET (*ID*)		Return semaphore back to semaphore pool, releasing it for future LOCKNEW requests.
LOCKSET (*ID*)	✓	Set semaphore to true and get its previous state; TRUE or FALSE.
LONG *Symbol* ⟨[*Count*]⟩		Declare long-sized symbol in VAR block.
⟨*Symbol*⟩ LONG *Data* ⟨[*Count*]⟩		Declare long-aligned and/or long-sized data in DAT block.
LONG [*BaseAddress*] ⟨[*Offset*]⟩	✓	Read/write long of main memory.
LONGFILL (*StartAddress*, *Value*, *Count*)		Fill longs of main memory with a value.
LONGMOVE (*DestAddress*, *SrcAddress*, *Count*)		Copy longs from one region to another in main memory.
LOOKDOWN (*Value*: *ExpressionList*)	✓	Get the one-based index of a value in a list.
LOOKDOWNZ (*Value*: *ExpressionList*)	✓	Get the zero-based index of a value in a list.
LOOKUP (*Index*: *ExpressionList*)	✓	Get value from a one-based index position of a list.
LOOKUPZ (*Index*: *ExpressionList*)	✓	Get value from a zero-based index position of a list.
NEXT		Skip remaining statements of REPEAT loop and continue with the next loop iteration.
OBJ *Symbol* ⟨[*Count*]⟩: "*Object*" ⟨↳ *Symbol* ⟨[*Count*]⟩: "*Object*"⟩		Declare symbol object references.
OUTA ⟨[*Pin(s)*]⟩	✓	Output register for 32-bit port A. Default is 0 (ground) upon cog startup.
PAR	✓	Cog Boot Parameter register.
PHSA	✓	Counter A Phase Lock Loop (PLL) register.
PHSB	✓	Counter B Phase Lock Loop (PLL) register.
PRI *Name* ⟨(*Par* ⟨,*Par*⟩)⟩ ⟨:*RVal*⟩ ⟨| *LVar* ⟨[*Cnt*]⟩⟩ ⟨,*LVar* ⟨[*Cnt*]⟩⟩ *SourceCodeStatements*		Declare private method with optional parameters, return value and local variables.
PUB *Name* ⟨(*Par* ⟨,*Par*⟩)⟩ ⟨:*RVal*⟩ ⟨| *LVar* ⟨[*Cnt*]⟩⟩ ⟨,*LVar* ⟨[*Cnt*]⟩⟩ *SourceCodeStatements*		Declare public method with optional parameters, return value and local variables.
QUIT		Exit from REPEAT loop immediately.
REBOOT		Reset the Propeller chip.
REPEAT ⟨*Count*⟩ → *Statement(s)*		Execute code block repetitively, either infinitely, or for a finite number of iterations.
REPEAT *Variable* FROM *Start* TO *Finish* ⟨STEP *Delta*⟩ → *Statement(s)*		Execute code block repetitively, for finite, counted iterations.
REPEAT ((UNTIL ¦ WHILE)) *Condition(s)* → *Statement(s)*		Execute code block repetitively, zero-to-many conditional iterations.
REPEAT → *Statement(s)* ((UNTIL ¦ WHILE)) *Condition(s)*		Execute code block repetitively, one-to-many conditional iterations.
RESULT	✓	Return value variable for PUB/PRI methods.
RETURN ⟨*Value*⟩	✓	Exit from PUB/PRI method with optional return *Value*.
ROUND (*FloatConstant*)	✓	Round floating-point constant to the nearest integer at compile-time, in any block.
SPR [*Index*]	✓	Special Purpose Register array.
STRCOMP (*StringAddress1*, *StringAddress2*)	✓	Compare two strings for equality.
STRING (*StringExpression*)	✓	Declare in-line string constant and get its address.

Spin Command	Returns Value	Description
STRSIZE (*StringAddress*)	✓	Get size, in bytes, of zero-terminate string.
TRUNC (*FloatConstant*)	✓	Remove fractional portion from floating-point constant at compile-time, in any block.
VAR *Size Symbol* ⟨[*Count*]⟩ ⟨((, ↳ *Size*)) *Symbol* ⟨[*Count*]⟩⟩		Declare symbolic global variables.
VCFG	✓	Video Configuration register.
VSCL	✓	Video Scale register.
WAITCNT (*Value*)		Pause cog's execution temporarily.
WAITPEQ (*State*, *Mask*, *Port*)		Pause cog's execution until I/O pin(s) match designated state(s).
WAITPNE (*State*, *Mask*, *Port*)		Pause cog's execution until I/O pin(s) do not match designated state(s).
WAITVID (*Colors*, *Pixels*)		Pause cog's execution until its Video Generator is available for pixel data.
WORD *Symbol* ⟨[*Count*]⟩		Declare word-sized symbol in VAR block.
⟨*Symbol*⟩ WORD *Data* ⟨[*Count*]⟩		Declare word-aligned and/or word-sized data in DAT block.
WORD [*BaseAddress*] ⟨[*Offset*]⟩	✓	Read/write word of main memory.
Symbol.WORD ⟨[*Offset*]⟩	✓	Read/write word-sized component of long-sized variable.
WORDFILL (*StartAddress*, *Value*, *Count*)		Fill words of main memory with a value.
WORDMOVE (*DestAddress*, *SrcAddress*, *Count*)		Copy words from one region to another in main memory.

6.3.1. Constants

Constants (pre-defined)			
Constant[1]	Description		
_CLKFREQ	Settable in Top Object File to specify System Clock frequency.		
_CLKMODE	Settable in Top Object File to specify application's clock mode.		
_XINFREQ	Settable in Top Object File to specify external crystal frequency.		
_FREE	Settable in Top Object File to specify application's free space.		
_STACK	Settable in Top Object File to specify application's stack space.		
TRUE	Logical true:	-1	($FFFFFFFF)
FALSE	Logical false:	0	($00000000)
POSX	Max. positive integer:	2,147,483,647	($7FFFFFFF)
NEGX	Max. negative integer:	-2,147,483,648	($80000000)
PI	Floating-point PI:	≈ 3.141593	($40490FDB)
RCFAST	Internal fast oscillator:	$00000001	(%00000000001)
RCSLOW	Internal slow oscillator:	$00000002	(%00000000010)
XINPUT	External clock/oscillator:	$00000004	(%00000000100)
XTAL1	External low-speed crystal:	$00000008	(%00000001000)
XTAL2	External medium-speed crystal:	$00000010	(%00000010000)
XTAL3	External high-speed crystal:	$00000020	(%00000100000)
PLL1X	External frequency times 1:	$00000040	(%00001000000)
PLL2X	External frequency times 2:	$00000080	(%00010000000)
PLL4X	External frequency times 4:	$00000100	(%00100000000)
PLL8X	External frequency times 8:	$00000200	(%01000000000)
PLL16X	External frequency times 16:	$00000400	(%10000000000)

[1] "Settable" constants are defined in Top Object File's CON block. See Valid Clock Modes for _CLKMODE. Other settable constants use whole numbers.

6.4. Propeller Assembly Instruction Table

The Propeller Assembly Instruction Table lists the instruction's 32-bit opcode, outputs and number of clock cycles. The opcode consists of the instruction bits (iiiiii), the "effect" status for the Z flag, C flag, result and indirect/immediate status (zcri), the conditional execution bits (cccc), and the destination and source bits (ddddddddd and sssssssss). The meaning of the Z and C flags, if any, is shown in the **Z Result** and **C Result** fields; indicating the meaning of a 1 in those flags. The Result field (R) shows the instruction's default behavior for writing (1) or not writing (0) the instruction's result value. The **Clocks** field shows the number of clocks the instruction requires for execution.

0 1	Zeros (0) and ones (1) mean binary 0 and 1.
i	Lower case "i" denotes a bit that is affected by immediate status.
d s	Lower case "d" and "s" indicate destination and source bits.
?	Question marks denote bits that are dynamically set by the compiler.
---	Hyphens indicate items that are not applicable or not important.
..	Double-periods represent a range of contiguous values.

iiiiii zcri cccc ddddddddd sssssssss	Instruction	Description	Z Result	C Result	R	Clocks
000000 000i 1111 ddddddddd sssssssss	WRBYTE D,S	Write D[7..0] to main memory byte S[15..0]	-	-	0	7..22 *
000000 001i 1111 ddddddddd sssssssss	RDBYTE D,S	Read main memory byte S[15..0] into D (0-extended)	Result = 0	-	1	7..22 *
000001 000i 1111 ddddddddd sssssssss	WRWORD D,S	Write D[15..0] to main memory word S[15..1]	-	-	0	7..22 *
000001 001i 1111 ddddddddd sssssssss	RDWORD D,S	Read main memory word S[15..1] into D (0-extended)	Result = 0	-	1	7..22 *
000010 000i 1111 ddddddddd sssssssss	WRLONG D,S	Write D to main memory long S[15..2]	-	-	0	7..22 *
000010 001i 1111 ddddddddd sssssssss	RDLONG D,S	Read main memory long S[15..2] into D	Result = 0	-	1	7..22 *
000011 000i 1111 ddddddddd sssssssss	HUBOP D,S	Perform hub operation according to S	Result = 0	-	0	7..22 *
000011 0001 1111 ddddddddd ------000	CLKSET D	Set the global CLK register to D[7..0]	-	-	0	7..22 *
000011 0011 1111 ddddddddd ------001	COGID D	Get this cog number (0..7) into D	Result = 0	-	1	7..22 *
000011 0001 1111 ddddddddd ------010	COGINIT D	Initialize a cog according to D	Result = 0	No cog free	0	7..22 *
000011 0001 1111 ddddddddd ------011	COGSTOP D	Stop cog number D[2..0]	-	-	0	7..22 *
000011 0011 1111 ddddddddd ------100	LOCKNEW D	Checkout a new LOCK number (0..7) into D	Result = 0	No lock free	1	7..22 *
000011 0001 1111 ddddddddd ------101	LOCKRET D	Return lock number D[2..0]	-	-	0	7..22 *
000011 0001 1111 ddddddddd ------110	LOCKSET D	Set lock number D[2..0]	-	Prior lock state	0	7..22 *
000011 0001 1111 ddddddddd ------111	LOCKCLR D	Clear lock number D[2..0]	-	Prior lock state	0	7..22 *
000100 001i 1111 ddddddddd sssssssss	MUL D,S	Multiply unsigned D[15..0] by S[15..0]	Result = 0	-	1	future
000101 001i 1111 ddddddddd sssssssss	MULS D,S	Multiply signed D[15..0] by S[15..0]	Result = 0	-	1	future
000110 001i 1111 ddddddddd sssssssss	ENC D,S	Encode magnitude of S into D, result = 0..31	Result = 0	-	1	future
000111 001i 1111 ddddddddd sssssssss	ONES D,S	Get number of 1's in S into D, result = 0..31	Result = 0	-	1	future
001000 001i 1111 ddddddddd sssssssss	ROR D,S	Rotate D right by S[4..0] bits	Result = 0	D[0]	1	4
001001 001i 1111 ddddddddd sssssssss	ROL D,S	Rotate D left by S[4..0] bits	Result = 0	D[31]	1	4
001010 001i 1111 ddddddddd sssssssss	SHR D,S	Shift D right by S[4..0] bits, set new MSB to 0	Result = 0	D[0]	1	4
001011 001i 1111 ddddddddd sssssssss	SHL D,S	Shift D left by S[4..0] bits, set new LSB to 0	Result = 0	D[31]	1	4
001100 001i 1111 ddddddddd sssssssss	RCR D,S	Rotate carry right into D by S[4..0] bits	Result = 0	D[0]	1	4
001101 001i 1111 ddddddddd sssssssss	RCL D,S	Rotate carry left into D by S[4..0] bits	Result = 0	D[31]	1	4
001110 001i 1111 ddddddddd sssssssss	SAR D,S	Shift D arithmetically right by S[4..0] bits	Result = 0	D[0]	1	4
001111 001i 1111 ddddddddd sssssssss	REV D,S	Reverse 32 S[4..0] bottom bits in D and 0-extend	Result = 0	D[0]	1	4
010000 001i 1111 ddddddddd sssssssss	MINS D,S	Set D to S if signed (D < S)	D = S	Signed (D < S)	1	4
010001 001i 1111 ddddddddd sssssssss	MAXS D,S	Set D to S if signed (D => S)	D = S	Signed (D < S)	1	4
010010 001i 1111 ddddddddd sssssssss	MIN D,S	Set D to S if unsigned (D < S)	D = S	Unsigned (D < S)	1	4
010011 001i 1111 ddddddddd sssssssss	MAX D,S	Set D to S if unsigned (D => S)	D = S	Unsigned (D < S)	1	4
010100 001i 1111 ddddddddd sssssssss	MOVS D,S	Insert S[8..0] into D[8..0]	Result = 0	-	1	4
010101 001i 1111 ddddddddd sssssssss	MOVD D,S	Insert S[8..0] into D[17..9]	Result = 0	-	1	4
010110 001i 1111 ddddddddd sssssssss	MOVI D,S	Insert S[8..0] into D[31..23]	Result = 0	-	1	4
010111 001i 1111 ddddddddd sssssssss	JMPRET D,S	Insert PC+1 into D[8..0] and set PC to S[8..0]	Result = 0	-	1	4
010111 000i 1111 --------- sssssssss	JMP S	Set PC to S[8..0]	Result = 0	-	0	4

iiiiii zcri cccc dddddddd ssssssss	Instruction	Description	Z Result	C Result	R	Clocks
010111 0011 1111 ???????? ssssssss	CALL #S	Like JMPRET, but assembler handles details	Result = 0	-	1	4
010111 0001 1111 --------- ---------	RET	Like JMP, but assembler handles details	Result = 0	-	0	4
011000 000i 1111 ddddddddd ssssssss	TEST D,S	AND S with D to affect flags only	Result = 0	Parity of Result	0	4
011001 000i 1111 ddddddddd ssssssss	TESTN D,S	AND !S into D to affect flags only	Result = 0	Parity of Result	0	4
011000 001i 1111 ddddddddd ssssssss	AND D,S	AND S into D	Result = 0	Parity of Result	1	4
011001 001i 1111 ddddddddd ssssssss	ANDN D,S	AND !S into D	Result = 0	Parity of Result	1	4
011010 001i 1111 ddddddddd ssssssss	OR D,S	OR S into D	Result = 0	Parity of Result	1	4
011011 001i 1111 ddddddddd ssssssss	XOR D,S	XOR S into D	Result = 0	Parity of Result	1	4
011100 001i 1111 ddddddddd ssssssss	MUXC D,S	Copy C to bits in D using S as mask	Result = 0	Parity of Result	1	4
011101 001i 1111 ddddddddd ssssssss	MUXNC D,S	Copy !C to bits in D using S as mask	Result = 0	Parity of Result	1	4
011110 001i 1111 ddddddddd ssssssss	MUXZ D,S	Copy Z to bits in D using S as mask	Result = 0	Parity of Result	1	4
011111 001i 1111 ddddddddd ssssssss	MUXNZ D,S	Copy !Z to bits in D using S as mask	Result = 0	Parity of Result	1	4
100000 001i 1111 ddddddddd ssssssss	ADD D,S	Add S into D	Result = 0	Unsigned Carry	1	4
100001 001i 1111 ddddddddd ssssssss	SUB D,S	Subtract S from D	Result = 0	Unsigned Borrow	1	4
100001 000i 1111 ddddddddd ssssssss	CMP D,S	Compare D to S	D = S	Unsigned Borrow	0	4
100010 001i 1111 ddddddddd ssssssss	ADDABS D,S	Add absolute S into D	Result = 0	Unsigned Carry[1]	1	4
100011 001i 1111 ddddddddd ssssssss	SUBABS D,S	Subtract absolute S from D	Result = 0	Unsigned Borrow[2]	1	4
100100 001i 1111 ddddddddd ssssssss	SUMC D,S	Sum either –S if C or S if !C into D	Result = 0	Signed Overflow	1	4
100101 001i 1111 ddddddddd ssssssss	SUMNC D,S	Sum either S if C or –S if !C into D	Result = 0	Signed Overflow	1	4
100110 001i 1111 ddddddddd ssssssss	SUMZ D,S	Sum either –S if Z or S if !Z into D	Result = 0	Signed Overflow	1	4
100111 001i 1111 ddddddddd ssssssss	SUMNZ D,S	Sum either S if Z or –S if !Z into D	Result = 0	Signed Overflow	1	4
101000 001i 1111 ddddddddd ssssssss	MOV D,S	Set D to S	Result = 0	S[31]	1	4
101001 001i 1111 ddddddddd ssssssss	NEG D,S	Set D to –S	Result = 0	S[31]	1	4
101010 001i 1111 ddddddddd ssssssss	ABS D,S	Set D to absolute S	Result = 0	S[31]	1	4
101011 001i 1111 ddddddddd ssssssss	ABSNEG D,S	Set D to –absolute S	Result = 0	S[31]	1	4
101100 001i 1111 ddddddddd ssssssss	NEGC D,S	Set D to either –S if C or S if !C	Result = 0	S[31]	1	4
101101 001i 1111 ddddddddd ssssssss	NEGNC D,S	Set D to either S if C or –S if !C	Result = 0	S[31]	1	4
101110 001i 1111 ddddddddd ssssssss	NEGZ D,S	Set D to either –S if Z or S if !Z	Result = 0	S[31]	1	4
101111 001i 1111 ddddddddd ssssssss	NEGNZ D,S	Set D to either S if Z or –S if !Z	Result = 0	S[31]	1	4
110000 000i 1111 ddddddddd ssssssss	CMPS D,S	Compare-signed D to S	D = S	Signed Borrow	0	4
110001 000i 1111 ddddddddd ssssssss	CMPSX D,S	Compare-signed-extended D to S+C	Z & (D = S+C)	Signed Borrow	0	4
110010 001i 1111 ddddddddd ssssssss	ADDX D,S	Add-extended S+C into D	Z & (Result = 0)	Unsigned Carry	1	4
110011 001i 1111 ddddddddd ssssssss	SUBX D,S	Subtract-extended S+C from D	Z & (Result = 0)	Unsigned Borrow	1	4
110011 000i 1111 ddddddddd ssssssss	CMPX D,S	Compare-extended D to S+C	Z & (D = S+C)	Unsigned Borrow	0	4
110100 001i 1111 ddddddddd ssssssss	ADDS D,S	Add-signed S into D	Result = 0	Signed Overflow	1	4
110101 001i 1111 ddddddddd ssssssss	SUBS D,S	Subtract-signed S from D	Result = 0	Signed Overflow	1	4
110110 001i 1111 ddddddddd ssssssss	ADDSX D,S	Add-signed-extended S+C into D	Z & (Result = 0)	Signed Overflow	1	4
110111 001i 1111 ddddddddd ssssssss	SUBSX D,S	Subtract-signed-extended S+C from D	Z & (Result = 0)	Signed Overflow	1	4
111000 001i 1111 ddddddddd ssssssss	CMPSUB D,S	Subtract S from D if D => S	D = S	Unsigned (D => S)	1	4
111001 001i 1111 ddddddddd ssssssss	DJNZ D,S	Dec D, jump if not zero to S (no jump = 8 clocks)	Result = 0	Unsigned Borrow	1	4 or 8
111010 000i 1111 ddddddddd ssssssss	TJNZ D,S	Test D, jump if not zero to S (no jump = 8 clocks)	Result = 0	0	0	4 or 8
111011 000i 1111 ddddddddd ssssssss	TJZ D,S	Test D, jump if zero to S (no jump = 8 clocks)	Result = 0	0	0	4 or 8
111100 000i 1111 ddddddddd ssssssss	WAITPEQ D,S	Wait for pins equal - (INA & S) = D	-	-	0	5+
111101 000i 1111 ddddddddd ssssssss	WAITPNE D,S	Wait for pins not equal - (INA & S) != D	-	-	0	5+
111110 001i 1111 ddddddddd ssssssss	WAITCNT D,S	Wait for CNT = D, then add S into D	-	Unsigned Carry	1	5+
111111 000i 1111 ddddddddd ssssssss	WAITVID D,S	Wait for video peripheral to grab D and S	-	-	0	5+
------ ---- 0000 --------- ---------	NOP	No operation, just elapses 4 clocks	-	-	–	4

* See Hub, section 4.4 on page 7.

1. ADDABS C out: If S is negative, C = the inverse of unsigned *borrow* (for D-S).
2. SUBABS C out: If S is negative, C = the inverse of unsigned *carry* (for D+S).

6.4.1. Assembly Conditions

Condition	Instruction Executes
IF_ALWAYS	always
IF_NEVER	never
IF_E	if equal (Z)
IF_NE	if not equal (!Z)
IF_A	if above (!C & !Z)
IF_B	if below (C)
IF_AE	if above/equal (!C)
IF_BE	if below/equal (C | Z)
IF_C	if C set
IF_NC	if C clear
IF_Z	if Z set
IF_NZ	if Z clear
IF_C_EQ_Z	if C equal to Z
IF_C_NE_Z	if C not equal to Z
IF_C_AND_Z	if C set and Z set
IF_C_AND_NZ	if C set and Z clear
IF_NC_AND_Z	if C clear and Z set
IF_NC_AND_NZ	if C clear and Z clear
IF_C_OR_Z	if C set or Z set
IF_C_OR_NZ	if C set or Z clear
IF_NC_OR_Z	if C clear or Z set
IF_NC_OR_NZ	if C clear or Z clear
IF_Z_EQ_C	if Z equal to C
IF_Z_NE_C	if Z not equal to C
IF_Z_AND_C	if Z set and C set
IF_Z_AND_NC	if Z set and C clear
IF_NZ_AND_C	if Z clear and C set
IF_NZ_AND_NC	if Z clear and C clear
IF_Z_OR_C	if Z set or C set
IF_Z_OR_NC	if Z set or C clear
IF_NZ_OR_C	if Z clear or C set
IF_NZ_OR_NC	if Z clear or C clear

6.4.2. Assembly Directives

Directive	Description
FIT ⟨*Address*⟩	Validate previous instr/data fit below an address.
ORG ⟨*Address*⟩	Adjust compile-time cog address pointer.
⟨*Symbol*⟩ RES ⟨*Count*⟩	Reserve next long(s) for symbol.

6.4.3. Assembly Effects

Effect	Results In
WC	C Flag modified
WZ	Z Flag modified
WR	Destination Register modified
NR	Destination Register not modified

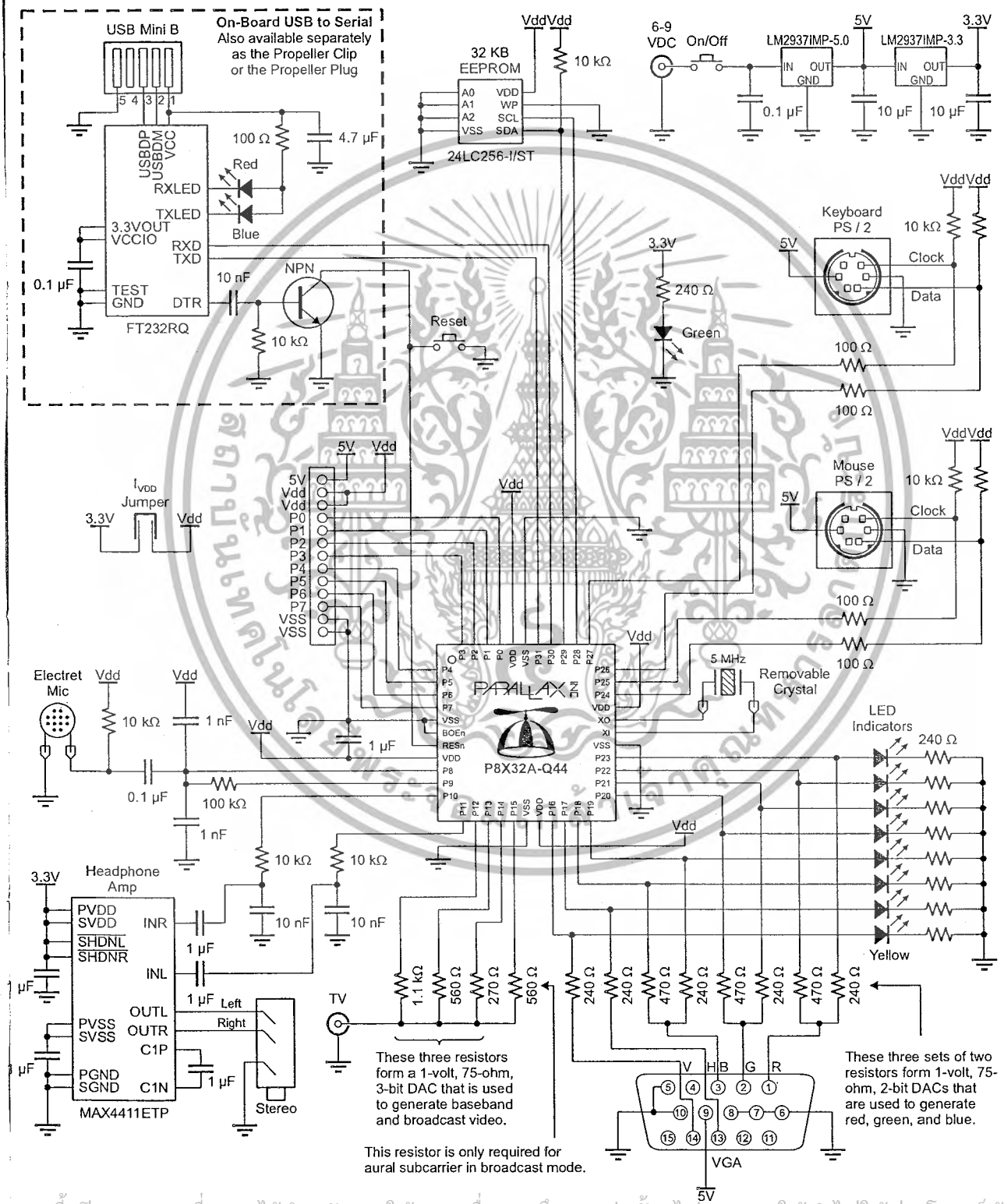
6.4.4. Assembly Operators

Propeller Assembly code can contain constant expressions, which may use any operators that are allowed in constant expressions. The table (a subset of Table 17) lists the operators allowed in Propeller Assembly.

Operator	Description
+	Add
+	Positive (+X); unary form of Add
-	Subtract
-	Negate (-X); unary form of Subtract
*	Multiply and return lower 32 bits (signed)
**	Multiply and return upper 32 bits (signed)
/	Divide (signed)
//	Modulus (signed)
#>	Limit minimum (signed)
<#	Limit maximum (signed)
^^	Square root; unary
||	Absolute value; unary
~>	Shift arithmetic right
|<	Bitwise: Decode value (0-31) into single-high-bit long; unary
>|	Bitwise: Encode long into value (0 - 32) as high-bit priority; unary
<<	Bitwise: Shift left
>>	Bitwise: Shift right
<-	Bitwise: Rotate left
->	Bitwise: Rotate right
><	Bitwise: Reverse
&	Bitwise: AND
|	Bitwise: OR
^	Bitwise: XOR
!	Bitwise: NOT; unary
AND	Boolean: AND (promotes non-0 to -1)
OR	Boolean: OR (promotes non-0 to -1)
NOT	Boolean: NOT (promotes non-0 to -1); unary
==	Boolean: Is equal
<>	Boolean: Is not equal
<	Boolean: Is less than (signed)
>	Boolean: Is greater than (signed)
=<	Boolean: Is equal or less (signed)
=>	Boolean: Is equal or greater (signed)
@	Symbol address; unary

7.0 PROPELLER DEMO BOARD SCHEMATIC

The Propeller Demo Board (Stock #32100) provides convenient connections to 32K EEPROM, replaceable 5 MHz crystal, 3. 3V and 5 V regulators, USB-to-serial programming/communication interface, VGA and NTSC video output, stereo output with 16 Ω headphone amplifier, microphone input, two PS2 mouse and keyboard jacks, eight LEDs, eight free I/O pins brought to a header for breadboard for prototyping, and a ground post for an oscilloscope probe. Overall PCB size: 3" x 3".



8.0 ELECTRICAL CHARACTERISTICS

8.1. Absolute Maximum Ratings

Stresses in excess of the absolute maximum ratings can cause permanent damage to the device. These are absolute stress ratings only. Functional operation of the device is not implied at these or any other conditions in excess of those given in the remainder of Section 7.0. Exposure to absolute maximum ratings for extended periods can adversely affect device reliability.

Table 18: Absolute Maximum Ratings	
Ambient temperature under bias	-55 °C to +125 °C
Storage temperature	-65 °C to +150 °C
Voltage on V_{dd} with respect to V_{ss}	-0.3 V to +4.0 V
Voltage on all other pins with respect to V_{ss}	-0.3 V to (V_{dd} + 0.3 V)
Total power dissipation	1 W
Max. current out of V_{ss} pins	300 mA
Max. current into V_{dd} pins	300 mA
Max. DC current into an input pin with internal protection diode forward biased	±500 µA
Max. allowable current per I/O pin	40 mA
ESD (Human Body Model) Supply pins	3 kV
ESD (Human Body Model) all non-supply pins	8 kV

8.2. DC Characteristics

(Operating temperature range: -55° C < T_a < +125° C unless otherwise noted)

Symbol	Parameter	Conditions	Min	Typ	Max	Units
V_{dd}	Supply Voltage		2.7	-	3.6	V
V_{ih}, V_{il}	Logic High Logic Low		0.6 V_{dd} V_{ss}		V_{dd} 0.3 V_{dd}	V V
I_{il}	Input Leakage Current	V_{in} = V_{dd} or V_{ss}	-1.0		+1.0	µA
V_{oh}	Output High Voltage	I_{oh} = 10 mA, V_{dd} = 3.3 V	2.85			V
V_{ol}	Output Low Voltage	I_{ol} = 10 mA, V_{dd} = 3.3 V			0.4	V
I_{BO}	Brownout Detector Current			3.8		µA
I	Quiescent Current	RESn = 0V, BOEn = V_{dd}, P_0-P_{31}=0V		600		nA

Note: Data in the Typical ("Typ") column is T_a = 25 °C unless otherwise stated.

8.3. AC Characteristics

(Operating temperature range: -55°C < T_a < +125°C unless otherwise noted)

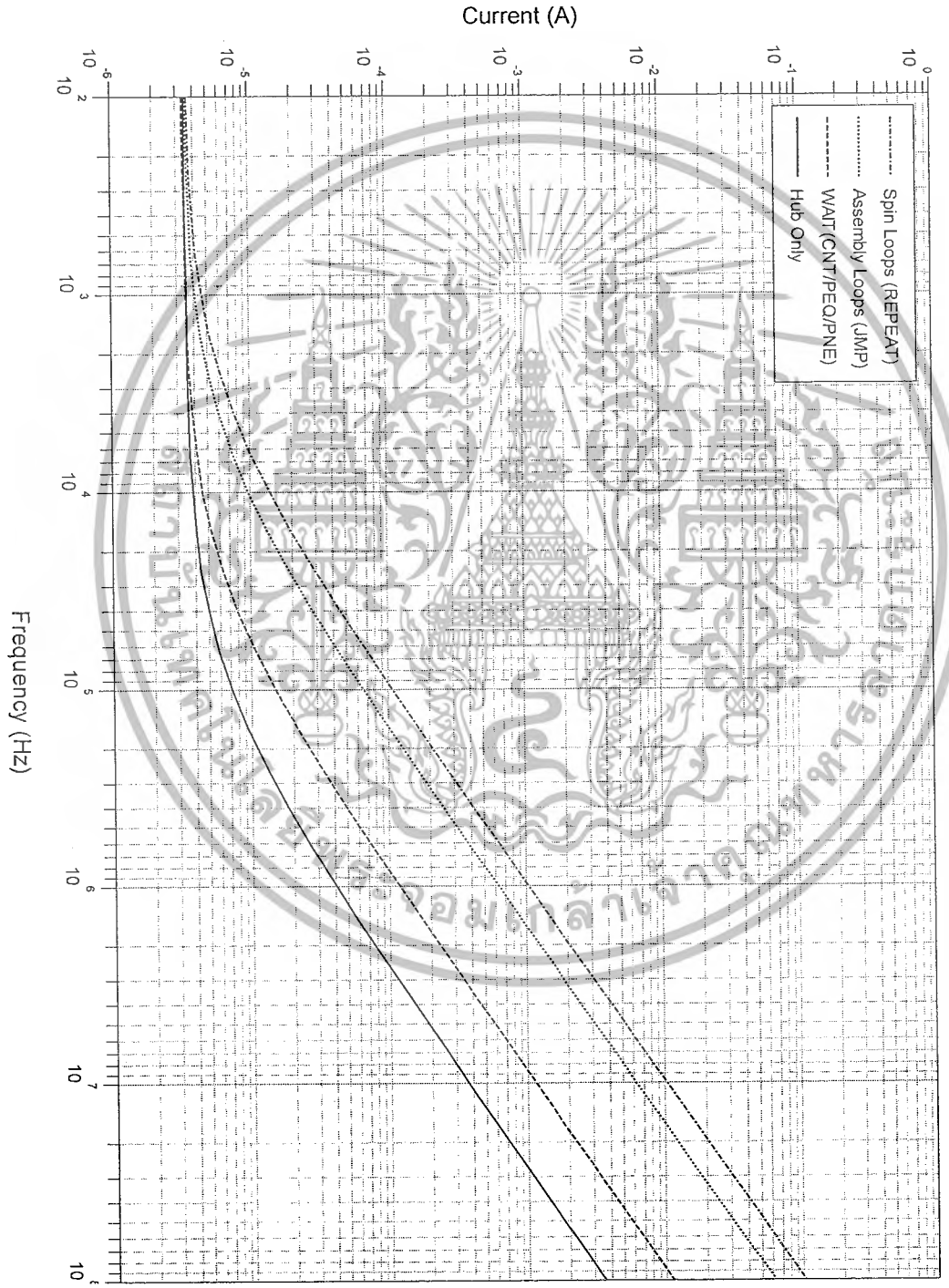
Symbol	Parameter	Min	Typ	Max	Units	Condition
F_{osc}	External XI Frequency	DC	-	80	MHz	
	Oscillator Frequency	DC 13 8 4	- 20 12 -	80 33 20 8	MHz kHz MHz MHz	Direct drive (no PLL) RCSLOW RCFAST Crystal using PLL
C_{in}	Input Capacitance		6	-	pF	

Note: Data in the Typical ("Typ") column is T_a = 25 °C unless otherwise stated.

9.0 CURRENT CONSUMPTION CHARACTERISTICS

9.1. Typical Current Consumption of 8 Cogs

This figure shows the typical current consumption of the Propeller under various operating conditions duplicated across all cogs. Brown out circuitry and the Phase-Locked Loop were disabled for the duration of the test. Current consumption is substantially constant over the operational temperature range.

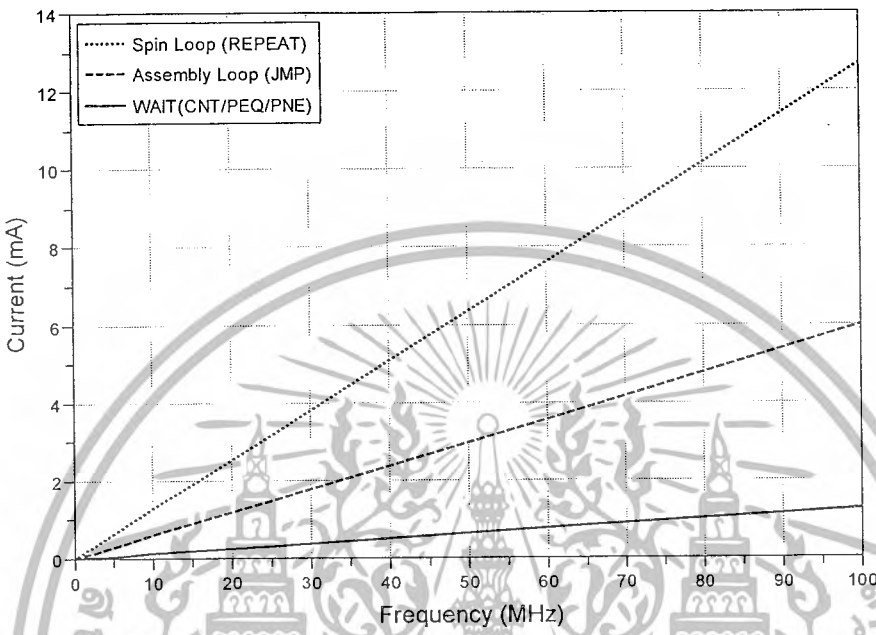


Typical Current Consumption of 8 cogs vs. Operating Frequency (3.3V, Ta = 25°C)

9.2. Typical Current of a Cog vs. Operating Frequency

This graph shows a cog's typical current consumption under various conditions, in isolation of other sources of current within the Propeller chip.

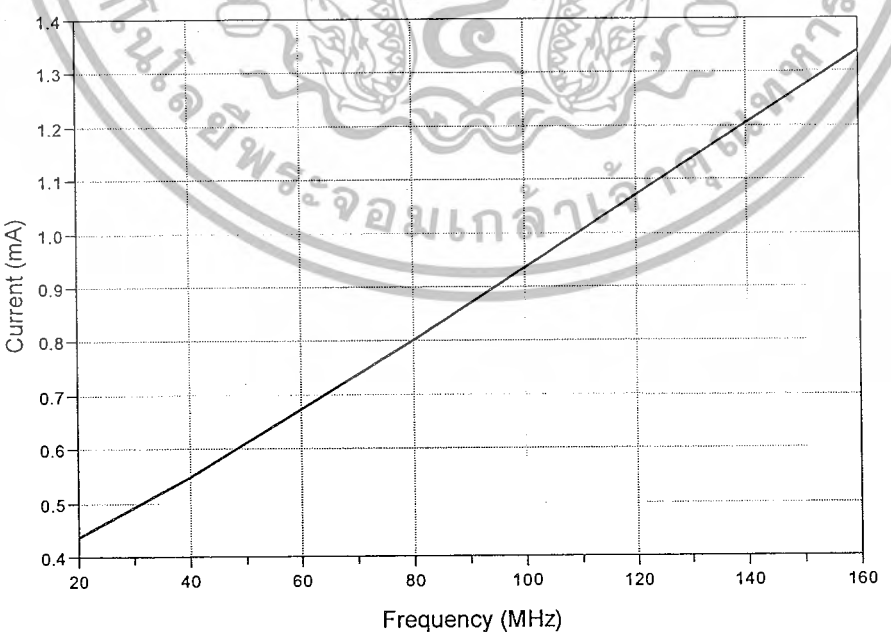
Typical Current of a Cog vs. Operating Frequency (Vdd = 3.3 V, Ta = 25° C)



9.3. Typical PLL Current vs. VCO Frequency

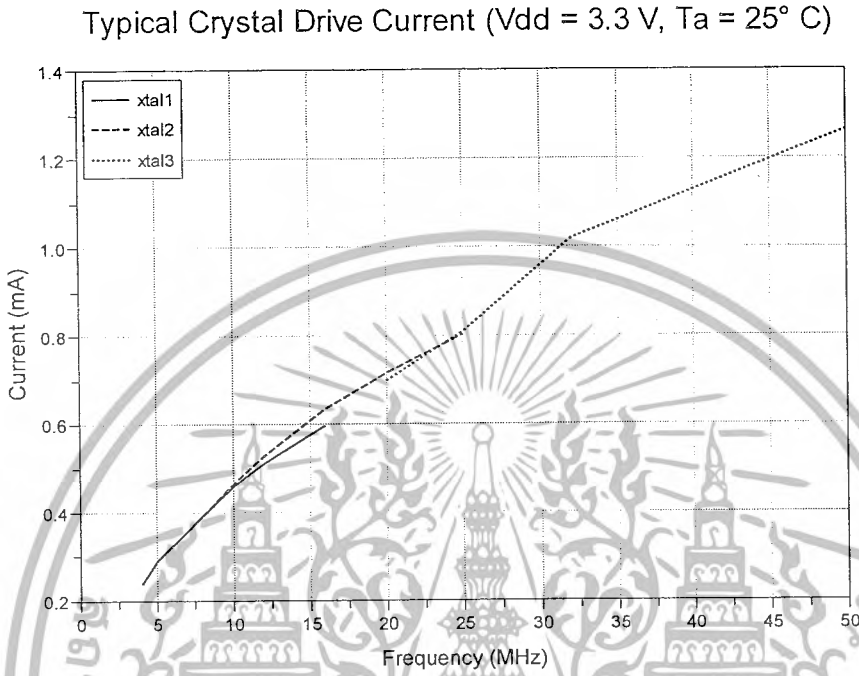
This graph shows the typical amount of current consumed by a Phase-Locked Loop as a function of the frequency of the Voltage Controlled Oscillator which is 16 times the frequency of the input clock.

Typical PLL Current vs. VCO Frequency (Vdd = 3.3 V, Ta = 25° C)



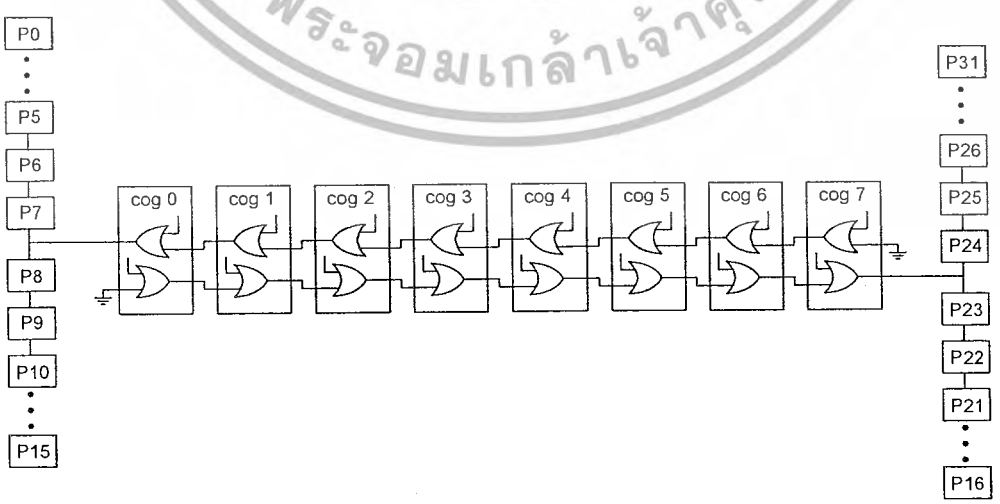
9.4. Typical Crystal Drive Current

This graph shows the current consumption of the crystal driver over a range of crystal frequencies and crystal settings, all data points above 25 MHz were obtained by using a resonator since the driver does not perform 3rd harmonic overtone driving required for crystals over 25 MHz.



9.5. Cog and I/O Pin Relationship

The figure below illustrates the physical relationship between the cogs and I/O pins. While there can be a 1 to 1.5 ns propagation delay in output transitions between the shortest and longest paths, the purpose of the figure is to illustrate the length of leads and their associated parasitic capacitance. This capacitance increases the amount of energy required to transition a pin's state and therefore increases the current draw for toggling a pin. So, the current consumed by Cog 7 toggling P0 at 20 MHz will be greater than Cog 0 toggling P7 at 20 MHz. The amount of current consumed by transitioning a pin's state is dependent on many factors including: temperature, frequency of transitions, external load, and internal load. As mentioned, the internal load is dependent upon which cog and pin are used. Internal load current for room temperature toggling of a pin at 20 MHz for a Propeller in a DIP package varies on the order of 300 µA.



9.6. Current Profile at Various Startup Conditions

The diagrams below show the current profile for various startup conditions of the Propeller chip dependent upon the presence of an EEPROM and PC.

Figure 7

Boot Sequence Current Profile for no PC and no EEPROM (P31 held low and P29 not connected (same as held low)).

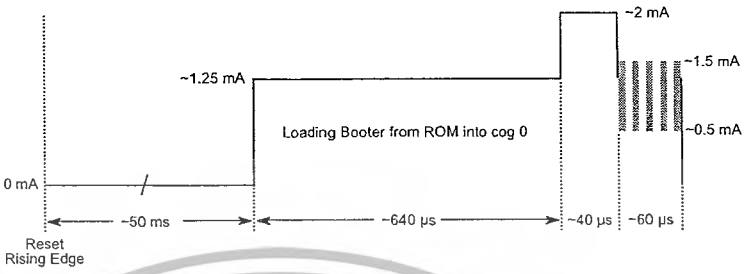


Figure 8

Boot Sequence Current Profile for PC (connected but idle) and no EEPROM. (P31 held high and P29 not connected).

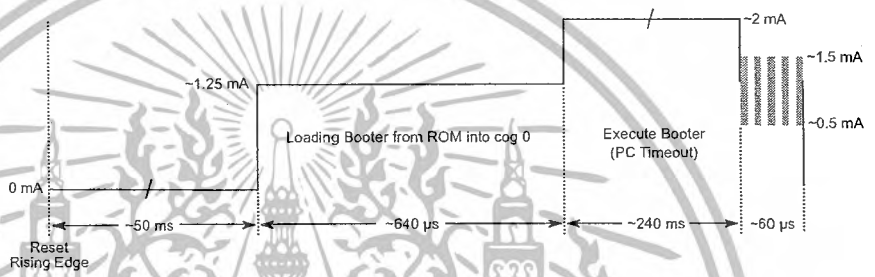


Figure 9

Boot Sequence Current Profile for no PC and no EEPROM (P31 held low and P29 held high).

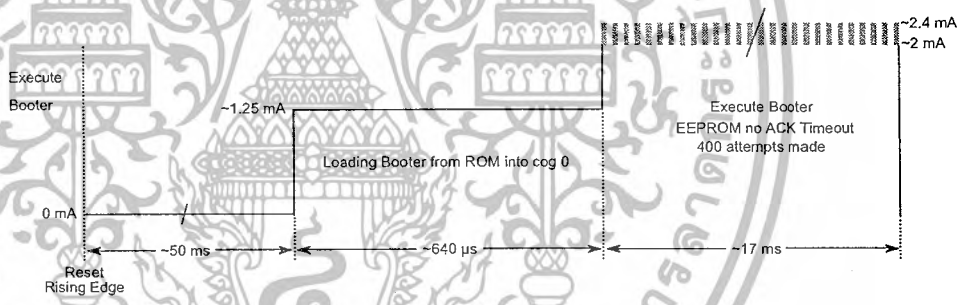


Figure 10

Boot Sequence Current Profile for no PC and EEPROM (P31 held low and P29 connected to EEPROM SDA).

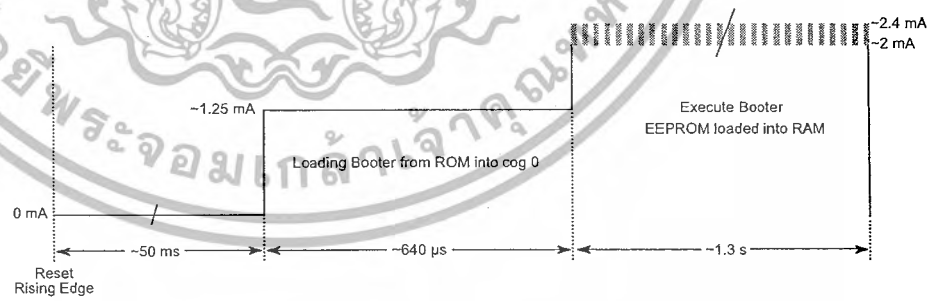
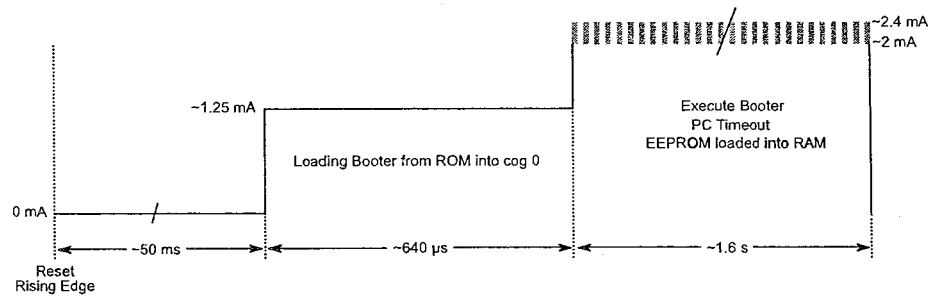


Figure 11

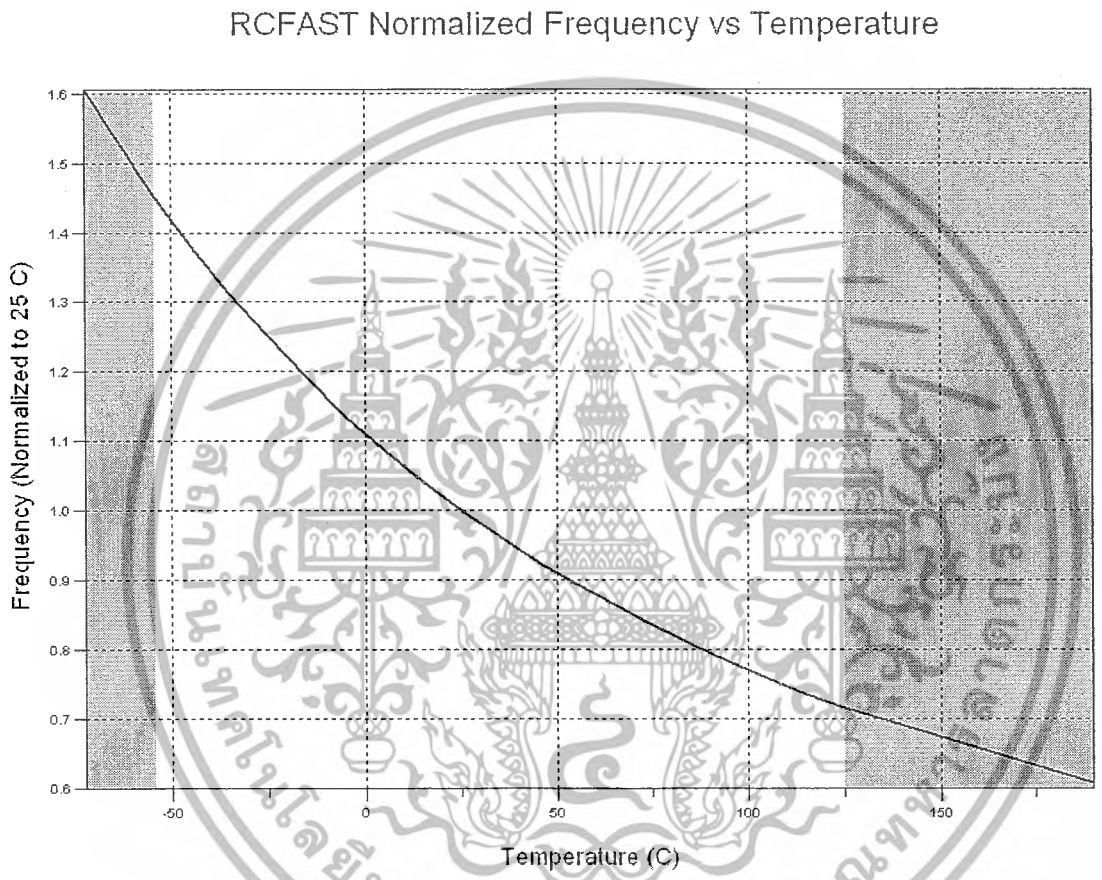
Boot Sequence Current Profile for PC (connected but idle) and EEPROM (P31 held high and P29 connected to EEPROM SDA).



10.0 TEMPERATURE CHARACTERISTICS

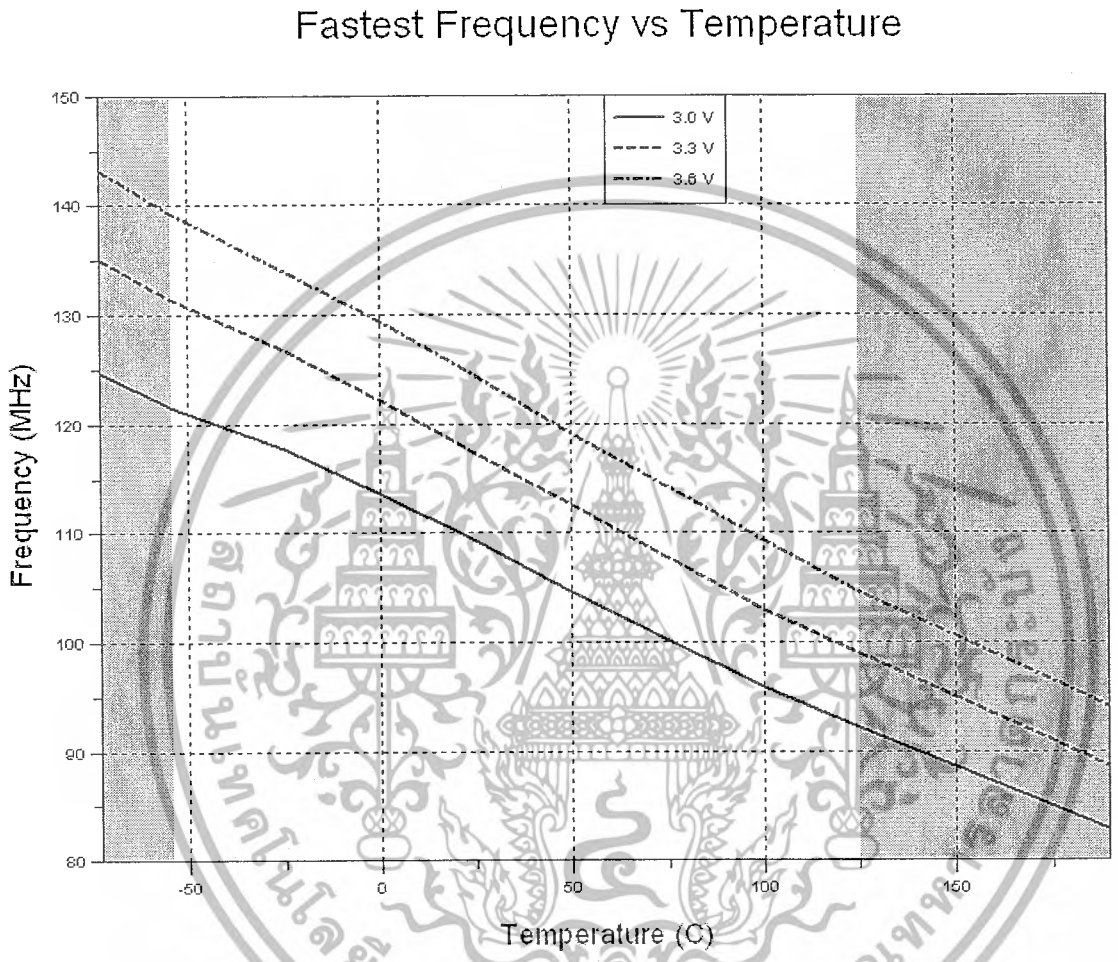
10.1. Internal Oscillator Frequency as a Function of Temperature

While the internal oscillator frequency is variable due to process variation, the rate of change as a function of temperature when normalized provides a chip invariant ratio which can be used to calculate the oscillation frequency when the ambient temperature is other than 25 °C (the temperature to which the graph was normalized). The absolute frequency at 25 °C varied from 13.26 to 13.75 MHz in the sample set. The section of the graph which has a white background is the military range of temperature; the sections in grey represent data which is beyond military temperature specification.



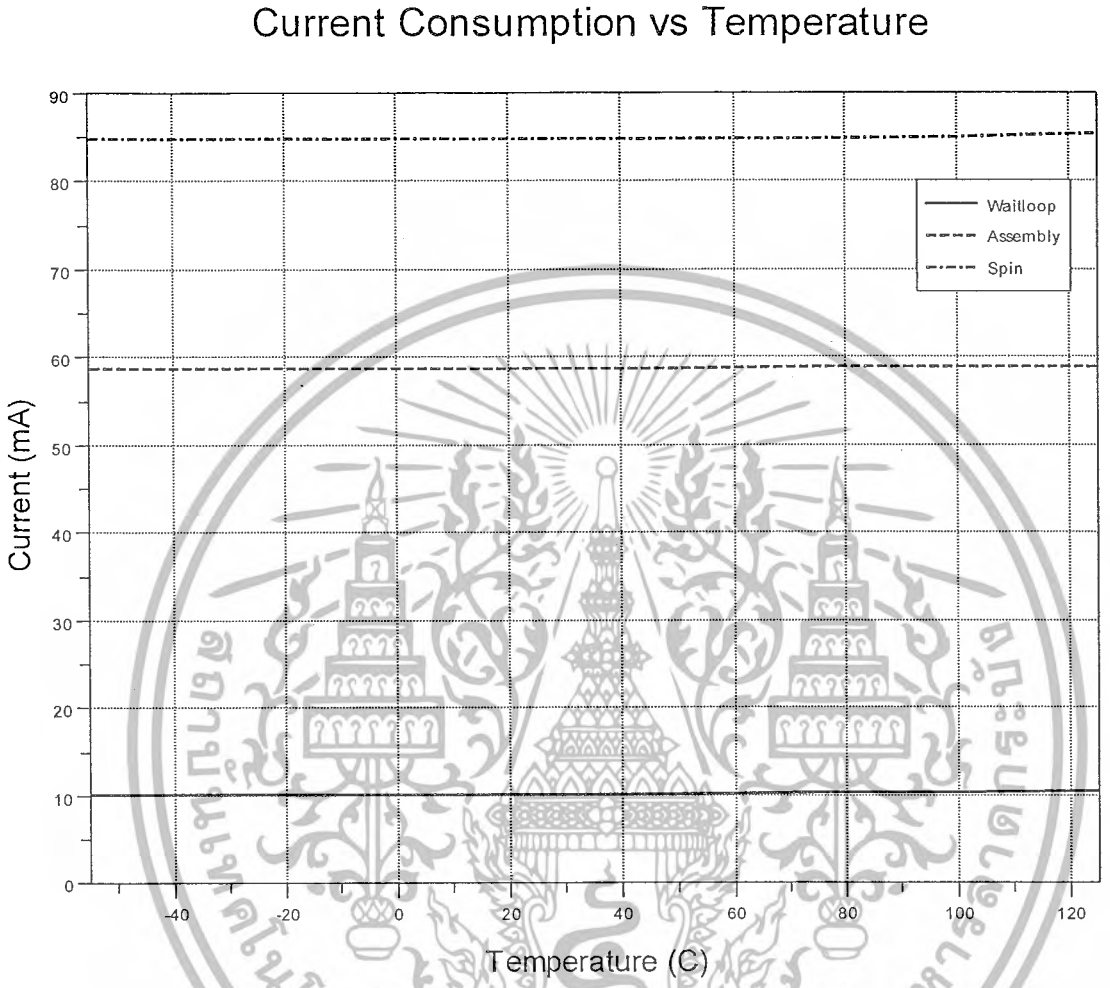
10.2. Fastest Operating Frequency as a Function of Temperature

The following graph represents a small sample average of a Propeller chip's fastest operating range. The test was performed in a forced air chamber using code run on all eight cogs, multiple video generators, and counter modules. A frequency was considered successful if the demo ran without fault for one minute. The curves represent an aggressive testing procedure (averaged, forced air, one minute time limit); therefore the designer must de-rate the curve to arrive at a stable frequency for a particular application. Again the grayed regions represent temperatures beyond the military temperature range.



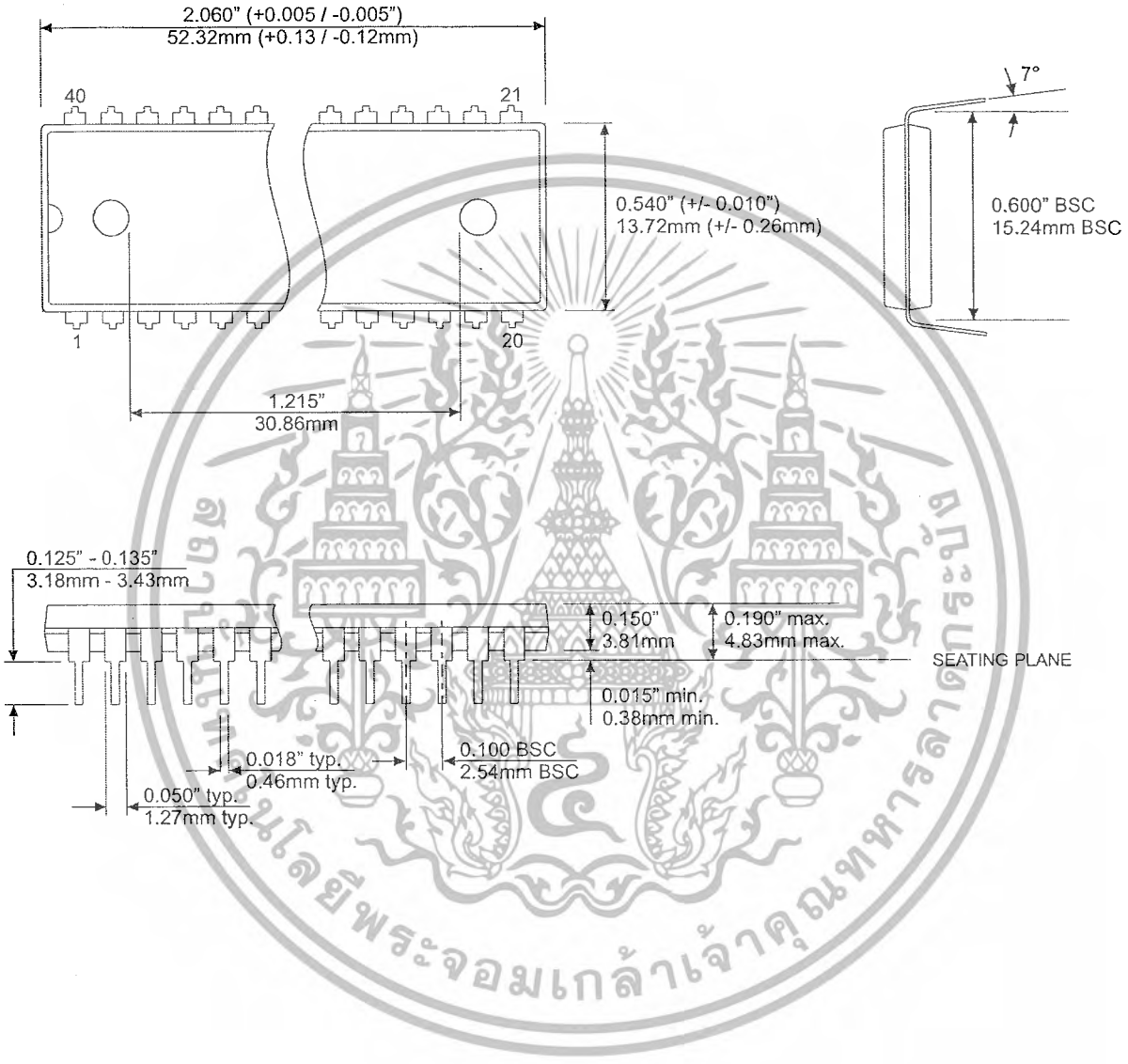
10.3. Current Consumption as a Function of Temperature

The following graph demonstrates the current consumption of the Propeller as a function of temperature. It is clear from the graph that current consumption is nearly independent of temperature over the entire military temperature range.

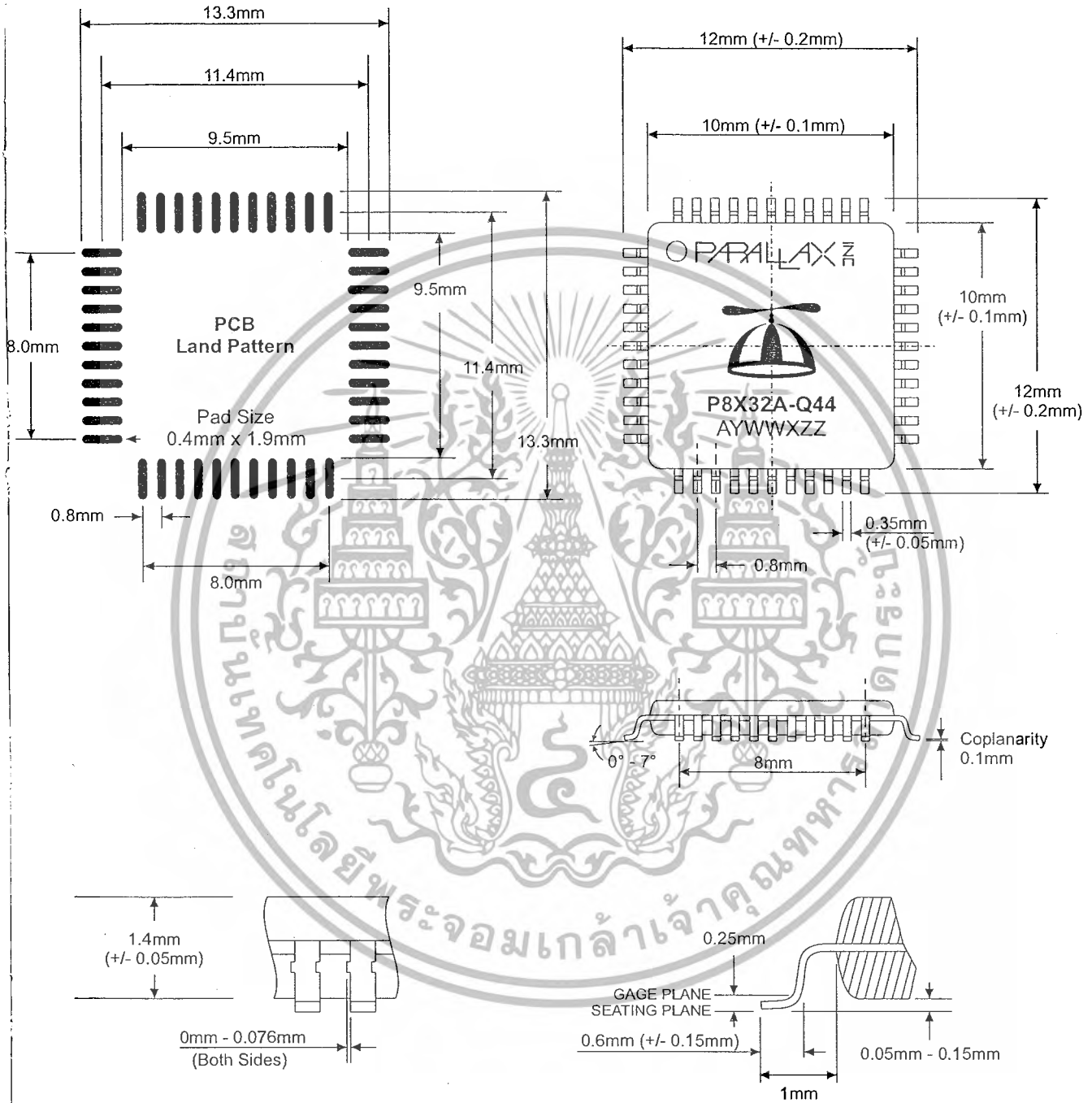


11.0 PACKAGE DIMENSIONS

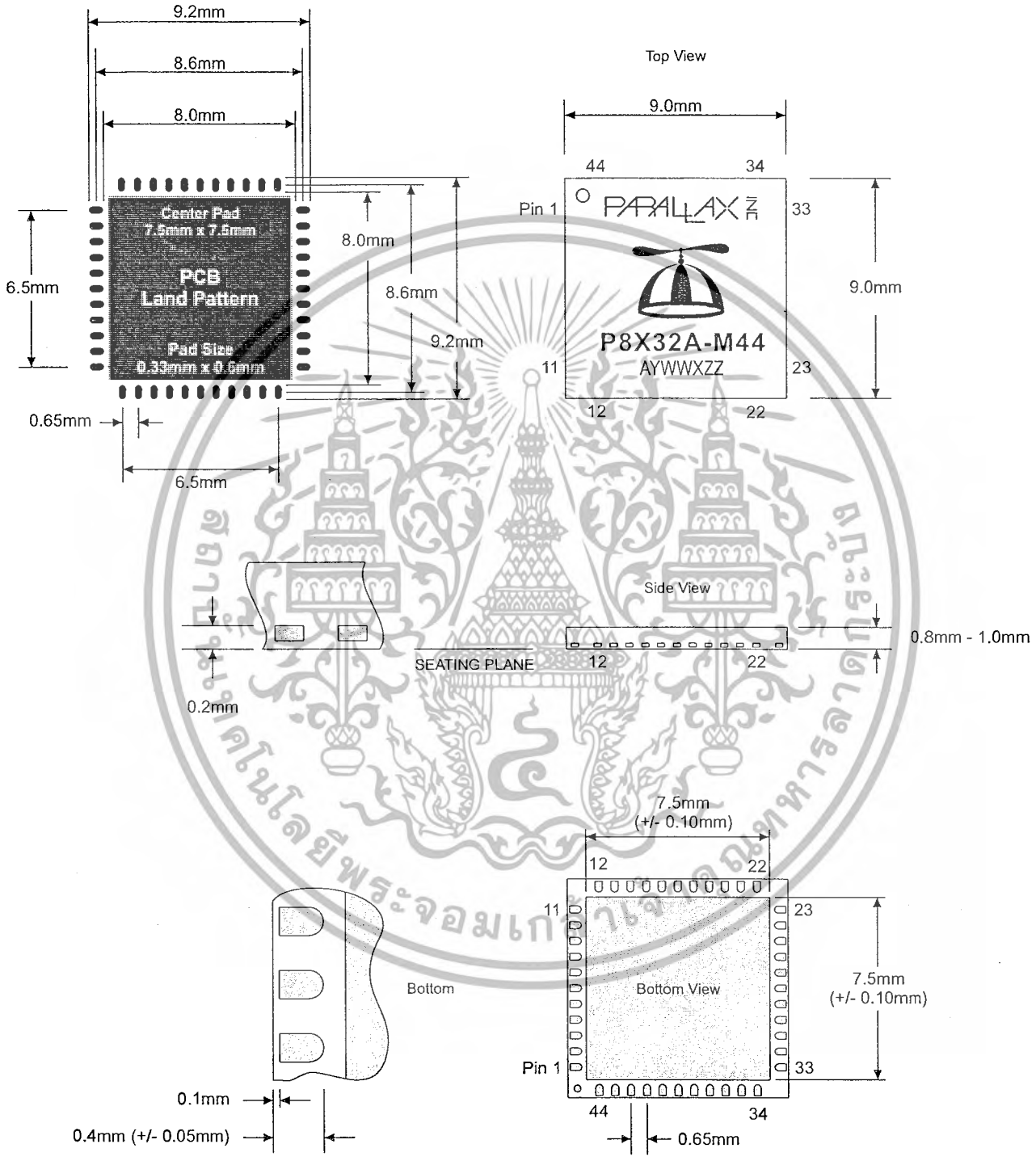
11.1. P8X32A-D40 (40-pin DIP)



11.2. P8X32A-Q44 (44-pin LQFP)



.3. P8X32A-M44 (44-pin QFN)



12.0 MANUFACTURING INFORMATION

12.1. Reflow Peak Temperature

Package Type	Reflow Peak Temp.
DIP	255+5/-0 °C
LQFP	255+5/-0 °C
QFN	255+5/-0 °C

12.2. Green/RoHS Compliance

All Parallax Propeller chip models are certified Green/RoHS Compliant. The Certificate of Compliance is available upon request and may be obtained by contacting the Parallax Sales Team.

Parallax Sales and Tech Support Contact Information

For the latest information on Propeller chips and programming tools, development boards, instructional materials, and application examples, please visit www.parallax.com/propeller.

Parallax, Inc.
599 Menlo Drive
Rocklin, CA 95765

Sales/Tech Support: (916) 624-8333
Toll Free in the Continental US: 1-888-512-1024
Sales: sales@parallax.com
Tech Support: support@parallax.com