การศึกษาภาษาบล็อก

STUDY ON BLOCK LANGUAGE

นางสาวชณัฏฐา เอี่ยมทศ
นายชัยวัฒน์ เดชคุณธรรม
นางสาวฐาปนี วงศ์กาฬสินธุ์

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต
สาขาวิชาวิศวกรรมการวัดคุม
ภาควิชาวิศวกรรมการวัดคุม คณะวิศวกรรมศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ปีการศึกษา 2550

STUDY ON BLOCK LANGUAGE

CHANATTHA IAMTOT

CHAIWAT DETKUNNATHAM

THAPANEE WONGKALASIN

A THESIS SUBMITTED IN PARTAIL FULFILLMENT

OF THE REQUIRMENT FOR THE DEGREE OF BACHELOR OF ENGINEERING IN

INSTRUMENTATION ENGINEERING

FACULTY OF ENGINEERING

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2007

ภาควิชาวิศวกรรมการวัดคุม

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ใบรับรองปริญญานิพนธ์

.....

หัวข้อปริญญานิพนธ์ การศึกษาภาษาบล็อก

STUDY ON BLOCK LANGUAGE

นักศึกษาผู้จัดทำ	นางสาวชณัฏฐา เอี่ยมทศ	รหัสนักศึกษา 47010134
	นายชัยวัฒน์ เดชคุณธรรม	รหัสนักศึกษา 47010172
	นางสาวฐาปนี วงศ์กาฬสินธุ์	รหัสนักศึกษา 47010201
ปริญญา	วิศวกรรมศาสตรบัณฑิต	
สาขาวิชา	วิศวกรรมการวัดคุม	
ปีการศึกษา	2550	

อาจารย์ผู้ควบคุมปริญญานิพนธ์	ลายมือชื่อ
รศ. ประภาษ อุดคกิมาพันธุ์ ผศ. พิทยา ปานนิล อาจารย์กฤษณ์ เสมอพิทักษ์	

ภาควิชารับรองแล้ว

(รศ. ประภาษ อุดคกิมาพันธุ์)

หัวหน้าภาควิชาวิศวกรรมการวัดคุม

หัวข้อปริญญานิพนธ์ การศึกษาภาษาบล็อก
STUDY ON BLOCK LANGUAGE

นักศึกษาผู้จัดทำ นางสาวชณัฏฐา เอี่ยมทศ รหัสนักศึกษา 47010134
นายชัยวัฒน์ เดชคุณธรรม รหัสนักศึกษา 47010172
นางสาวฐาปนี วงศ์กาฬสินธุ์ รหัสนักศึกษา 47010201

อาจารย์ที่ปรึกษา รศ. ประภาษ อุดคกิมาพันธุ์
ผศ. พิทยา ปานนิล
อาจารย์ กฤษณ์ เสมอพิทักษ์

ปีการศึกษา 2550

บทคัดย่อ

ในปัจจุบันนี้มีการนำไมโครคอนโทรลเลอร์มาประยุกต์ใช้งาน เพื่อควบคุมการทำงานของระบบที่ต้องการ โดยการเขียนโปรแกรมที่เป็นลำดับของคำสั่ง ของไมโครคอนโทรลเลอร์ ซึ่งแต่ละโปรแกรมที่ใช้พัฒนาตัวไมโครคอนโทรลเลอร์ จะมีวิธีการเขียนที่แตกต่างกันตามชนิดของไมโครคอนโทรลเลอร์ ดังนั้นโครงงานนี้จึงได้สร้างภาษาบล็อกสำหรับควบคุมการทำงานของไมโครคอนโทรลเลอร์ โดยสร้างชุดคำสั่งแยกเป็นส่วน ๆ ซึ่งแต่ละชุดคำสั่ง สามารถควบคุมส่วนต่างๆภายในไมโครคอนโทรลเลอร์ได้ ดังนั้นเมื่อนำชุดคำสั่งต่าง ๆ มาจัดลำดับการทำงานให้ต่อเนื่อง ตามที่ผู้ใช้ออกแบบในโปรแกรมประยุกต์ที่ถูกสร้างขึ้นเพื่อเชื่อมต่อกับไมโครคอนโทรล-เลอร์ก็จะทำให้ผู้ใช้งานสามารถเข้าใจและใช้งานได้อย่างไม่ยุ่งยากซับซ้อน

Thesis Title	Study on Block Language	
Authors	Miss Chanattha	Iamtot
	Mr. Chaiwat	Detkunnatham
	Miss Thapanee	Wongkalasin
Thesis Advisor	Assoc.Prof. Prapart	Ukkakimapan
	Asst.Prof. Pittaya	Pannil
	Mr. Krit	Samerpitak
Year	2007	

ABSTRACT

Nowaday we apply the microcontroller to control the operation system by writing the sequence of command programs into the microcontroller. Each developed program for microcontroller will have the difference method of writing the program which depended on the kinds of microcontroller. So that this project builds the Block Language to control the operation of microcontroller by building the group of command and separate each part of it then the group of command can control all the parts of microcontroller. When take the group of command to rearrange continuous operation as designed by user for connecting with the microcontroller. User could be understanded and used them without complicated.

กิตติกรรมประกาศ

ผู้เขียนขอขอบพระคุณ รศ.ประภาษ อุดคกิมาพันธุ์ เป็นอย่างสูง ที่ได้ให้ความกรุณาสละเวลารับเป็นอาจารย์ที่ปรึกษาในปริญญานิพนธ์เล่มนี้ ซึ่งได้ให้คำแนะนำถึงประเด็นต่าง ๆ ในการศึกษาและชี้แนวทางในการแก้ปัญหา การค้นคว้าหาข้อมูลเพิ่มเติม อันเป็นประโยชน์ในการวิเคราะห์และสรุปผลการศึกษา รวมทั้งการแก้ไขงานให้สมบูรณ์เป็นอย่างยิ่ง นอกจากนี้ต้องขอขอบพระคุณ อาจารย์ กฤษณ์ เสมอพิทักษ์ ที่ได้กรุณาสละเวลาอันมีค่ารับเป็นอาจารย์ที่ปรึกษาปริญญานิพนธ์เรื่องนี้ และได้ให้ความกรุณาในการแนะนำและให้คำปรึกษาในการปรับปรุงแก้ไขปริญญานิพนธ์ให้สำเร็จลุล่วงไปด้วยความเรียบร้อย

ขอขอบคุณเจ้าหน้าที่ในภาควิชาวิศวกรรมการวัดคุม ที่กรุณาให้ความสะดวกในการทำปริญญานิพนธ์ ขอบคุณห้องสมุดต่าง ๆ ซึ่งเป็นแหล่งข้อมูลที่สำคัญในการเรียนและทำงานวิจัยฉบับนี้ ขอขอบคุณผู้ให้ข้อมูลทุกท่าน ทั้งจากภาควิชาวิศวกรรมคอมพิวเตอร์ วิศวกรรมอิเล็คทรอนิคส์ วิศวกรรมคอนโทรล และคณะครุศาสตร์อุตสาหกรรม สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ขอบคุณบุคคลต่าง ๆ ที่มิอาจกล่าวนามได้หมด ณ ที่นี้ ขอขอบคุณแรงสนับสนุนและกำลังใจที่ได้รับจากครอบครัวตลอดจนเพื่อน ๆ จนทำให้ปริญญานิพนธ์ฉบับนี้เสร็จสมบูรณ์ลงได้

สุดท้ายนี้ หากปริญญานิพนธ์ฉบับนี้จะก่อให้เกิดประโยชน์ต่อผู้สนใจในการศึกษาและพัฒนาภาษาบล็อก นับเป็นความปีติอย่างยิ่งที่ได้ทำปริญญานิพนธ์เล่มนี้ขึ้น และหากมีข้อผิดพลาดประการใด ผู้จัดทำขออภัยไว้ ณ ที่นี้

คณะผู้จัดทำ

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

ตารางที่ **หน้า**

สารบัญรูป

รูปที่ **หน้า**

สารบัญรูป (ต่อ)

รูปที่ **หน้า**

บทที่ 1
บทนำ

1.1 ความสำคัญของปริญญานิพนธ์

โครงงานนี้ได้นำเสนอการทำงานของโปรแกรมที่ถูกเขียนขึ้นมา เพื่อใช้ในการควบคุมการทำงานของไมโครคอนโทรลเลอร์ที่เรียกว่า ภาษาบล็อก ซึ่งภาษาบล็อกได้มีการนำหลักการคิดเป็นลำดับมาจากแผนภาพบล็อกไดอะแกรมโดยในแต่ละบล็อกจะมีขอบเขตการทำงานที่แตกต่างกัน ซึ่งในการทำงานของภาษาบล็อกจะเป็นการนำบล็อกแต่ละบล็อกมาวางต่อกันตามกระบวนการควบคุมที่ได้ออกแบบไว้ตามความต้องการ หลังจากนั้นคำสั่งทั้งหมดจะถูกส่งไปยังไมโครคอนโทรลเลอร์ที่ใช้ตัวประมวลผลตระกูล dsPIC โดยในโครงงานนี้ได้เลือกใช้ไมโครคอนโทรลเลอร์ dsPIC30F6010A เพื่อใช้รองรับการทำงานของภาษาบล็อกโดยใช้โปรแกรม Visual Basic สร้างหน้าต่างที่ประกอบไปด้วยบล็อกคำสั่ง เพื่อให้ผู้ใช้งานเลือกลำดับการทำงานของไมโครคอนโทรลเลอร์และใช้สำหรับแสดงผลหรือแก้ไขการทำงาน

1.2 วัตถุประสงค์ของปริญญานิพนธ์

1. เพื่อศึกษาวิธีการคิดเป็นขั้นตอน โดยเชื่อมโยงการลำดับความคิดโดยออกมาทางรูปแบบแผนภาพบล็อกไดอะแกรมที่เชื่อมโยงและเรียงลำดับกันเป็นกระบวนการอย่างง่าย
2. เพื่อสร้างเครื่องมือในการควบคุมที่ใช้งานง่ายและผู้ใช้สามารถศึกษาเพื่อให้เข้าใจได้ง่าย
3. เพื่อเป็นทางเลือกอีกทางเลือกหนึ่งของผู้ใช้งาน

1.3 ขอบเขตของปริญญานิพนธ์

ทางคณะผู้จัดทำโครงงานได้แบ่งส่วนงานออกเป็น 2 ส่วนดังนี้

ส่วนที่ 1 ส่วนของหน้าจอ User Interface สำหรับผู้ใช้งานเพื่อใช้ในการออกแบบระบบที่ต้องการควบคุมการทำงานของไมโครคอนโทรลเลอร์

ส่วนที่ 2 ส่วนของไมโครคอนโทรลเลอร์ dsPIC30F6010A

เนื่องจากผู้ใช้งานจะต้องควบคุมการทำงานของไมโครคอนโทรลเลอร์ โดยผ่านทางUser Interface ดังนั้นโค้ดในการเขียนโปรแกรมของทั้งสองส่วนจะต้องมีความสัมพันธ์กัน

บทที่ 2
ทฤษฎีที่เกี่ยวข้อง

2.1 โปรแกรมวิชวล เบสิก (Visual Basic)

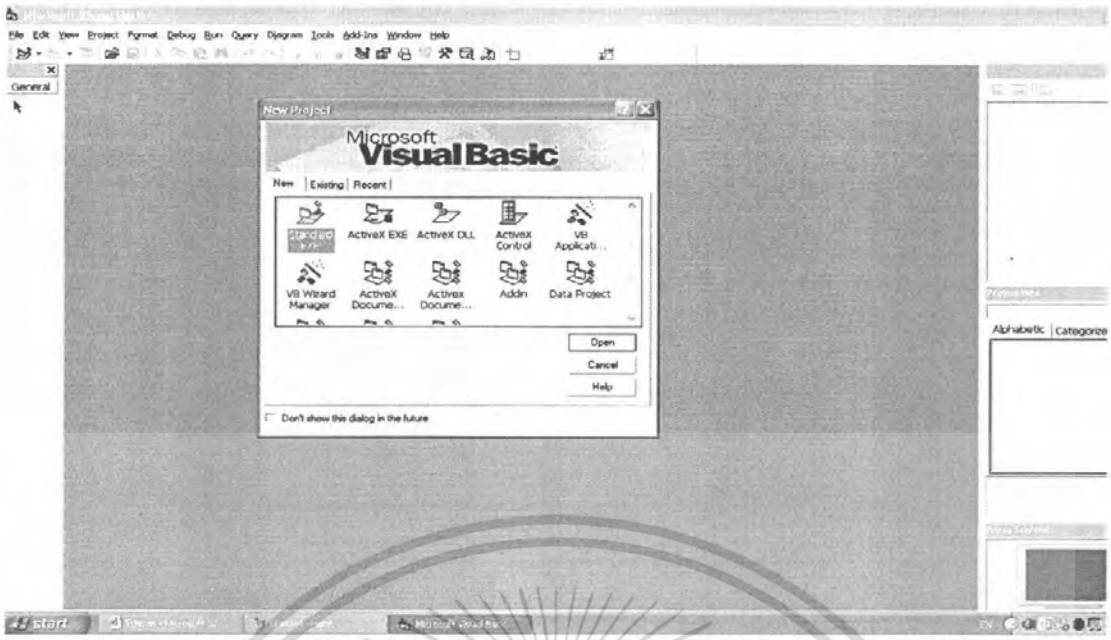
โปรแกรมวิชวล เบสิกเป็นโปรแกรมที่ใช้กันอย่างกว้างขวางในปัจจุบัน สามารถรองรับกับโปรแกรมต่าง ๆ ของบริษัทไมโครซอฟท์ได้เป็นอย่างดี แม้แต่ในไมโครซอฟท์ออฟฟิศเองก็ยังต้องใช้วิชวล เบสิกสำหรับเขียนโปรแกรมต่อเติมเพื่อการทำงานในระดับสูง หรือแก้ไขส่วนที่บกพร่องเล็ก ๆ น้อย ๆ ของโปรแกรม

ในส่วนของภาษาที่ใช้ในการเขียนโปรแกรมนั้น ในด้านกราฟฟิกพัฒนามาจากภาษาเบสิก (Basic) ซึ่งเป็นภาษาที่มีมาตั้งแต่ดั้งเดิมในสมัยที่ยังใช้ดอส(Dos) ในการพัฒนาโปรแกรมในดอนนั้นมีภาษาเบสิกหลายตัว อาทิ QBasic , MBasic เป็นต้น

ส่วนที่สำคัญที่ทำให้วิชวล เบสิกกลายมาเป็นโปรแกรมที่นิยมใช้กันมากในปัจจุบันก็เพราะตัวภาษาเบสิกถูกนำมาใช้เขียนโปรแกรมวินโดว์ ทำให้ผู้ใช้สามารถใช้งานโปรแกรมได้อย่างสะดวกและมีความหลากหลายมากขึ้น

ในโครงงานนี้เลือกใช้โปรแกรมวิชวล เบสิกเวอร์ชั่น 6.0 ซึ่งเป็นเวอร์ชั่นที่นิยมใช้กันมากในปัจจุบัน ทั้งในงานด้านกราฟฟิกต่าง ๆ รวมไปถึงการนำมาใช้ในการควบคุมในงานอุตสาหกรรมประเภทต่าง ๆ อีกด้วย

นอกจากนั้นยังรวมไปถึงแหล่งความรู้ทางด้านบุคคล เอกสารหรือหนังสือ ซึ่งมีอยู่มากมายสำหรับเวอร์ชั่น6.0 ทำให้เมื่อเกิดปัญหาจะสามารถได้โดยไม่ลำบากนัก รวมถึงการเพิ่มทักษะในการเขียนโค้ดต่าง ๆ ให้งานออกมาดียิ่งขึ้น



รูปที่ 2.1 หน้าต่างพื้นฐานของวิชวล เบสิก ให้เลือกโหมดการทำงาน

รูปที่ 2.2 หน้าต่างแสดงอุปกรณ์และเครื่องมือพื้นฐานของวิชวล เบสิก

2.1.1 คุณสมบัติเด่นของวิชวล เบสิก

นอกจากความง่ายและความสะดวกสบายในการใช้ และศึกษาโปรแกรมแล้ว วิชวล เบสิก มีข้อดีพอสรุปได้ดังนี้

1. ใช้เขียนโปรแกรมได้ทั้งวินโดว์และเว็บวิชวล เบสิก ในเวอร์ชั่นหลังๆ ได้รับการพัฒนาให้สามารถเขียนโปรแกรมให้ทำงานได้ทั้งบนวินโดว์และบนอินเตอร์เน็ต โดยที่ผู้สนใจเขียนโปรแกรมบนวินโดว์สามารถประยุกต์ใช้ความรู้ที่มีอยู่ในการพัฒนาเว็บแอพพลิเคชันได้ง่าย โดยสามารถเรียนรู้ซึ่งปืนภาษาวิชวล เบสิกได้อย่างรวดเร็ว

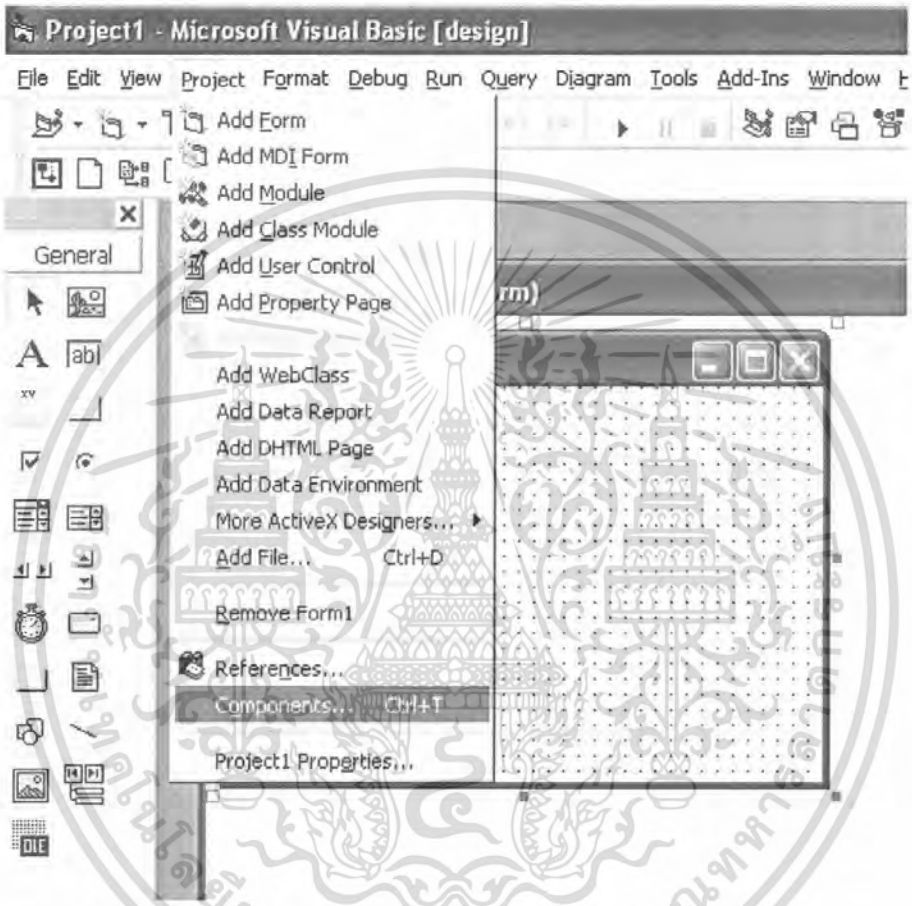
2. แก้ไขโปรแกรมที่เขียนขึ้นได้ง่าย ในบางครั้งการเขียนโปรแกรมอาจจะมีข้อผิดพลาด การแก้ไขหรือปรับปรุง ในภายหลังวิชวล เบสิกนั้นทำได้ง่ายโดยมีเครื่องมือต่างๆ คอยช่วยเหลืออยู่

3. มีเอกสารให้ศึกษาอ้างอิงมาก จากที่กล่าวมาแล้วว่าโปรแกรมวิชวล เบสิกเป็นที่นิยมใช้มากในปัจจุบัน เมื่อมีปัญหาหรือข้อสงสัยเกิดขึ้นจะหาคำตอบได้ง่ายจากผู้มีความรู้รวมทั้งเอกสารในรูปแบบต่าง ๆ รวมถึงจากเว็บไซท์ของไมโครซอฟท์และเว็บไซท์อื่นๆ

4. เป็นโปรแกรมที่มีอนาคต วิชวล เบสิกเป็นภาษาเขียนโปรแกรมที่มีไมโครซอฟท์ถือว่าเป็นภาษาที่สำคัญมากจึงถูกพัฒนามาอย่างต่อเนื่อง คาดว่าเวอร์ชั่นใหม่ ๆ จะถูกเพิ่มความสามารถอย่างมาก ทั้งตัวภาษาและตัวผลงานที่ได้จากการเขียนโปรแกรม

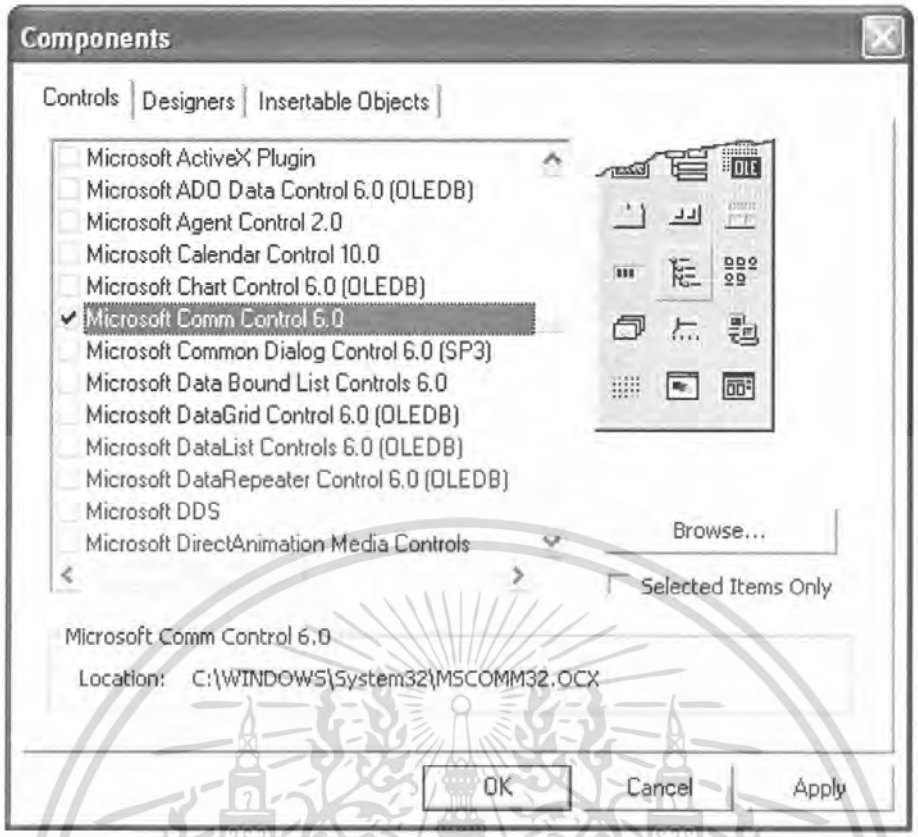
2.1.2 การเขียนโปรแกรมติดต่อและควบคุม Serial Port ด้วย Visual Basic

คอนโทรลที่สำคัญในการทำให้ Visual Basic สามารถสื่อสารผ่านพอร์ตอนุกรมได้นั้นก็คือคอนโทรล MSCommในการใช้งานคอนโทรล MSComm นั้นเราจะต้องทำการเพิ่มคอนโทรลนี้เข้าไปใน ToolBox ของโปรแกรม Visual Basic ซึ่งสามารถกระทำได้โดยคลิกที่เมนู Project แล้วเลือก Component ดังรูป



รูปที่ 2.3 เพิ่มคอมโพเนนต์ MSComm

จากนั้นจะปรากฏไดอะล็อก Components ขึ้นมา จากนั้นให้คลิกเลือกที่ Microsoft Comm Control 6.0 แล้วคลิกที่ปุ่ม OK เมื่อคลิกที่ปุ่ม OK



รูปที่ 2.4 ไดอะล็อก Components เลือกที่รายการ MSComm

จากนั้นก็ปรากฏภายใน Toolbox จะมีไอคอนรูปโทรศัพท์ ซึ่งเป็นไอคอนของคอนโทรล MSComm ปรากฏขึ้นมาให้เราเลือกใช้งาน

รูปที่ 2.5 คอนโทรล MSComm พร้อมทำงาน

การกำหนดคุณสมบัติ (Property) ที่สำคัญในการใช้งาน MSComm ให้สามารถต่อกับพอร์ตได้

1. Propertyชื่อ CommPortใช้ในการกำหนดหมายเลขของพอร์ตอนุกรมที่เราต้องการติดต่อ

ตัวอย่าง ถ้าเรากำหนดให้การเขียนโปรแกรมติดต่อกับพอร์ต Com1 จะเขียนเป็น

MSComm1.CommPort = 1

2. Property ชื่อ Settings คือการตั้งค่าของการรับส่งข้อมูล ซึ่งจะต้องรู้ด้วยว่าอัตราบอดของอุปกรณ์ที่จะติดต่อด้วยเป็นเท่าไร โดยมีรายละเอียดการใส่ต่างๆค่าดังนี้ กำหนดอัตรา Baud Rate หรือความเร็วในการส่งข้อมูล มีหน่วยเป็นบิตต่อวินาที,พาริตี้,จำนวนของบิตข้อมูล,จำนวนของบิตปิดท้าย

ตัวอย่าง เรากำหนดให้มีการเขียนโปรแกรมใช้งานที่ Baud Rate = 9600 บิตต่อวินาที ไม่มีพาริตี้ จำนวนบิตข้อมูลเท่ากับ 8 บิต และมีบิตปิดท้าย 1 บิต

MSComm1.Settings = "9600, N, 8, 1"

3. Property ชื่อ PortOpen ใช้สำหรับเปิดและปิดการใช้งานพอร์ตอนุกรม

ตัวอย่าง เราจะเปิดใช้งานพอร์ตอนุกรม ให้กำหนดค่า Value เป็น True เขียนโค้ดได้ดังนี้

MSComm1.PortOpen = True

แต่ถ้าต้องการปิดพอร์ตอนุกรม ให้กำหนดค่า Value เป็น False

MSComm1.PortOpen = False

2.2 ความหมายของ OOP

OOP (ย่อจาก Object Oriented Programming การเขียนโปรแกรมแบบวัตถุวิธี) เป็นรูปแบบ (Paradigm) หรือแนวคิดอย่างหนึ่ง อันมีจุดมุ่งหมายเพื่อการสร้างซอฟท์แวร์ คำว่า Object ในที่นี้หมายถึงวัตถุหรือสิ่งของที่จับต้องได้ ไม่ได้หมายถึงวัตถุประสงค์ จุดมุ่งหมายหรือกรรม (ผู้ถูกกระทำอย่างประธาน กริยา และกรรม ในรูปประโยค)

มีผู้ตีความว่าแนวคิดนี้คือ"การเขียนโปรแกรมโดยมุ่งที่เป้าหมาย (มิได้เน้นที่กระบวนการ)" ผู้เขียนเห็นว่าการตีความดังกล่าวไม่สู้จะถูกต้อง เพราะการเขียนโปรแกรมตามลัทธินี้มีการสร้าง object ขึ้นจริง แม้จะเป็นobjectอันจับต้องไม่ได้เพราะอยู่ในสภาพซอฟท์แวร์ แต่ก็มีเจตนาจะเลียนแบบ object ที่เป็นรูปธรรมอย่างเต็มที่

เรื่อง OOP เป็นเรื่องของ object ดังนั้นสิ่งแรกที่ต้องรู้คือความหมายของคำว่า object นิยามของ object คือ "หน่วยหนึ่งของโปรแกรมซึ่งมีหน้าที่การทำงานอันเฉพาะเจาะจง และถูกกำหนดปฏิสัมพันธ์กับโปรแกรมหน่วยอื่น ๆ ไว้อย่างแน่ชัด" ในภาษา C# เราสร้าง object จากคลาส

2.2.1 Class กับ Object

คลาสกับออบเจ็กเป็นสองสิ่งที่มีความเกี่ยวข้องกันโดยตรง คลาสคือโค้ดที่เราเขียนขึ้นเพื่อทำหน้าที่เป็นพิมพ์เขียวของ object การสร้าง object จากคลาสเรียกว่าการทำ instantiation การสร้าง object จากคลาสเทียบได้กับการทำขนมครกสิงคโปร์ เตาขนมจะมีหลุมหลายหลุม แต่ละหลุมมีลวดลายไม่เหมือนกัน ขนมครกที่ได้จากแต่ละหลุมจึงมีรูปร่างต่างๆ กัน คลาสคือหลุมหนึ่งหลุม ขนมครกที่ได้คือ instance ของ object เราเรียก object หนึ่ง object ว่าหนึ่ง instance เราสามารถสร้าง object ได้หลาย ๆ instance จากคลาสเพียงคลาสเดียว จากตัวอย่างขนมครกสิงคโปร์ในวันหนึ่ง ๆ แม่ค้าจะทำขนมครกได้เป็นจำนวนมากจากหลุมแต่ละหลุม ส่วนในภาษา C# หากเราสร้างคลาสหนึ่งคลาส ยกตัวอย่างเช่นเป็นคลาสเพื่อนิยาม node หลังจากนั้นเราอาจเขียนโปรแกรมสร้าง binary tree ซึ่งขณะทำงานมันอาจจะสร้าง object จากคลาส node ได้หลายล้าน instance ภายในหนึ่งวินาที

การใช้งานคลาสทำได้สองวิธี วิธีแรกคือการนำไปใช้สร้าง object และใช้งานผ่าน object ดังที่อธิบายไปแล้ว อีกวิธีหนึ่งคือเรียกใช้โดยตรงโดยไม่ต้องสร้าง object เรียกว่า static class แม้การใช้งานคลาสวิธีนี้จะไม่มี object ทำให้ดูเหมือนไม่เป็นหลักการ OOP แต่ในบางสถานการณ์ก็ถือว่ามีความเหมาะสม

2.2.2 Object กับ Type

เรื่องของ type เป็นเรื่องสำคัญในOOPจึงพบคำว่าtypeอยู่เสมอ จึงมีความจำเป็นที่จะต้องทำความเข้าใจความหมายของคำว่าtypeให้ตรงกับความหมายใน OOP และภาษา C# ก่อน ในภาษาที่ไม่ใช่ภาษา OOP อย่างภาษา C มี type ต่าง ๆ เตรียมไว้ล่วงหน้า เช่น int (จำนวนเต็ม) float (จำนวนมีทศนิยม) Char (ตัวอักษร) เป็นต้น type เหล่านี้คือชนิดของข้อมูล ยกตัวอย่างเช่นเมื่อเราเขียนโค้ดว่า int foo; ซึ่งทำหน้าที่นิยามตัวแปรชนิดจำนวนเต็ม Compiler (ตัวแปลภาษา) จะรู้ทันทีว่าควรทำเช่นใดกับตัวแปรชนิดนี้ เพราะ int เป็น type ที่ถูกกำหนดไว้ล่วงหน้าแล้ว (ภายในตัว Compiler ภาษา C) เรียกว่า build-in type (Primitive type หรือ Abstract data type)

ภาษา C รับรู้ type อยู่จำนวนหนึ่ง หากเราพบว่า type ที่มีมาให้ไม่สามารถตอบสนองต่อความต้องการของเราได้ เราสามารถนิยาม type ขึ้นเองได้เรียกว่า User Defined Data Type (ต่อไปจะเรียกย่อว่า UDT) โดยใช้ Structure

คลาสก็เหมือน Structure เรานิยามคลาส เพราะเราต้องการสร้าง UDT ขอใช้อุปมาขนมครกสิงค์โปรอีกครั้ง ขนมครกสิงค์โปรหนึ่งเตาจะมีหลายหลุม แต่ละหลุมมีลวดลายต่างกัน เช่นหลุมหนึ่งมีลายเป็นปลา อีกหลุมหนึ่งมีลายเป็นหอย ขนมครกที่ได้จากหลุมแต่ละหลุมจึงมีรูปร่างต่างกันหรือมี type ต่างกัน สิ่งที่ทำให้ type ของขนมครกแตกต่างกันคือตัวหลุม

หลุมแต่ละหลุมบนขนมครกสิงค์โปรเทียบได้กับคลาสหนึ่งคลาส จุดมุ่งหมายในการนิยามคลาสก็เหมือนStructureคือเราต้องการสร้าง type ใหม่ object ที่ถูกสร้างจากคลาสจะมี type ตามที่คลาสนั้นได้นิยามไว้

ภาษา C# สนับสนุนการสร้าง type ใหม่อย่างพิสดาร เมื่อนิยามแล้ว UDT กับ build-in type จะมีศักดิ์ศรีเสมอกัน (คือ compiler จะปฏิบัติต่อ type ที่เราสร้างในลักษณะ first class เช่นเดียวกับ build-in type).NET Framework libraryเองก็เป็นแหล่งรวม type หลายพัน type ที่เราสามารถนำมาประกอบเป็นtypeใหม่ที่มีการทำงานซับซ้อนได้อย่างสะดวกรวดเร็ว การเขียนโปรแกรมแบบOOP เราจะใช้เวลาส่วนมากไปกับการออกแบบและนิยาม type ขึ้นใหม่ เพื่อนำไปใช้สร้าง object ต่างๆ เมื่อนำ object ทั้งหมดมาประกอบเข้าด้วยกันเพื่อให้ทำงานร่วมกัน จะทำให้โปรแกรมของเราสร้างผลลัพธ์ได้ตามความประสงค์

2.2.3 การใช้งาน object

การใช้งาน object ต้องทำได้ง่าย เราอาจมองว่า object เป็น"กล่องดำ" ในทางวิทยาศาสตร์เราจะเรียกอุปกรณ์ที่เราสามารถนำมาใช้ประโยชน์ได้ โดยไม่จำเป็นต้องรู้การทำงานภายในของมันว่ากล่องดำ ยกตัวอย่างเช่นโทรศัพท์ เรานำมันมาใช้ประโยชน์เพื่อการสื่อสารได้โดยไม่จำเป็นต้องรู้ว่ามันทำงานได้อย่างไร เช่นเดียวกันในภาษา C# เราสามารถนำ object มาใช้งานได้โดยไม่จำเป็นต้องรู้ว่ามันมี source code ภายในเป็นอย่างไร

เมื่อเราต้องการเรียกให้ object ทำงานบางอย่าง เราจะเรียก method ของมันยกตัวอย่างเช่นโทรศัพท์มี method คือการโทรฯออก การวางสาย การพักสาย การบันทึกเลขหมายฯ ในภาษา C# เราจะนิยาม method เหล่านี้ โดยการเขียนโค้ดหนึ่งชุดคล้ายการนิยามฟังก์ชันในภาษาC object จะผนวกกระบวนการ(method)และข้อมูล(information) ไว้ภายในตัวของมันเอง จากตัวอย่างโทรศัพท์เก็บเลขหมายที่เราบันทึกไว้โดยเราไม่จำเป็นต้องรู้ว่ามันเก็บไว้ที่ไหนอย่างไร เรารู้เพียงวิธีดึงเลขหมายที่บันทึกไว้ออกมาใช้งานก็พอ ในภาษา C# object จะเก็บข้อมูลภายในไว้ใน field (คือตัวแปรท้องถิ่นของคลาส)โปรแกรมที่เรียกใช้ object ไม่จำเป็น(และไม่สามารถ)เข้าถึงหรือเปลี่ยนแปลงข้อมูลใน field ได้โดยตรง แต่สามารถเข้าถึงทางอ้อมได้ผ่านส่วนเชื่อมต่อที่เรียกว่า property

หลักของ OOP จะมีอยู่สามข้อใหญ่ ๆ คือ

1. Encapsulation
2. Inheritance
3. Polymorphic

1. Encapsulation

ว่ากันว่านักเขียนโปรแกรมคนหนึ่ง เมื่ออ่านตำรา OOP เห็นคำว่า encapsulation ก็ไม่เข้าใจ และปวดหัวจึงหยิบยาแก้ปวดมารับประทาน ก่อนรับประทานได้อ่านที่ฉลากเห็นเขียนกำกับว่าให้รับประทานครั้งละสองแคปซูล คือ สองเม็ด ลองแกะแคปซูลออกดูก็เห็นมีผงยาขมอัดแน่นอยู่ เนื่องจากผู้ผลิตยาต้องการให้รับประทานได้โดยไม่ขม จึงทำเยลลาตินเป็นหลอดแคปซูลหุ้มผงยาไว้ นักเขียนโปรแกรมเห็นดังนั้นก็เข้าใจความหมายของ encapsulation และหายปวดหัวโดยไม่ต้องรับประทานยา เขาเข้าใจว่า encapsulation คือการผนึกโค้ดให้เป็นแคปซูลเหมือนแคปซูลยาเพื่อให้ผู้อื่นนำไปใช้ได้โดยง่าย

Encapsulation (อ่านว่า เอ็นแคปซูเลชัน แปลว่า การบรรจุลงในแคปซูล) คือแนวคิดว่า object ควรแยกโค้ด "ส่วนเชื่อมต่อ" (interface)กับโค้ด "ส่วนประมวลผลและข้อมูล" (logic หรือ process และ data หรือ field) ออกจากกัน ข้อมูลและส่วนประมวลผลทั้งหมดควรถูกซ่อนไว้เบื้องหลัง interface หากเราจัดให้มีinterfaceที่ดีและรักษาความคงเส้นคงวาของ interfaceไว้ แม้เราจะเปลี่ยนโค้ดส่วนประมวลผลไปทั้งหมดโปรแกรมประยุกต์ก็ยังสามารถใช้งาน object ได้ตามปกติ ยกตัวอย่างเช่น ผู้เขียนสร้างคลาสชื่อ BCCทำหน้าที่คำนวณผลรวมของข้อมูลเพื่อใช้ตรวจสอบความถูกต้องในการรับ-ส่งข้อมูลระหว่างคอมพิวเตอร์ ผู้นำคลาสนี้ไปใช้ไม่จำเป็นต้องรู้ว่าผู้เขียนใช้วิธีหาค่า (algorithm) อย่างไร ผู้ใช้เพียงสร้าง object จากคลาสนี้และส่งชุดข้อมูลให้มัน มันก็จะส่งผลลัพธ์กลับมาให้ หากผู้เขียนปรับปรุง algorithm เพื่อให้คลาสBCCทำงานมีประสิทธิภาพสูงขึ้น ผู้ใช้ก็ไม่จำเป็นต้องแก้ไขโค้ดแต่อย่างใด

บ่อยครั้งที่การสร้างซอฟท์แวร์ตามวิธี OOP จะได้ซอฟท์แวร์ที่ประกอบด้วย object จำนวนมาก object แต่ละตัวถูกสร้างจากคลาสซึ่งมีที่มาต่างๆ กัน เช่นมาจาก .NET Framework มาจากผู้เขียนเอง มาจากบุคคลที่สาม (thirdparty) การทำให้ object แต่ละตัวเป็นอิสระต่อกัน ในขณะที่สามารถทำงานสอดคล้องกันได้เป็นอย่างดี จำเป็นต้องมีข้อตกลงหรือแนวทางที่ทุกฝ่ายใช้ร่วมกัน encapsulation คือแนวทางหลักอันหนึ่งที่ใช้ในการสร้างซอฟท์แวร์ตามวิธี OOP

2. Inheritance

วิธีง่ายที่สุดเพื่อดูว่าภาษาใดเป็นภาษาที่สนับสนุน OOP หรือไม่ ให้ตรวจสอบว่าภาษานั้นสนับสนุน Inheritance หรือไม่ ถ้าไม่แสดงว่าภาษานั้นไม่เป็น OOP Inheritance (อ่านว่าอินเฮอริ-แตนซ์ แปลว่า "สืบสันดาน" หรือ "การสืบทอดมรดก") เป็นหลักการสำคัญหนึ่งในสามหลักการของ OOP หลักการนี้มีไว้เพื่อให้สามารถต่อยอดงานใหม่จากงานเดิมที่เคยทำไว้แล้ว โดยไม่ต้องเริ่มจากศูนย์

3. Polymorphism

คำจำกัดกอนของOOPที่ผู้คนหวาดกลัวมากที่สุดคือคำว่า polymorphism เพราะมันชวนให้นึกถึงพิธีอันลี้ลับที่ hacker ประกอบในที่รโหฐาน แต่ตรงกันข้ามอันที่จริง polymorphism เป็นสิ่งเรียบง่าย

ที่แม้เด็กเล็กๆก็เข้าใจทุกคน ท่านเคยเห็นเด็กเล่นดินน้ำมันหรือไม่ เด็กชอบดินน้ำมันเพราะนอกจากจะมีสีสันสวยงาม แล้วยังนำมาปั้นให้เป็นรูปทรงอะไรก็ได้ตามใจชอบคำว่า polymorphism (อ่านว่า โพลีมอร์ฟิส) มาจากคำในภาษากรีกสองคำคือ poly (โพลี) แปลว่า หลายและ morphism (มอร์ฟิส) แปลว่ารูปร่าง polymorphism จึงแปลว่า"หลายรูปทรง"หรือ"แปลงสภาพได้" เช่นเดียวกับ ดินน้ำมันที่เด็ก ๆ นำมาปั้นเป็นรูปทรงต่างๆได้หลายแบบ อาจสงสัยว่าอะไรที่เปลี่ยนรูปร่างได้และการเปลี่ยนรูปร่างได้มีประโยชน์อย่างไร คำตอบคือสิ่งที่เปลี่ยนรูปร่างได้คือmethod ที่สืบสันดานมาจาก base class ประโยชน์คือช่วยให้เรานำคลาสมาใช้ได้โดยสามารถเปลี่ยนแปลง method บาง method ของ base class ให้ตรงกับความต้องการ (เรียกว่า method override) หรือสร้าง method ใหม่ทับของเดิมไปเลย (เรียกว่า method hiding)

2.3 การสื่อสารผ่านทางพอร์ตอนุกรม (Serial Port Communication)

ในอดีตนั้นการเขียนโปรแกรมติดต่อ และควบคุมฮาร์ดแวร์มีความเฉพาะตัวเป็นอย่างยิ่ง การควบคุมในอดีตควบคุมผ่านภาษาแอสแซมบลีซึ่งเป็นภาษาที่ผูกติดกับฮาร์ดแวร์มาก แม้จะให้ผลการทำงานดี แต่การเขียนโปรแกรมและการแก้ไขโปรแกรมทำได้ยากมากเพราะภาษาแอสแซมบลีเป็นภาษาที่เข้าใจยากมากและเป็นภาษาที่ไม่เป็นโครงสร้าง

ต่อมาเราได้ใช้งานภาษาโปรแกรมยุคใหม่ๆ เพื่อใช้ในการเขียนโปรแกรมไม่ว่าจะเป็นภาษา C , Pascal ซึ่งภาษาCจะได้รับความนิยมมาก เพราะมีประสิทธิภาพใกล้เคียงกับภาษาแอสแซมบลีและเข้าใจได้ง่ายกว่า

แต่ในปัจจุบันการเขียนโปรแกรมส่วนใหญ่ จะเป็นการเขียนโปรแกรมแบบติดต่อผู้ใช้ซึ่งจะเขียนโปรแกรมได้ง่ายและรวดเร็ว ทำให้ภาษาโปรแกรมอย่าง Visual Basic เป็นที่นิยมอย่างมากทำให้มีความพยายามที่จะนำเอาภาษาโปรแกรมยุคใหม่ ๆ เหล่านี้ มาใช้ในการติดต่อควบคุมฮาร์ดแวร์ แม้ว่าในระยะแรกจะลำบาก แต่ปัจจุบันได้รับการพัฒนาให้ครอบคลุมเกือบทั้งหมดแล้ว

ในโครงงานนี้จะใช้การติดต่อสื่อสารระหว่างโปรแกรมกับฮาร์ดแวร์ ผ่านทางพอร์ตอนุกรม โดยอ้างมาตรฐาน RS 232 ใช้ในโปรแกรม วิชวล เบสิก

2.3.1 พื้นฐานการสื่อสารแบบอนุกรม

ถึงแม้ว่าการสื่อสารแบบอนุกรมในเครื่องคอมพิวเตอร์นั้น จะมีความเร็วในการสื่อสารช้ากว่าแบบขนาน ทั้งนี้ก็เพราะว่าการเคลื่อนย้ายข้อมูลแบบอนุกรมนั้นเป็นการส่งข้อมูลครั้งละ 1 บิต แต่พอร์ตขนานนั้นสามารถส่งข้อมูลได้ครั้งละหลายๆบิตพร้อมกัน ส่งผลให้การสื่อสารข้อมูลแบบอนุกรมมีความเร็วต่ำกว่าแบบขนาน

ตารางที่ 2.1 ตารางแสดงความแตกต่างของพอร์ตขนานและพอร์ตอนุกรม

พอร์ตขนาน (Parallel Port)	พอร์ตอนุกรม (Serial Port)
สร้างวงจรแล้วจะไม่สามารถใช้งานร่วมกันได้	สร้างวงจรแล้วบางวงจรจะไม่สามารถใช้งานร่วมกันได้
พอร์ตขนานมีการส่งข้อมูลแบบขนานคือได้ทีละหลาย ๆ บิต พร้อม ๆ กัน	พอร์ตอนุกรมมีการส่งข้อมูลแบบทีละบิต
การส่งข้อมูลของพอร์ตขนานมีความรวดเร็วในการส่งข้อมูลแต่ละชุด	การส่งข้อมูลแบบอนุกรมต้องใช้ระยะเวลานานในการส่งข้อมูลแต่ละชุด
การส่งข้อมูลของพอร์ตขนานเป็นการส่งข้อมูลในระยะที่ไม่ไกลมากนัก	การส่งข้อมูลได้ในระยะทางไกล ๆ

แต่การส่งข้อมูลแบบอนุกรมนั้น มีข้อที่เหนือกว่าการส่งข้อมูลแบบขนานคือ การสามารถส่งข้อมูลได้ระยะทางที่ไกลกว่าแบบขนาน อีกทั้งสายสัญญาณที่ใช้ยังมีน้อยกว่าการส่งข้อมูลแบบขนานอีกด้วย

การสื่อสารแบบอนุกรมสามารถแบ่งออกเป็น 3 รูปแบบ

1. Simplex สามารถส่งข้อมูลได้อย่างเดียว เป็นการสื่อสารแบบทางเดียว
2. Half – Duplex สามารถส่งข้อมูลไปยังปลายทาง และสามารถรับข้อมูลจากปลายทางได้ แต่ไม่สามารถส่งข้อมูลและรับข้อมูลได้ในเวลาเดียวกัน
3. Full – Duplex สามารถรับและส่งข้อมูลได้ในเวลาเดียวกัน

2.3.2 รูปแบบของการสื่อสารข้อมูลอนุกรม

สามารถแบ่งประเภทการสื่อสารตามลักษณะสัญญาณในการส่งได้ 2 แบบ

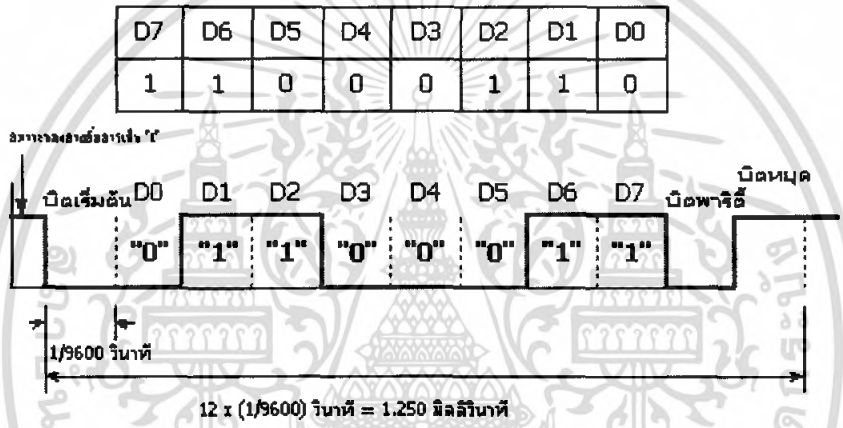
2.3.2.1 การสื่อสารข้อมูลอนุกรมแบบอะซิงโครนัส

เป็นวิธีการรับและส่งข้อมูลโดยไม่ต้องอาศัยสัญญาณนาฬิกาส่งร่วมไปด้วย แต่จะใช้อัตราความเร็วของจำนวนข้อมูลต่อวินาที และจะทำการเพิ่มบิตข้อมูลบางอย่างร่วมไปกับการส่งข้อมูลจริง เพื่อจะได้ทำการตรวจสอบข้อมูลได้อย่างถูกต้องมากยิ่งขึ้นแสดงดังรูปที่ 2.6 ซึ่งประกอบด้วยกัน 4 ส่วนคือ

1. บิตเริ่มต้น (Start bit) จะมีขนาด 1 บิต จะเป็นระดับลอจิกตรงกันข้ามกับระดับลอจิกของสภาวะสายสื่อสารขณะที่ยังไม่มีการส่งข้อมูล
2. บิตข้อมูล (Data bit) จะเริ่มจากบิตที่มีนัยสำคัญต่ำสุดก่อน หรือบิต LSB ก่อน โดยข้อมูลที่จะส่งอาจจะมีขนาด 5 , 6 , 7หรือ 8 บิตก็ได้
3. บิตแสดงสภาวะเลขคู่หรือเลขคี่ (Parity bit) มีขนาด 1 บิต โดยบิตนี้จะนำไป

ต่อท้ายกับบิตข้อมูล ค่าของบิตนี้ขึ้นอยู่กับจำนวนค่าของข้อมูลที่เป็น “1” โดยเลือกการส่งข้อมูลเป็นแบบพาริตี้คู่หรือพาริตี้คี่ ตัวอย่าง ถ้ากำหนดให้มีการส่งข้อมูลแบบพาริตี้คู่ แต่ข้อมูลมีเลข 1 เป็นจำนวนคี่ก็จะให้บิตพาริตี้นี้เป็น “1” เพื่อจะได้จำนวนเลข “1” เป็นคู่นั้นเอง ทำนองเดียวกันทางด้านรับเองก็ต้องมีการตรวจสอบจำนวนข้อมูลที่รับเข้ามาเป็น “1”รวมทั้งบิตพาริตี้ 1 บิต ถ้ามีค่า “1” เป็นจำนวนคู่แสดงว่าข้อมูลที่รับเข้ามาถูกต้อง สามารถกำหนดการรับและส่งข้อมูลเป็นแบบ NONE โดยไม่ต้องมีการตรวจสอบพาริตี้บิตก็ได้

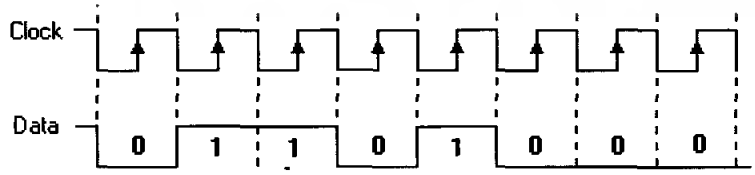
4. บิตสุดท้ายหรือบิตหยุด (Stop bit) เป็นการระบุถึงขอบเขตของการสิ้นสุดข้อมูล โดยจะทำให้ขาข้อมูลมีสถานะลอจิกเป็น“1” ซึ่งอาจมีจำนวนมากกว่า หนึ่งบิตก็ได้ เช่น 1 บิต 1.5บิต หรือ 2 บิต เป็นต้น



รูปที่ 2.6 แสดงการส่งข้อมูลขนาด 8 บิตแบบอนุกรมพร้อมด้วย บิตเริ่มต้น, บิตพาริตี้, บิตหยุด ด้วยความเร็ว 9600 บิตต่อวินาที

2.3.2.2 การสื่อสารแบบซิงโครนัส

การรับส่งข้อมูล จะมีสัญญาณนาฬิกา ซึ่งเป็นตัวกำหนด จังหวะเวลา การส่งข้อมูลร่วมอยู่ด้วยอีกเส้นหนึ่ง ใช้คู่กับสัญญาณข้อมูล ตัวอย่างเช่น การส่งสัญญาณจากคีย์บอร์ด



รูปที่ 2.7 สัญญาณการสื่อสารแบบซิงโครนัส

2.4 การใช้งานพอร์ตอนุกรม RS232

รูปที่ 2.8 ลักษณะของคอนเนคเตอร์แบบ D-Type

หัวต่อแบบD-Typeที่ใช้ในการสื่อสารแบบอนุกรมของเครื่องคอมพิวเตอร์นั้น จะมีอยู่2แบบ คือแบบ 9 ขา และ 25 ขา บางครั้งเราจะเรียกว่า DB9 และ DB25 ซึ่งหัวต่อทั้งสองชนิดจะมีลักษณะการทำงานของสัญญาณเหมือนกัน แต่การจัดเรียงไม่เหมือนกัน

รูปที่ 2.9 พอร์ตอนุกรมของ PC DB9 ตัวผู้ (Male)

รูปที่ 2.10 พอร์ตอนุกรมของอุปกรณ์ภายนอก DB9 ตัวเมีย (Female)

พอร์ตอนุกรมของPCจะเป็นคอนเน็คเตอร์แบบDB9 ตัวผู้ (Male) พอร์ตอนุกรมของอุปกรณ์ภายนอกจะเป็นคอนเน็คเตอร์แบบ DB9 ตัวเมีย (Female)

การจัดขา ของคอนเน็กเตอร์ อนุกรมแบบ DB9 และหน้าที่การใช้งานต่างๆ

รูปที่ 2.11 DB9 ตัวผู้ เมื่อมองจากด้านหลัง

ตารางที่ 2.2 ตารางแสดงรายละเอียดของสายสัญญาณต่างๆ ของพอร์ตอนุกรม

Pin	**Description**	**Type**
1	Data Carrier Detect (DCD)	Input
2	Received Data (RXD)	Input
3	Transmitted Data (TXD)	Output
4	Data Terminal Ready (DTR)	Output
5	Signal Ground (GND)	Input
6	Data Set Ready (DSR)	Input
7	Request To Send (RTS)	Output
8	Clear to Send (CTS)	Input
9	Ring Indicator (RI)	Input

สำหรับรายละเอียดของสายสัญญาณนั้นประกอบไปด้วย

Transmit Data : TD ใช้สำหรับส่งข้อมูลอนุกรมออกจากคอมพิวเตอร์

Receive Data : RD ใช้สำหรับรับข้อมูลอนุกรมเข้ามายังคอมพิวเตอร์

Request to Sent : RTS ใช้สำหรับส่งข้อมูลไปยังอุปกรณ์ปลายทาง เพื่อร้องขอให้อุปกรณ์ปลายทางส่งข้อมูลกลับมา

Clear to Sent : CTS ใช้สำหรับตรวจสอบว่าอุปกรณ์ที่เชื่อมต่อด้วยพร้อมที่จะรับข้อมูลหรือไม่ โดยจะคอยรับสัญญาณ RTS เมื่อทุกอย่างพร้อมก็จะทำการส่งข้อมูลออกทางขา TD

Data Set Ready : DSR ใช้สำหรับตรวจสอบการเชื่อมต่อกันระหว่างคอมพิวเตอร์กับอุปกรณ์ปลายทาง จะใช้คู่กับขา DTR

Signal Ground : SG เป็นกราวน์ระบบ

Carrier Detect : CD ขานี้จะ Active เมื่อมีการส่งสัญญาณ Carrier จากโมเด็ม

Data Terminal Ready : DTR ใช้สำหรับบอกให้อุปกรณ์ปลายทางรับรู้ว่าต้องติดต่อด้วยโดยขา DTR นี้ต้องเชื่อมต่อกับขา DTR ของอุปกรณ์ปลายทาง

Ring Indicator : RI ขานี้จะ Active เมื่อโมเด็มได้รับสัญญาณเรียกเข้าจากสายโทรศัพท์

2.4.1 องค์ประกอบของการรับส่งข้อมูลแบบอนุกรม

สื่อสารแบบอนุกรมที่นิยมใช้กับคอมพิวเตอร์นั้น เป็นการสื่อสารข้อมูลแบบอะซิงโครนัส นั่นคือต้องใช้สายสัญญาณเส้นเดียวทำหน้าที่ทั้งส่งส่วนที่เป็นข้อมูล และส่วนที่ใช้ควบคุมการส่งข้อมูล ดังนั้นข้อมูลที่อ่านได้แต่ละบิตจากการส่งข้อมูลแบบอนุกรมจึงต้องแยกถูกแยกแยะว่าใช้สำหรับวัตถุประสงค์ใด โดยเราสามารถแบ่งได้เป็น 4 ส่วน คือ

1. Start Bit	ขนาด 1 บิต
2. บิตข้อมูล (Data Character)	ขนาด 7 บิต หรือ 8 บิต
3. Parity Bit	ขนาด 1 บิต
4. Stop Bit	ขนาด 1 บิต หรือ 2 บิต

แต่ละตัวอักษรที่ถูกส่งออกไปเป็นกลุ่ม จะประกอบไปด้วยบิตเริ่มต้น บิตข้อมูล บิตพาริตี้ (จะมีหรือไม่มีก็ได้) และบิตจบ โดยเราพอจะสรุปหน้าที่ของแต่ละส่วนได้ดังนี้

Start Bit หรือบิตเริ่มต้นจะใส่ที่จุดเริ่มต้นเสมอ เพื่อเตือนอุปกรณ์ฝ่ายรับว่าข้อมูลกำลังจะมาถึง

Data Character หรือ บิตข้อมูล การส่งบิตข้อมูลจะส่งเป็นกลุ่ม ๆโดยทั่วไปจะส่ง 7 หรือ 8 บิต ซึ่งเพียงพอสำหรับการส่ง Ascii Word

Parity Bit หรือ บิตพาริตี้ใช้ในการตรวจสอบความถูกต้องของข้อมูลที่ส่ง เราจะใส่บิตพาริตี้เข้าไป แต่ทั้งตัวรับและตัวส่งจะต้องรู้กันว่าใช้พาริตี้แบบไหนในการส่งข้อมูล ซึ่งหลักการในการกำหนดบิตพาริตี้มีหลายแบบดังนี้

พาริตี้คู่ (Even Parity) ค่าของบิตพาริตี้นี้ เมื่อรวมกับทุก ๆ บิตของข้อมูลแล้วจะต้องมีบิตที่เป็นเลข 1 เป็นเลขคู่

พาริตี้คี่ (Odd Parity) ค่าของบิตพาริตี้นี้ เมื่อรวมกับทุกๆบิตของข้อมูลแล้ว จะต้องมีจำนวนบิตที่เป็นเลข 1 เป็นเลขคี่

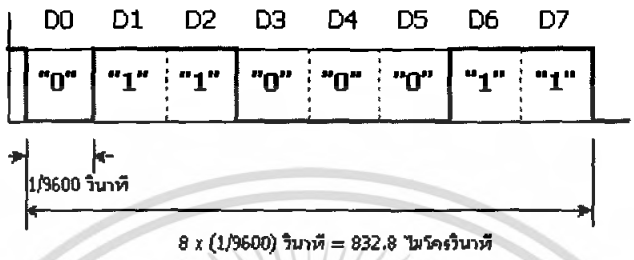
ไม่มีพารีตี้ (None) ถ้าตั้งบิตพาริตี้เป็น None ทั้งภาครับและส่งจะไม่มีการตรวจสอบพาริตี้ Stop Bit หรือบิตจบเป็นบิตที่ส่งมาปิดท้ายข้อมูล

2.4.2 อัตราเร็วในการสื่อสารข้อมูลอนุกรม

ในการสื่อสารข้อมูลแบบอนุกรมเพื่อรับหรือส่งข้อมูลจะเป็นลักษณะของกลุ่มข้อมูล ดังนั้นอัตราความเร็วจะต้องมีค่าเท่ากันระหว่างการรับและการส่งโดยทั่วไปเราจะระบุความเร็วของจำนวนบิตในการรับและส่งข้อมูลเป็นจำนวนของบิตที่จะส่งใน 1 วินาที โดยเรียกความเร็วในการส่งข้อมูลว่าอัตราบอด (Baud Rate) ซึ่งมีหน่วยเป็นบิตต่อวินาที เช่น 300, 1,200, 2,400, 4,800 และ 9,600 บิตต่อวินาที ในรูป2.12 ถ้าหากมีการส่งข้อมูลด้วยความเร็ว 9600 บิตต่อวินาที จะใช้เวลาในการรับส่ง

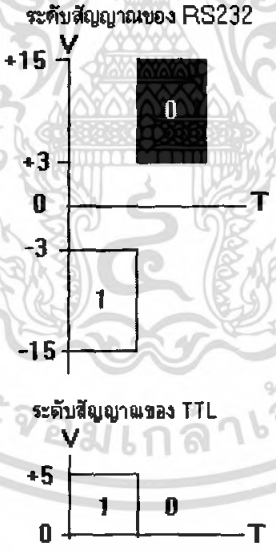
ข้อมูลหนึ่งบิตมีค่าเท่ากับ 1/9600 หรือ 104.1 ไมโครวินาที และเวลาในการรับส่งข้อมูลทั้ง 8 บิตจะมีค่าเท่ากับ 8 x 104.1 หรือ 832.8 ไมโครวินาที

D7	D6	D5	D4	D3	D2	D1	D0
1	1	0	0	0	1	1	0



รูปที่ 2.12 แสดงการส่งข้อมูลแบบอนุกรมด้วยความเร็ว 9600 บิตต่อวินาที

2.4.3 ระดับสัญญาณของ RS232



รูปที่ 2.13 ระดับสัญญาณของ RS232C และระดับ สัญญาณของ TTL

สัญญาณรบกวนที่เกิดขึ้นในสายนำสัญญาณมักจะมีแรงดันเป็นบวก เมื่อเทียบกับกราวน์ เพื่อป้องกันสัญญาณรบกวนนี้ จึงออกแบบแรงดันของโลจิก "1" เป็นลบคืออยู่ในช่วง -3V ถึง -15V ส่วนแรงดันของโลจิก "0" อยู่ในช่วง+3V ถึง +15V และเหตุที่ระดับสัญญาณของ RS232

อยู่ในช่วง+15Vถึง-15V ก็เพื่อให้ต่อสายสัญญาณไปได้ไกลขึ้น ดังนั้นจึงจำเป็นจะต้องมีวงจรเปลี่ยนระดับแรงดันของ RS232 มาเป็นระดับแรงดันของ TTL

2.5 ข้อมูลเบื้องต้นของ dsPIC

dsPIC คือชื่อของไมโครคอนโทรลเลอร์ 16 บิต จาก Microchip Technology Inc. ผู้ผลิตไมโครคอนโทรลเลอร์ PIC ซึ่งรู้จักกันเป็นอย่างดีในแวดวงพัฒนาระบบไมโครคอนโทรลเลอร์โดย Microchip Technology ได้กำหนดชื่ออย่างเป็นทางการสำหรับไมโครคอนโทรลเลอร์อนุกรมใหม่นี้ว่า Digital Signal Controller หรือ DSC นั่นหมายความว่า dsPIC เป็นไมโครคอนโทรลเลอร์ที่ได้รับการออกแบบมาเป็นพิเศษ เพื่องานประมวลผลสัญญาณดิจิตอลสำหรับสร้างระบบควบคุมอัตโนมัติที่มีความสามารถสูง

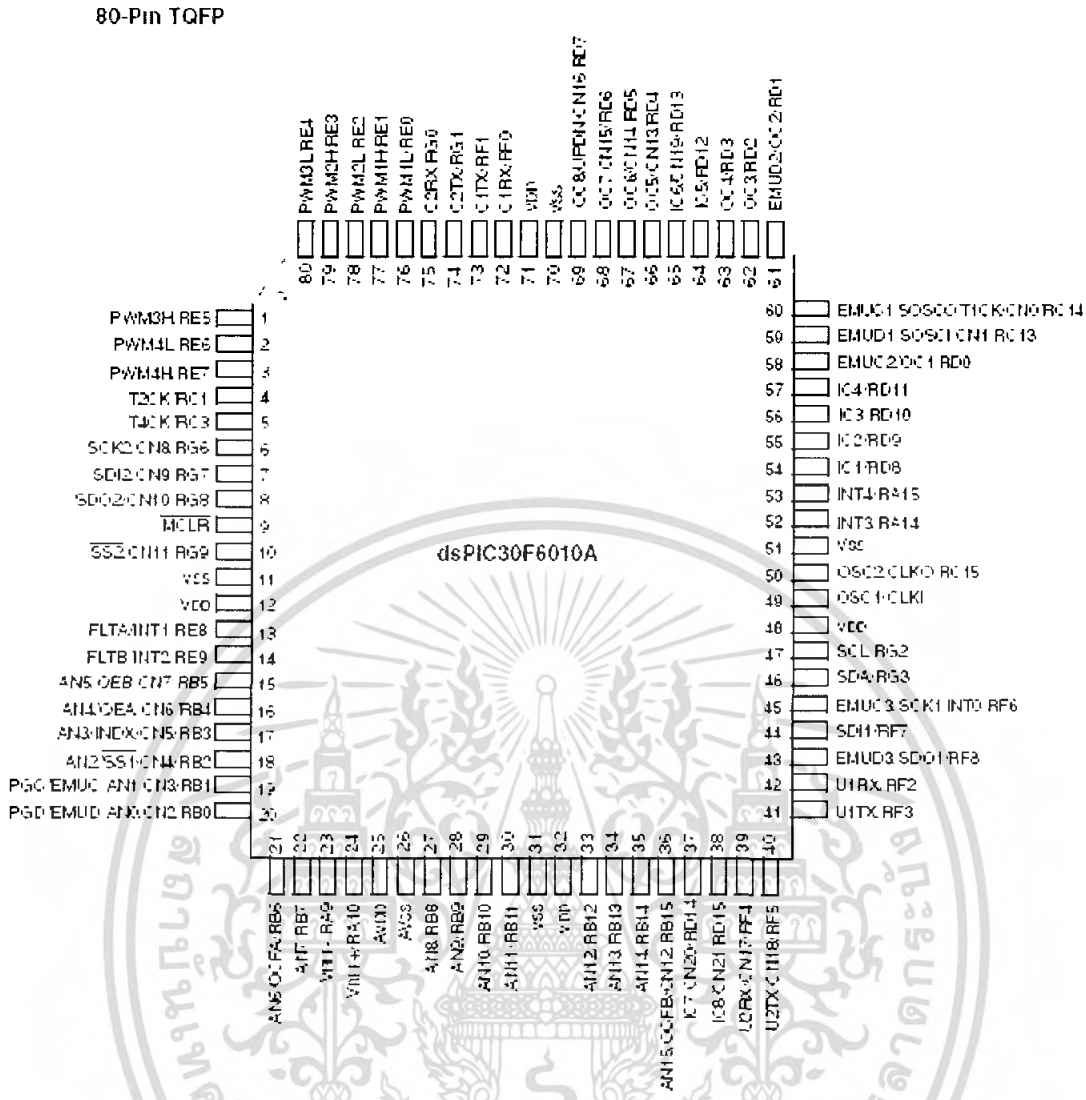
คุณสมบัติโดยรวมของ dsPIC

1. คุณสมบัติของซีพียู
2. คุณสมบัติด้านการประมวลสัญญาณดิจิตอล
3. คุณสมบัติของโมดูลฟังก์ชันพิเศษ

2.5.1 dsPIC 30F6010A

dsPIC 30F6010A คือไมโครคอนโทรลเลอร์ 16 บิต ซึ่งมีขาต่อใช้งาน 80 ขา สำหรับหน้าที่ของแต่ละขาได้อธิบายในตารางที่ภาคผนวก

รูปที่ 2.14 ลักษณะภายนอกของ dsPIC 30F6010A



รูปที่ 2.15 ลักษณะการทำงานของขาต่างๆ ของ dsPIC 30F6010A

ในโครงงานนี้ได้มีการนำฟังก์ชั่นการทำงานของไมโครคอนโทรลเลอร์ตระกูล dsPIC30F6010A มาใช้งานได้แก่

2.5.1.1 UART (Universal Asynchronous Receiver Transmitter)

1. คุณลักษณะพิเศษของ UART

- มีการส่งข้อมูลแบบ Full-Duplex
- มีฟังก์ชันพาริตี้คู่, พาริตี้คี่ หรือไม่มีพาริตี้ (สำหรับข้อมูล 8 บิต)
- มี Stop bit 1 และ 2 บิต
- มีอัตราเร็วในการส่งข้อมูลมากที่สุด 16 บิต

- มีช่วงอัตราเร็วในการส่งข้อมูลระหว่าง 38 bps ถึง 1.875 Mbps ที่ความถี่30MHz
- มีบัฟเฟอร์ของการส่งข้อมูล 4 word
- มีบัฟเฟอร์ของการรับข้อมูล 4 word
- มีการสนับสนุนการขัดจังหวะในการหาที่อยู่ (บิตที่ 9 มีค่าเท่ากับ 1)
- แยกการอินเตอร์รัปการถ่ายทอดข้อมูลและการรับข้อมูล
- มีโหมดการวนกลับเพื่อหาข้อผิดพลาดของการทำงาน

2. Enabling and Setting Up UART

Enabling the UART ในโหมดUARTจะเริ่มการทำงาน โดยการเซตค่าที่บิต UARTENในรีจิสเตอร์ UxMODE (เมื่อ x = 1หรือ 2) ในการทำงานครั้งแรก ขา UxTX และขา UxRX จะ เหมือนเป็น output และ input ตามลำดับ สิ่งที่สำคัญอีกสิ่งหนึ่งก็คือ รีจิสเตอร์ TRIS และรีจิสเตอร์ LATCH จะถูกเซตค่าให้เปรียบเสมือนเป็นขาพอร์ตอินพุท/เอาท์พุท ขา UxTX จะมีค่าเท่ากับ 1 เมื่อไม่มีการส่งข้อมูล

Disabling the UART UART จะหยุดการทำงาน โดยเคลียร์ค่าที่บิต UARTEN ในรีจิสเตอร์UxMODE ซึ่งหลังจากการรีเซตค่าจะกลายเป็นสภาวะdefault ถ้าUARTหยุดการทำงาน ขาอินพุท เอาท์พุท ที่ใช้ทั้งหมดจะทำงานเป็น port pin ภายใต้การควบคุมของบิต Latch และบิต TRIS เมื่อUARTหยุดทำงานจะเหมือนการตั้งค่าสภาวะว่างเปล่าให้กับบัฟเฟอร์นั้นข้อมูลที่อยู่ภายในบัฟเฟอร์จะหายไปและอัตราเร็วในการส่งข้อมูลจะรีเซตค่าใหม่

ค่าผิดพลาดและสถานะ Flagที่เกี่ยวข้องกับโมดูล UART จะเซตค่าใหม่ทั้งหมดเช่น บิต URXDA, OERR, FERR, PERR, UTXEN, UTXBRK และ UTXBF เนื่องจากว่าบิต RIDLE และ บิต TRMT เซตค่าใหม่ บิตควบคุมอื่น ๆ เช่น ADDEN, URXISEL<1:0>, UTXISEL และรีจิสเตอร์UxMODE และ UxBRG จะไม่มีผล

ในการเคลียร์ค่าบิต UARTEN ในขณะที่ UART กำลังทำงานอาจทำได้ไม่สำเร็จ ซึ่งจะต้องรอจนกว่าการส่งข้อมูล การรับข้อมูลและการเซตค่าต่าง ๆ ที่ได้กล่าวไว้จะเสร็จสิ้น

Setting up data,Parity and Stop bit selection บิตควบคุม PDSEL<1:0> ที่อยู่ในรีจิสเตอร์ UxMODE ที่เคยถูกใช้เป็นตัวเลือกความยาวของข้อมูลและพาริตี้ที่ใช้ในการส่งข้อมูล โดยที่ความยาวของข้อมูลอาจจะเป็น 8 บิตที่ใช้พาริตี้คู่ พาริตี้คี่หรือไม่พาริตี้หรือ จะเป็น 9 บิตที่ไม่ใช้พาริตี้

บิตSTSELจะหมายถึงบิตหยุด 1 หรือ 2 บิตที่จะถูกเรียกใช้ในระหว่างการส่งข้อมูล ในการตั้งค่าให้กับ UART ที่มีความยาวของข้อมูลการส่ง 8 บิต ไม่ใช้พาริตี้และมีบิตหยุด 1 บิตจะถูกกำหนดไว้ว่า “ชนิดของตัวแปร, 8, N, 1”

3. Transmitting Data

Transmitting in 8-bit data mode

ขั้นตอนการกำหนดค่าในการส่งข้อมูล 8 บิต

1. ตั้งค่าให้กับ UART ขั้นแรกต้องกำหนดความยาวให้กับข้อมูล พาริตี้และจำนวนบิตหยุดที่ต้องการ จากนั้นให้อินเตอร์รัปต์ของการส่ง-รับข้อมูลเริ่มทำงานและตั้งค่าบิตเริ่มต้นในรีจิสเตอร์ UxMODE และ UxSTA รวมทั้งกำหนดค่าอัตราเร็วในการส่งข้อมูลในรีจิสเตอร์ UxBRG ด้วย

2. ให้ UART เริ่มทำงานโดยการตั้งค่าที่บิต UARTEN (UxMODE<15>)

3. ตั้งค่าบิต UTXEN (UxSTA<10>) เป็นการเริ่มส่งข้อมูลได้

หมายเหตุ บิต UTXEN ต้องเซตค่าหลังจากเซตค่าให้กับบิต UARTEN แล้ว

4. กำหนดไบต์ที่ต้องการให้เป็นไบต์ต่ำสุดที่ต้องการส่งให้กับUxTXREGค่านั้นจะถูกส่งไปยังTransmit Shift Register (UxTSR)ทันทีและบิตถัดมาจะเริ่ม shift ตามจังหวะขอบขาขึ้นของสัญญาณอัตราเร็วของข้อมูล

5. การอินเตอร์รัปต์การส่งข้อมูล จะถูกสร้างขึ้นอยู่กับค่าบิตควบคุม UTXISEL (UxSTA<15>)

Transmitting in 9-bit data mode

ลำดับขั้นตอนในการส่งข้อมูล 9 บิตจะคล้ายกับการส่งข้อมูล 8 บิต เว้นแต่ว่าใน 16 บิตจะมี 7 บิตที่ถูกเคลียร์ค่าเสมอและต้องเขียนในรีจิสเตอร์ UxTXREG

Transmit buffer (UxTXB)บัฟเฟอร์ของการส่งข้อมูลจะมีค่ากว้าง 9 บิตและมี 4 ตัวอักษร รวมทั้งTransmit Shift Register(UxTSR)ผู้ใช้งานจะสามารถใช้งานได้อย่างมีประสิทธิภาพถ้าใช้บัฟเฟอร์ 5-FIFO (First-In, First-Out) และรีจิสเตอร์ UTXBF จะแสดงสถานะของบิตไม่ว่าบัฟเฟอร์การส่งข้อมูลนั้นจะเต็มหรือไม่ก็ตาม ถ้าผู้ใช้พยายามที่จะเขียนข้อมูลลงไปในบัฟเฟอร์ที่เต็มแล้ว ข้อมูลใหม่ที่ผู้ใช้ใส่เข้าไปจะไม่ปรากฏอยู่ใน FIFO และไม่มีข้อมูลใดเลื่อนหรือหายไปในบัฟเฟอร์นั้น แต่จะเห็นต่อเมื่อบัฟเฟอร์นั้นเกิดกรณี overrun FIFO จะรีเซตระหว่างการทำงานต่อเมื่ออุปกรณ์ใดก็ตามถูกรีเซต แต่จะไม่ส่งผลกระทบเมื่ออุปกรณ์นั้นกำลังทำงานหรือ wake-up จากโหมด power-saving

Transmit Interrupt

Transmit Interrupt Flag (U1TXIF or U2TXIF) ถูกตั้งอยู่ที่เดียวกันกับรีจิสเตอร์ Interrupt Flag ตัวส่งสัญญาณ จะถูกเซตขอบขาการทำงานที่บิต UxTXIF ซึ่งเงื่อนไขในการกำหนด Interrupt จะขึ้นอยู่กับบิตควบคุม UTXISEL ดังนี้

1. ถ้า UTXISEL = 0 interrupt จะเริ่มทำงาน เมื่อข้อมูลถูกส่งจากบัฟเฟอร์ไปยัง UxTSR นั่นหมายความว่าบัฟเฟอร์นั้นมีพื้นที่ว่างอย่างน้อยที่สุด 1 word

2. ถ้า UTXISEL = 1 interrupt จะเริ่มทำงาน เมื่อข้อมูลถูกส่งจากบัฟเฟอร์ไปยัง UxTSR และบัฟเฟอร์ของการส่งนั้นจะว่าง ในการสลับการทำงานของ interrupt ทั้ง 2 โหมดในระหว่างการทำงานอาจเป็นไปได้หรือบางครั้งอาจยืดหยุ่นได้มากกว่า

TRANSMIT BREAK ในการเซตบิต UTXBRK (UxSTA<11>) จะทำให้สายส่ง UxTX กลายเป็นลอจิก 0 ซึ่งบิต UTXBRK จะถูกยกเลิกการส่งข้อมูลทั้งหมด ดังนั้นปกติแล้วผู้ใช้ควรจะรอตัวส่งข้อมูลจนกระทั่งเข้าสู่โหมด Idle ก่อนที่จะทำการตั้งค่าบิต UTXBRK

ในการส่งตัวอักษรเพื่อทำการหยุดข้อมูล บิต UTXBRK ต้องตั้งค่ารอบเวลาความเร็วในการส่งข้อมูลอย่างน้อยมีค่าเท่ากับ 13 ต่อจากนั้นบิต UTXBRK จะต้องถูกเคลียร์ค่าเพื่อสร้างบิตหยุด ผู้ใช้งานต้องรอรอบความเร็วในการส่งข้อมูลอย่างน้อย 1 หรือ 2 รอบเพื่อให้เป็นไปตามกฎของบิตหยุดก่อนที่จะเพิ่ม UxTXB หรือเริ่มทำการส่ง

4. Receiving Data

โหมดการรับข้อมูล 8 บิตหรือ 9 บิต

ขั้นตอนการปฏิบัติในการรับข้อมูล 8 บิตหรือ 9 บิต

1. ตั้งค่าและ Enable UART (ดูหัวข้อ 18.3 "Transmitting Data")

2. อินเตอร์รัปต์ของการรับข้อมูลจะถูกสร้างขึ้น เมื่อข้อมูลหนึ่งตัวหรือมากกว่าหนึ่งตัวเคยได้รับมาก่อนแล้ว ขึ้นอยู่กับการตั้งค่าอินเตอร์รัปต์การรับข้อมูลที่ระบุในบิต URXISEL

3. ถ้าการยกเลิกความผิดพลาดเกิดขึ้นต้องอ่านค่าบิต OERR เพื่อดูความหมายและสามารถตั้งค่าบิต OERR ได้โดยซอฟท์แวร์

4. อ่านข้อมูลที่ได้รับจาก UxRXREG จากนั้นข้อมูลใน UxRXREG จะถูกเคลื่อนไปยัง word ต่อไปของ FIFO และค่าของ PERR และ FERR จะถูกอัพเดตขึ้นมา

Receive Buffer

บัฟเฟอร์ที่ใช้รับข้อมูลมี 4 word รวมไปถึง Receive Shift Register (UxRSR) ถ้าจะใช้งานให้ได้มีประสิทธิภาพผู้ใช้ควรจะต้องใช้บัฟเฟอร์ FIFO ที่มีขนาด 5 words

URXDA (UxSTA<0>) = 1 จะบ่งบอกว่าบัฟเฟอร์นั้นมีข้อมูลอยู่แล้ว แต่ถ้า URXDA = 0 จะหมายถึงว่าบัฟเฟอร์นั้นว่างเปล่า ถ้าผู้ใช้พยายามที่จะอ่านค่าจากบัฟเฟอร์นี้ ค่าเก่าที่อยู่ในบัฟเฟอร์นี้จะถูกอ่านและจะไม่มีข้อมูลใดเลื่อนออกไปภายใน FIFO นี้

FIFO จะรีเซตค่าตามที่เซตให้กับอุปกรณ์มันจะไม่ส่งผลกระทบเมื่ออุปกรณ์กำลังทำงานหรือเข้าสู่การทำงานปกติจากโหมด power-saving

Receive Interrupt

แฟลก Receive Interrupt (U1RXIF or U2RXIF) สามารถอ่านค่าได้จากแฟลก interrupt เหมือนกัน แฟลก interrupt จะเซตค่าโดยตัวรับเงื่อนไขของการเซตค่าตัวรับ interrupt ขึ้นอยู่กับการเซตค่าบิตควบคุมใน URXISEL<1:0> (UxSTA<7:6>)

1. ถ้า URXISEL<1:0> = 00 หรือ 01 interrupt จะถูกสร้างขึ้นตลอดเวลา ข้อมูลจะถูกส่งจาก Receive Shift Register (UxRSR) ไปยังบัฟเฟอร์ของการรับข้อมูลซึ่งอาจมีข้อมูลมากกว่าหรือเท่ากับ 1 ตัวอักษรก็เป็นได้

2. ถ้า URXISEL<1:0> = 10 interrupt จะถูกสร้างขึ้น เมื่อข้อมูลส่งจากReceive Shift Register (UxRSR) ไปยังบัฟเฟอร์ของการรับข้อมูลด้วยเหตุที่ว่าการส่งข้อมูลจะต้องประกอบไปด้วย 3 ตัวอักษร

3. ถ้า URXISEL<1:0> = 11 interrupt จะเซตค่า เมื่อข้อมูลถูกส่งจากReceive Shift Register (UxRSR) ไปยังบัฟเฟอร์ของการรับข้อมูล ด้วยเหตุที่ว่าการส่งข้อมูลจะต้องประกอบไปด้วย 4 ตัวอักษร ในการเปลี่ยนการทำงานระหว่างโหมด interrupt ระหว่างการทำงานสามารถทำได้โดยทั่วไปแล้วจะไม่ค่อยเหมาะสมเท่าที่ควร

5. Reception Error Handling

Receive Buffer Over Run Error (OERR BIT)

การเซตบิต OERR (UxSTA<1>)

a) บัฟเฟอร์ของการรับข้อมูลจะต้องเต็ม

b) Receive Shift Register จะต้องเต็ม แต่ไม่สามารถส่งข้อมูลไปยังบัฟเฟอร์ของการรับข้อมูลได้

ในการเซตบิตOERRครั้งแรกไม่มีข้อมูลใดในUxRSRเลื่อน (จนกระทั่งบิต OERR จะถูกเคลียร์หรือรีเซตค่า)

Framing Error (FERR)

บิต FERR (UxSTA<2>) จะถูกเซตค่าถ้า เป็น 0 หมายถึงว่าจะถูกค้นหาแทนที่บิตหยุด ถ้าเลือกบิตหยุด2บิต ทั้ง2บิตจะต้องมีค่าเป็น1ไม่อย่างนั้นแล้ว FERR จะถูกตั้งค่าบิต FERR สามารถอ่านได้เท่านั้น ไม่สามารถเปลี่ยนแปลงได้และจะถูกเคลียร์จากการรีเซต

Parity Error (PERR)

บิต PERR (UxSTA<3>) ถูกเซตค่า ถ้าพาริตี้ของข้อมูลการรับนั้นผิดบิตที่ผิดพลาด สามารถนำไปปรับใช้ได้ถ้าเลือกพาริตี้ (คู่หรือคี่) บิต FERR สามารถอ่านได้เท่านั้นไม่สามารถเปลี่ยนแปลงได้และจะถูกเคลียร์จากการรีเซต

IDLE Status

เมื่อให้ตัวรับข้อมูลทำงานบิตRIDLE(UxSTA<4>)จะมีค่าเป็น 0 ระหว่างค่าสุดท้ายของบิตหยุดและบิตถัดไปของบิตเริ่มต้น บิตRIDLEจะมีค่าเท่ากับ 1 UART จะเข้าสู่โหมด Idle

Receive Break

ตัวรับข้อมูลจะนับและตั้งค่าจำนวนบิตเวลาที่แน่นอนเอาไว้ ซึ่งจะขึ้นอยู่กับการตั้งค่าใน บิตPDSEL (UxMODE<2:1>) และบิต STSEL (UxMODE<0>)

ถ้าการหยุดใช้เวลานานกว่า 13 บิตเวลา การรับค่าจะพิจารณาหลังจากระบุบิตเวลาไว้ที่ PDSEL และ STSEL แล้ว บิต URXDA และ บิต FERR จะเซตค่าให้เท่ากับ 0 เพื่อให้ตัวรับค่า FIFO ตัว Interrupt จะเริ่มทำงานถ้ามีการเซตค่า RIDLE ที่ถูกต้องเหมาะสม

เมื่อโมดูลของการรับมีสัญญาณของการหยุดเป็นเวลานาน และผู้รับถูกค้นหาบิตเริ่มต้น บิตข้อมูลและบิตFERRผู้รับจะต้องรอบิตหยุดก่อนที่จะหาบิตเริ่มต้นตัวต่อไปไม่สามารถสมมติเงื่อนไขของการหยุดที่บิตเริ่มต้นตัวต่อไปได้

การหยุดถูกพิจารณาจากตัวอักษร 0 และการตั้งค่าFERRตัวอักษรที่เป็นความหมายของการหยุดจะถูกบรรจุลงในบัฟเฟอร์ จะไม่มีการรับข้อมูลใด ๆ ปรากฏจนกระทั่งได้รับบิตหยุดก่อน

**หมายเหตุ RIDLE จะเกิดค่าสูงสุดเมื่อไม่เคยได้รับบิตหยุดมาก่อน

6. Address Detect Mode

ตั้งค่าบิต ADDEN (UxSTA<5>) ให้ทำงาน ซึ่งจะทำให้บิตที่9 แสดงค่าเป็น1 เป็นการแสดงที่อยู่ของตัวรับมากกว่าเป็นข้อมูล ในโหมดนี้สามารถใช้ได้กับการติดต่อข้อมูล 9 บิตเท่านั้น การควบคุมบิต URXISEL จะไม่มีผลกระทบใด ๆ ต่อโหมดการทำงานนี้

7. Loopback Mode

การตั้งค่าบิต LPBACK ให้ทำงานซึ่งเป็นโหมดพิเศษที่ขา UxTX ต่อภายในกับขา UxRX เมื่อเรา configureให้กับโหมด loopback ขา UxRX จะยกเลิกการติดต่อภายในจาก UART อย่างไรก็ตามขา UxTX ยังคงทำหน้าที่ต่อเช่นเดิมในการเลือกโหมดการทำงาน

1. configure UART เพื่อเข้าสู่โหมดการทำงาน
2. กำหนดค่า LPBACK = 1 เพื่อให้โหมด Loopback ทำงาน
3. เริ่มการส่งข้อมูลด้วยการกำหนดในหัวข้อ "Transmitting Data"

8. Baud Rate Generator (BRG)

UART มีอัตราเร็วในการส่งได้สูงสุด 16 บิต รีจิสเตอร์ Baud Rate Generater (UxBRG) สามารถอ่านและเขียนได้ อัตราเร็วในการส่งสามารถคำนวณได้ดังนี้

BRG = ค่าของ 16 บิตในรีจิสเตอร์ UxBRG (0-65535)

F_{CY} = ค่าความถี่ ($1/T_{CY}$)

สมการอัตราเร็วในการส่งข้อมูล Baud Rate = F_{CY}/ (16*(BRG+1)) ดังนั้นอัตราเร็วในการส่งข้อมูลที่เป็นไปได้สูงสุดมีค่าเท่ากับ F_{CY}/16 (ถ้า BRG = 0) และค่าอัตราเร็วในการส่งข้อมูลที่เป็นไปได้ต่ำสุดมีค่าเท่ากับF_{CY}/ (16*65536) ถ้าใช้อัตราเร็วในการส่งข้อมูลสูงสุด16บิตที่30 MIPS จะมีอัตราเร็วการส่งข้อมูลต่ำสุดเท่ากับ 28.5 bps

9. UART Operation During CPU

Sleep and Idle Modes

UART OPERATION DURING CPU SLEEP MODE

เมื่ออุปกรณ์เข้าสู่โหมดSleepนาฬิกา ทั้งหมดจะหยุดการทำงานและมีลอจิกเป็น 0 ในขณะที่กำลังส่งข้อมูล ถ้าเข้าสู่โหมดSleep การส่งข้อมูลนั้นจะล้มเหลว ขาUxTXจะมีค่าเป็นลอจิก 1 เช่นเดียวกันถ้ากำลังรับข้อมูลแล้วเข้าสู่โหมดSleep จะทำให้การรับข้อมูลนั้นล้มเหลว UxSTA, UxMODE, รีจิสเตอร์การส่ง, รีจิสเตอร์การรับ, บัฟเฟอร์และรีจิสเตอร์UxBRG จะไม่ส่งผลกระทบต่อโหมด Sleep

ถ้าบิตwake(UxMODE<7>)ถูกตั้งค่าก่อนที่อุปกรณ์จะเข้าสู่โหมดsleepจากนั้นขอบขาลงของUxRXจะเริ่มรับข้อมูล ในโหมดReceive Interrupt Select (URXISEL)จะไม่มีผลกระทบต่อฟังก์ชันนี้ ถ้าการขัดจังหวะการรับข้อมูลเริ่มทำงานอาจจะทำให้อุปกรณ์กลับมาทำงานหลังจากอยู่ในโหมดsleepได้ บิตUARTENจะต้องตั้งค่าตามการขัดจังหวะของการทำงาน

UART Operation during CPU Idle mode

ใน UART บิต USIDL จะถูกเลือก โมดูลจะหยุดการทำงานเมื่ออุปกรณ์นั้นเข้าสู่โหมด Idle หรือไม่เช่นนั้น โมดูลนั้นจะดำเนินการต่อเช่น ถ้า USIDL = 0 โมดูลนี้จะดำเนินการต่อในโหมด Idle แต่ถ้า USIDL = 1 โมดูลจะหยุดการทำงานบนโหมด Idle

2.5.1.2 ADC

1. คุณสมบัติโดยสรุปของโมดูลแปลงสัญญาณอนาลอกเป็นดิจิตอล

ตัวแปลงจากอนาล็อกเป็นดิจิตอลความเร็วสูง สามารถแปลงสัญญาณอินพุทจากอนาล็อกไปเป็นเลขดิจิตอลแบบ 10 บิต โมดูลนี้ถูกทำให้เป็นฐานบนสถาปัตยกรรม Successive Approximation Register (SAR) และจัดเตรียมค่าสูงสุดของอัตราสุ่ม ตัวอย่างของ1Msps โมดูล A/D มีอินพุทอนาล็อก 16 อินพุท ซึ่งถูกทำให้มีหลากหลายค่าภายใน 4 ตัวอย่างและโฮลด์แอมพลิไฟล์ค่าเอาท์พุทของตัวอย่างและโฮลด์อินพุทภายในตัวแปลงซึ่งทำให้เกิดผลลัพธ์ สัญญาณอะนาล็อกนั้นอ้างอิงแรงดันไฟฟ้าเป็นสิ่งที่ซอฟแวร์เลือก เช่นเดียวกันกับแหล่งจ่ายแรงดันไฟฟ้า (AV_{DD} / AV_{SS}) หรือระดับแรงดันไฟฟ้าบนขา (V_{REF+}/ V_{REF-} และA/D คอนเวอร์เตอร์ มีลักษณะอย่างหนึ่งคือสามารถทำงานได้ในขณะที่อยู่ในสลีฟโหมด)

2. รีจิสเตอร์ที่ใช้งานในโมดูล ADC

ในโมดูลADCของไมโครคอนโทรลเลอร์dsPICมีรีจิสเตอร์ที่ใช้ควบคุมและกำหนดค่าอันประกอบด้วย

- ADCON1 (A/D Control Register1): รีจิสเตอร์ควบคุมโมดูล ADC ตัวที่ 1
- ADCON2 (A/D Control Register2): รีจิสเตอร์ควบคุมโมดูล ADC ตัวที่ 2
- ADCON3 (A/D Control Register3): รีจิสเตอร์ควบคุมโมดูล ADC ตัวที่ 3
- ADCHS (A/D Input Select Register): รีจิสเตอร์เลือกช่องของวงจร S/H ที่ต่อกับขาพอร์ตอินพุตอนาล็อกที่ต้องการแปลงสัญญาณ
- ADPCFG (A/D Port Configuration Register): รีจิสเตอร์กำหนดค่าทางฮาร์ดแวร์ของอินพุตอะนาล็อก
- ADCSSL(A/D Input Scan Select Register): รีจิสเตอร์เลือกช่องอินพุตที่ต้องการแปลงสัญญาณ

A/D Result Buffer

โมดูลจำกัดด้วยพอร์ทคู่ 16 words บัฟเฟอร์อ่านอย่างเดียว เรียกใช้ ADCBUF0 ถึง ADCBUFF ผลลัพธ์ไปที่บัฟเฟอร์ A/D แรมกว้าง10บิท แต่อ่านภายในรูปแบบแตกต่าง16-bit words ปริมาณความจุของการแปลง 16 A/D บัฟเฟอร์ผลลัพธ์อยู่ที่รีจิสเตอร์ ADCBUF0 ถึง ADCBUFF ไม่สามารถเขียนโดยผู้ใช้ซอฟแวร์

Conversion Operation

หลังจากโมดูลA/D ถูกกำหนดจะได้ตัวอย่างเป็นการเริ่มต้นโดยเซ็ทค่าบิท SAMP ซึ่งแตกต่างกับแหล่งจ่ายดังนั้นใน programmablc bit, timer time-outs และเหตุการณ์ต่างๆ ภายนอกได้มาทำให้สิ้นสุดและเริ่มการแปลง เมื่อตัวแปลงA/Dเสร็จสมบูรณ์ ผลลัพธ์ถูกโหลดเข้าภายใน ADCBUF0 ถึง ADCBUFFและ A/D Interrupt Flag ADIF ถึงบิทสิ้นสุดเป็นการเซ็ทหลังจากเลขของตัวอย่างที่เฉพาะเจาะจงโดยบิท SMPI

ขั้นตอนการใช้ A/D มีดังนี้

1. กำหนดโมดูล A/D
   - กำหนดขาอนาล็อก แรงดันอ้างอิงและดิจิตอล I/O
   - เลือก A/D input channels
   - เลือก A/D conversion clock
   - เลือก A/D conversion trigger
   - เปิดโมดูล A/D
2. กำหนด A/D Interrupt (ถ้าต้องการ)
   - เคลียร์บิท ADIF
   - เลือก A/D Interrupt priority
3. เริ่มการสุ่มตัวอย่าง
4. รอการร้องขอเวลาเพิ่มเติม
5. Trigger เพิ่มเติมจบ เริ่มทำการแปลง
6. รอการแปลง A/D สำเร็จหรือว่ารอ A/D Interrupt อย่างใดอย่างหนึ่ง
7. อ่านผลลัพธ์บัฟเฟอร์ A/D แล้วเคลียร์ค่า ADIF ถ้าต้องการ

2.5.1.3 PWM

1. คุณสมบัติโดยสรุปของโมดูล MCPWM

- ความละเอียดของสัญญาณ PWM ที่สร้างขึ้นเท่ากับ $T_{cy}/2$
- ในโมดูล MCPWM 1 ชุด มี2 เอาต์พุต
- สามารถใช้งานเอาต์พุตของโมดูล MCPWM แยกกันอย่างอิสระและร่วมกัน
- เมื่อทำงานในแบบร่วมกันหรือคอมพลีเมนทารี สามารถกำหมดค่าวิกฤต(dead time) เพื่อช่วยการขับมอเตอร์ 3 เฟสให้เป็นไปอย่างมีประสิทธิภาพ
- สามารถเลือกโหมดเอาต์พุตได้ 4 โหมด
- โหมดปรับขอบสัญญาณ (Edge aligned mode)
- โหมดสัญญาณเดี่ยว (Single event mode)
- โหมดปรับสัญญาณกึ่งกลาง (Center aligned mode)
- โหมดปรับสัญญาณกึ่งกลางและปรับปรุงค่า (Center aligned mode with double updates)
- มีอินพุตสำหรับจับความผิดพลาดในการทำงาน (FAULT) แบบโปรแกรมได้
- สามารถสร้างสัญญาณกระตุ้นส่งไปยังโมดูลแปลงสัญญาณอนาลอกเพื่อกำหนดจังหวะการทำงานให้สัมพันธ์กัน

2. รีจิสเตอร์ที่ใช้งานในโมดูล MCPWM (Motor Control PWM)

ในโมดูล MCPWM ของไมโครคอนโทรลเลอร์ dsPIC มีรีจิสเตอร์ที่ใช้ควบคุมและกำหนดค่า อันประกอบด้วย

- PTCON รีจิสเตอร์ควบคุมฐานเวลาในการกำเนิดสัญญาณ PWM
- PTMR รีจิสเตอร์กำหนดค่าฐานเวลาของการกำเนิดสัญญาณ PWM
- PTPERรีจิสเตอร์กำหนดคาบเวลาของฐานเวลาสำหรับการกำเนิดสัญญาณPWM
- SEVTCMP รีจิสเตอร์เปรียบเทียบค่า
- PWMCON1 รีจิสเตอร์ควบคุม PWM #1
- PWMCON2 รีจิสเตอร์ควบคุม PWM #2
- DTCON1 รีจิสเตอร์ควบคุมค่าเวลาวิกฤตหรือ Dead Time #1
- DTCON2 รีจิสเตอร์ควบคุมค่าเวลาวิกฤตหรือ Dead Time #2
- FLTACONรีจิสเตอร์ควบคุมการตรวจจับความผิดปกติของการขับมอเตอร์ชุด A
- FLTBCONรีจิสเตอร์ควบคุมการตรวจจับความผิดปกติของการขับมอเตอร์ชุด B
- PDC1 รีจิสเตอร์กำหนดค่าดิวตี้ไซเกิลของโมดูลกำหนดสัญญาณPWMชุดที่1

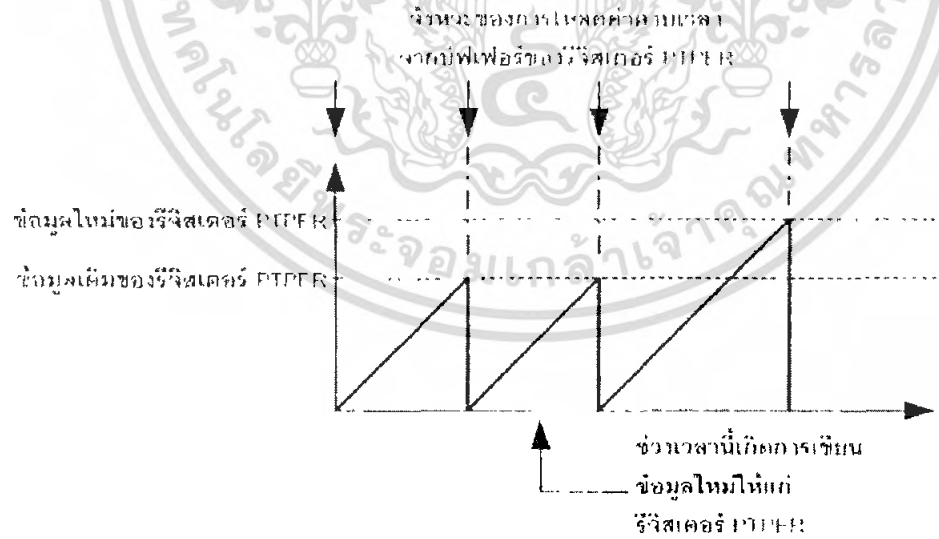
- PDC2 รีจิสเตอร์กำหนดค่าดิวตี้ไซเกิลของโมดูลกำหนดสัญญาณPWMชุดที่2
- PDC3 รีจิสเตอร์กำหนดค่าดิวตี้ไซเกิลของโมดูลกำหนดสัญญาณPWMชุดที่3
- PDC4 รีจิสเตอร์กำหนดค่าดิวตี้ไซเกิลของโมดูลกำหนดสัญญาณPWMชุดที่4

3. คาบเวลาของสัญญาณ PWM

รีจิสเตอร์ PTPER ถูกใช้สำหรับกำหนดค่าการนับคาบเวลาของรีจิสเตอร์ PTMR ผู้พัฒนาต้องระบุข้อมูลขนาด 15 บิตลงในบิต 0 ถึง 14 ของรีจิสเตอร์ PTPER เมื่อโมดูลนี้ทำงาน จนกระทั่งค่าของรีจิสเตอร์ PTMR เท่ากับ PTPER ค่าฐานเวลาจะรีเซตเป็น "0" หรือเปลี่ยนทิศทางการนับค่าในสัญญาณนาฬิกาลูกถัดไปขึ้นอยู่กับการกำหนดโหมดทำงาน

คาบเวลาของฐานเวลามีขนาดของบัฟเฟอร์เป็น 2 เท่า เพื่อรองรับการเปลี่ยนแปลงค่าในระหว่างการทำงานได้โดยปราศจากการรบกวน นั่นคือรีจิสเตอร์PTPERจะมีรีจิสเตอร์บัฟเฟอร์สำรองรับค้าที่ต้องการเปลี่ยนแปลงใหม่ในระหว่างที่กำลังทำงานกับค่าเดิม โดยรีจิสเตอร์บัฟเฟอร์นี้ผู้ใช้งานไม่สามารถเข้าถึงได้ ข้อมูลสำหรับกำหนดค่าคาบเวลาจะถูกเขียนลงในรีจิสเตอร์PTPER แตกต่างกันไปตามโหมดการทำงานของฐานเวลา PWM

ก) ในโหลดเปลี่ยนแปลงค่าอิสระและโหมดทำงานครั้งเดียว ข้อมูลจากรีจิสเตอร์ PTPER จะถูกโหลดลงในรีจิสเตอร์คาบเวลา เมื่อรีจิสเตอร์ PTMR ถูกรีเซตเป็น "0" หลังจากที่ค่าของรีจิสเตอร์ PTMR ตรงกับค่าของ PTPER ดังรูปที่2.16



รูปที่ 2.16 ไดอะแกรมเวลาแสดงการโหลดค่าของรีจีสเตอร์ PTPER เมื่อฐานเวลาของ PWMทำงานในโหมดเปลี่ยนแปลงค่าอิสระและโหมดทำงานครั้งเดียว

ข้อมูลสำหรับกำหนดคาบเวลาของสัญญาณ PWM ที่เขียนไปยังรีจิสเตอร์ PTPER สามารถคำนวณได้จากสมการต่อไปนี้

$$PTPER = \frac{F_{CY}}{F_{PWM} \times PTMR\_\mathrm{Pr}escaler} - 1$$

โดย

F_{CY} = ความถี่ของการทำงาน

F_{PWM} = ความถี่ของสัญญาณ

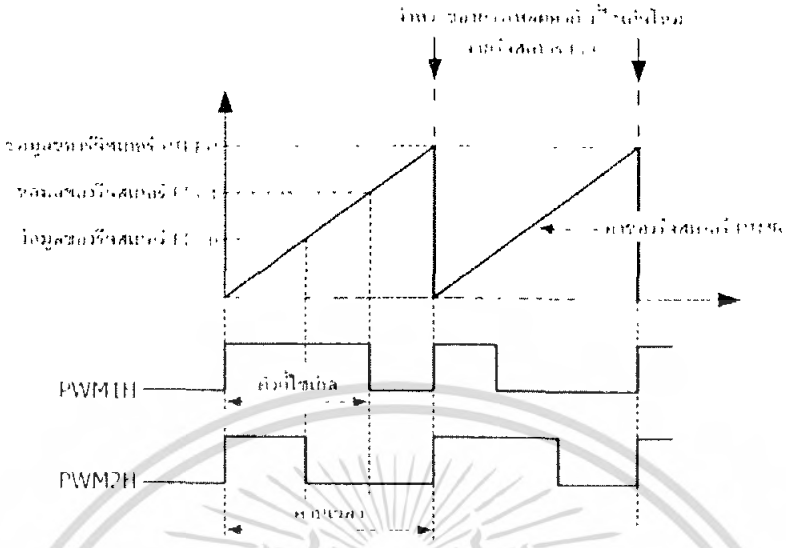
4. โหมดการทำงานของส่วนกำเนิดสัญญาณ PWM ในโมดูล MCPWM

มีด้วยกัน 4 แบบหลักคือ

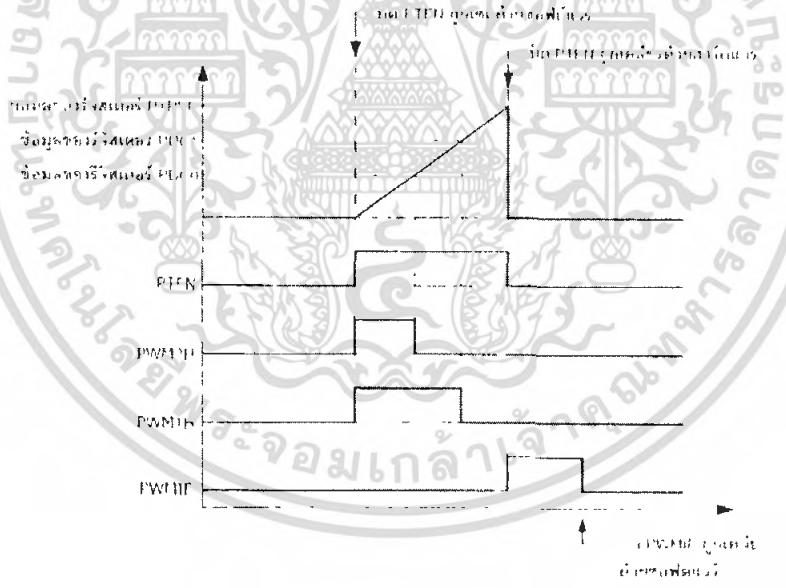
1. โหมดปรับขอบสัญญาณ (Edge aligned mode)
2. โหมดสัญญาณเดี่ยว (Single event mode)
3. โหมดปรับสัญญาณกึ่งกลาง (Center aligned mode)
4. โหมดปรับสัญญาณกึ่งกลางและปรับปรุงค่า(Center aligned mode with double updates) ซึ่งสัมพันธ์กับโหมดการทำงานของฐานเวลา PWM โดย

- เมื่อฐานเวลา PWM ทำงานในโหมดเปลี่ยนแปลงค่าอิสระ ส่วนกำเนิดสัญญาณ PWM จะทำงานในโหมดปรับขอบสัญญาณ
- เมื่อฐานเวลา PWM ทำงานในโหมดทำงานครั้งเดียว ส่วนกำเนิดสัญญาณ PWM จะทำงานในโหมดสัญญาณเดี่ยว
- เมื่อฐานเวลา PWM ทำงานในโหมดนับค่าขึ้นลงอย่างต่อเนื่อง ส่วนกำเนิดสัญญาณ PWM จะทำงานในโหลดปรับสัญญาณกึ่งกลาง
- เมื่อฐานเวลา PWM ทำงานในโหมดนับค่าขึ้นลงอย่างต่อเนื่อง พร้อมปรับปรุงค่า ส่วนกำเนิดสัญญาณ PWM จะทำงานในโหมดปรับสัญญาณกึ่งกลางพร้อมปรับปรุงค่า

การทำงานของส่วนกำเนิดสัญญาณ PWM ในโหมดปรับขอบสัญญาณ



รูปที่ 2.17 ไดอะแกรมเวลาแสดงการเกิดสัญญาณ PWM เมื่อทำงานในโหมดปรับขอบสัญญาณ



รูปที่ 2.18 ไดอะแกรมเวลาแสดงการเกิดสัญญาณ PWM เมื่อทำงานในโหมดสัญญาณเดี่ยว

เป็นไดอะแกรมเวลาแสดงการเกิดสัญญาณ PWM เมื่อทำงานในโหมดปรับขอบสัญญาณเริ่มต้นด้วยการกำหนดให้ฐานเวลา PWM ทำงานในโหมดเปลี่ยนแปลงค่าอิสระ ดังนั้นคาบเวลาของสัญญาณ PWM จะถูกกำหนดโดยค่าที่โหลดให้แก่รีจิสเตอร์ PTPER ส่วนดิวตี้ไซเกิลถูกกำหนดโดยค่าในรีจิสเตอร์ PDCx

ในกรณีที่ค่าดิวตี้ไซเกิลไม่เป็นศูนย์ วงจรเอาต์พุตของส่วนกำเนิดสัญญาณ PWM ทุกชุดที่ได้รับการเอ็นเอเบิล จะทำงานที่จุดเริ่มต้นของคาบเวลาของสัญญาณ PWM หรือเมื่อค่าในรีจิสเตอร์ PTMR เท่ากับศูนย์ และหยุดทำงานเมื่อค่าของรีจิสเตอร์ PTMR ตรงกับค่าดิวตี้ไซเกิลของส่วนกำเนิดสัญญาณ PWM

ถ้าหากค่าในรีจิสเตอร์PDCxเป็นศูนย์ วงจรเอาต์พุตของส่วนกำเนิดสัญญาณ PWM จะไม่ทำงานใดๆ นั่นหมายความว่าในโหมดนี้จะสามารถกำเนิดสัญญาณ PWM ได้ก็ต่อเมื่อค่ารีจิสเตอร์ PDCx ต้องมากกว่าค่าที่กำหนดในรีจิสเตอร์ PTPER

2.6 หลักการเบื้องต้นของภาษาซี

2.6.1 กล่าวนำ

ภาษาซีเป็นภาษาที่ถือกำเนิดมายาวนาน โดยแต่เดิมนั้นภาษาซีถูกพัฒนาขึ้นเพื่อให้เป็นภาษาที่ใช้สำหรับการสร้างระบบปฏิบัติการยูนิกซ์ เนื่องจากในขณะนั้นระบบปฏิบัติการยูนิกซ์เขียนด้วยภาษาแอสเซมบลี(Assembly)ซึ่งเป็นภาษาที่ยึดติดกับฮาร์ดแวร์ของเครื่อง ดังนั้นการที่จะย้ายระบบปฏิบัติการไปใช้กับเครื่องอื่นจึงเป็นเรื่องเป็นไปไม่ได้เลย ซึ่งนับเป็นข้อเสียที่ค่อนข้างใหญ่สำหรับภาษาแอสเซมบลี ดังนั้นภาษาซีซึ่งเป็นภาษาที่ไม่ยึดติดกับฮาร์ดแวร์จึงถูกพัฒนาขึ้นมา ในปัจจุบันภาษาซีไม่ได้จำกัดอยู่เพียงแค่งานสร้างระบบปฏิบัติการเท่านั้น แต่สามารถนำไปใช้สร้างโปรแกรมเพื่องานในทุกประเภทเช่น งานเกี่ยวกับการคำนวณ ควบคุมการทำงานของอุปกรณ์หรือฮาร์ดแวร์ชนิดต่าง ๆ หรือสร้างโปรแกรมสำหรับจัดพิมพ์เอกสาร เป็นต้น

2.6.2 การสั่งงานคอมพิวเตอร์ด้วยภาษาโปรแกรม

ภาษาที่เครื่องคอมพิวเตอร์เข้าใจเรียกว่า ภาษาเครื่อง (Machine Languagc) ซึ่งอยู่ในรูปแบบของรหัสเลขฐานสอง(Binary Codc)ซึ่งประกอบไปด้วยตัวเลขอยู่เพียงแค่ 2 ตัวคือ 0 และ 1 เรียงสลับและต่อกันเป็นความหมายที่เครื่องเข้าใจ

ต่อมาจึงมีการพัฒนาภาษาสำหรับสื่อสารกับคอมพิวเตอร์ที่มนุษย์เข้าใจ โดยใช้ตัวอักษรภาษาอังกฤษมากำหนดเป็นรูปแบบคำสั่งซึ่งส่วนใหญ่เป็นคำที่มีความหมายในภาษาอังกฤษ แต่การที่จะทำให้คอมพิวเตอร์เข้าใจภาษาที่พัฒนาขึ้นมาได้นั้น ต้องมีตัวกลางเพื่อทำหน้าที่แปลภาษานั้นเป็นภาษาเครื่องอีกทีซึ่งสามารถแบ่งระดับตามลักษณะและการทำงานของแต่ละภาษาได้เป็น2ระดับดังนี้

2.6.2.1 ภาษาระดับต่ำ (Low Level Language)

เป็นภาษาที่ใกล้เคียงกับภาษาเครื่องมากที่สุด สามารถเขียนคำสั่งเพื่อติดต่อสั่งงานกับอุปกรณ์ฮาร์ดแวร์ได้โดยตรง โปรแกรมที่เขียนด้วยภาษาระดับต่ำจะทำงานได้รวดเร็วมากแต่

ปัญหาของภาษาระดับต่ำก็คือเขียนยาก รูปแบบคำสั่งจดจำและทำความเข้าใจได้ยาก นอกจากนี้อีกเหตุผลที่สำคัญคือ ภาษาระดับต่ำยึดติดกับชนิดของเครื่องคอมพิวเตอร์ นั่นหมายถึงถ้าเราเขียนโปรแกรมด้วยภาษาระดับต่ำสำหรับคอมพิวเตอร์แบบหนึ่ง โปรแกรมนั้นจะไม่สามารถนำไปใช้กับคอมพิวเตอร์อีกแบบได้ ตัวอย่างของภาษาระดับต่ำได้แก่ แอมเซมบลี

2.6.2.2 ภาษาระดับสูง (High Level Language)

เป็นภาษาที่ใกล้เคียงกับภาษาที่ใช้สื่อสารกันตามปกติ คำสั่งต่าง ๆ มักเป็นคำที่มีความหมายในภาษาอังกฤษ ทำให้จดจำและเขียนได้ง่าย แต่ข้อเสียของภาษาระดับสูงก็คือมักไม่มีคำสั่งในการติดต่อกับอุปกรณ์ฮาร์ดแวร์โดยตรงและทำงานได้ช้ากว่าภาษาระดับต่ำ ตัวอย่างของภาษาระดับสูงได้แก่ Pascal Cobol หรือ Basic เป็นต้น

เนื่องจากภาษาระดับต่ำและระดับสูงต่างก็มีข้อดีและข้อเสียแตกต่างกัน ดังนั้นภาษาซีจึงพัฒนาขึ้นมาโดยปรับปรุงข้อเสียและนำเอาข้อดีของภาษาทั้ง 2 ระดับมาใช้ โดยคำสั่งของภาษาซีเป็นคำที่มีความหมายในภาษาอังกฤษ สามารถจดจำและเขียนได้ง่ายเหมือนกับภาษาระดับสูงแต่ภาษาซีทำงานได้อย่างรวดเร็ว และมีคำสั่งที่ให้ผู้เขียนโปรแกรมเรียกใช้เพื่อติดต่อสั่งงานกับอุปกรณ์ฮาร์ดแวร์โดยตรงเหมือนกับภาษาระดับต่ำ ดังนั้นภาษาซีจึงถูกจัดให้เป็นภาษาระดับกลาง

2.6.3 จุดเด่นของภาษาซี

ในปัจจุบัน ภาษาซีได้รับการยอมรับและใช้งานกันอย่างกว้างขวาง เนื่องมาจาก

- ภาษาซีเป็นภาษาที่มีการกำหนดมาตรฐานสำหรับเครื่องคอมพิวเตอร์ทุกรุ่น และระบบปฏิบัติการทุกชนิด ทำให้โครงสร้างทางภาษาฟังก์ชันไลบรารี(Library) ต่างๆ สามารถนำไปใช้งานระหว่างเครื่องแต่ละรุ่นและระบบปฏิบัติการแต่ละชนิดได้
- โปรแกรมที่เขียนขึ้นด้วยภาษาซีมีขนาดเล็กและทำงานได้เร็ว
- ภาษาซีมีโครงสร้างทางภาษาที่ดีและเครื่องหมายสำหรับดำเนินการ ไม่ว่าจะเป็นการคำนวณทางคณิตศาสตร์ ตรรกศาสตร์หรือการเปรียบเทียบมีประสิทธิภาพการทำงานสูง
- สามารถเขียนคำสั่งภาษาซี เพื่อควบคุมการทำงานของอุปกรณ์ฮาร์ดแวร์บางส่วนได้
- มีฟังก์ชันสำเร็จรูปสำหรับงานประเภทต่าง ๆให้เลือกใช้มากมาย ซึ่งช่วยประหยัดเวลาในการเขียนคำสั่ง นอกจากนั้นถ้าฟังก์ชันที่ภาษาซีเตรียมไว้ให้ใช้งานได้ไม่ตรงดามต้องการทั้งหมดเราสามารถเขียนคำสั่งเพิ่มเติมลงไปได้

บทที่ 3
การสร้างและการออกแบบ

การสร้างและการออกแบบปริญญานิพนธ์นี้ ใช้หลักการคิดและการออกแบบ โดยได้นำหลักการคิดเป็นลำดับมาจากแผนภาพบล็อกไดอะแกรม ซึ่งแบ่งออกได้เป็นสองส่วนคือ ส่วนที่เป็น User Interface และส่วนที่นำข้อมูลที่ได้จากผู้ใช้งานมาแสดงผล อธิบายได้ดังนี้

3.1 ส่วน User Interface

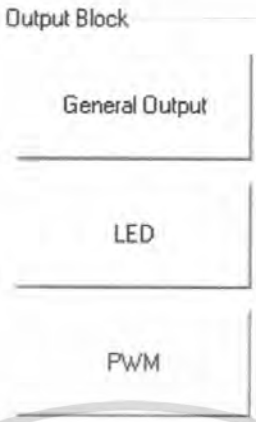
ส่วนที่เป็น User Interface คือ ส่วนที่ใช้ติดต่อกับผู้ใช้งานโดยใช้โปรแกรมVisual Basic สร้างหน้าต่างที่ประกอบไปด้วยบล็อกคำสั่งเพื่อให้ผู้ใช้งานเลือกลำดับการทำงาน ซึ่งผู้ใช้งานสามารถเลือกบล็อกที่ออกแบบ ในหน้าจอ User Interface และป้อนค่าภายในแต่ละบล็อก ได้ตามที่ผู้ใช้งานต้องการ เพื่อที่ให้โปรแกรมประมวลผลต่อไปและแสดงผลออกมาตามต้องการ

3.1.1 รูปแบบของบล็อกที่ใช้งาน แบ่ง BLOCK WINDOW ออกเป็น 2 ส่วน

1. Input Block ได้แก่ ADC, General Input, UART, Delay

รูปที่ 3.1 อินพุตบล็อก

2. Output Block ได้แก่ General Output, LED, UART, PWM



รูปที่ 3.2 เอาท์พุตบล็อก

3. Function List เป็นพื้นที่ที่ใช้ในการเก็บฟังก์ชัน

รูปที่ 3.3 Function List

4. Function Control ได้แก่ Save, Load, Edit, Properties

รูปที่ 3.4 Function Control

- Save เพื่อใช้ในการเก็บบันทึกการทำงานของฟังก์ชันที่ใช้งานไปยัง Function List
- Load เพื่อใช้ในการเรียกฟังก์ชันจาก Function List ที่บันทึกไว้มาใช้งาน
- Edit เพื่อใช้แก้ไขค่าของฟังก์ชัน
- Properties เพื่อแสดงรายละเอียดต่าง ๆ ของฟังก์ชัน

3.1.2 MENU BAR

1. FILE
- Run คือ คำสั่งทั้งหมดที่ถูกเซตค่าเพื่อส่งข้อมูลไปประมวลผล
- Remove all คือ คำสั่งที่ต้องการลบบล็อกทั้งหมดที่ไม่ต้องการใช้งาน
- Save คือ คำสั่งที่ใช้เก็บข้อมูลทั้งหมด
- Load คือ คำสั่งที่เรียกข้อมูลที่เก็บไว้ขึ้นมาใช้
- Exit คือ คำสั่งออกจากโปรแกรม

2. SETTING
- Comport คือ หมายเลขพอร์ตที่ใช้ติดต่อ RS-232
- Baud Rate คือ ความเร็วในการส่งข้อมูล ซึ่งมีหน่วยเป็นบิตต่อวินาที (bps)
- Data bit คือ บิตข้อมูล
- Stop bit คือ ส่วนปิดท้ายเพื่อระบุบิตสุดท้ายของข้อมูล
- Connect คือ คำสั่งที่ใช้เชื่อมต่อกับไมโครคอนโทรลเลอร์ผ่าน Serial port

3.2 ส่วนที่นำข้อมูลจากผู้ใช้มาแสดงผล

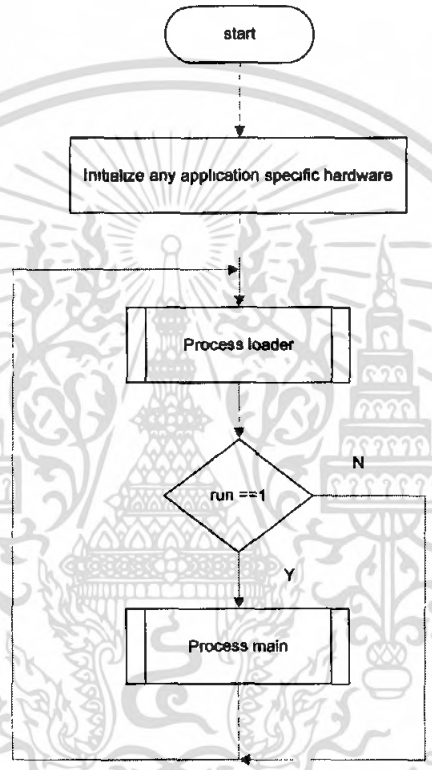
เป็นการทดลองการเชื่อมต่อระหว่างไมโครคอนโทรลเลอร์กับหน้า User Interface โดยการเขียนโปรแกรม ได้มีการเลือกใช้ไมโครคอนโทรลเลอร์ dsPIC30F6010A เป็นบอร์ดที่ใช้รองรับหน้าจอ User Interface โดยใช้โปรแกรม Visual Basic สร้างหน้าต่างที่ประกอบไปด้วยบล็อกคำสั่งเพื่อให้ผู้ใช้งานเลือกลำดับการทำงานตามกระบวนการที่ต้องการควบคุม และสามารถSettingค่า

ต่างๆ ภายในบล็อกได้ จากนั้นคำสั่งทั้งหมดจะถูกส่งไปยังบอร์ดไมโครคอนโทรลเลอร์ที่ใช้ตัวประมวลผลตระกูล dsPIC30F6010A ซึ่งใช้สำหรับแสดงผลหรือแก้ไขการทำงาน

3.3 การออกแบบ

ในขั้นตอนการออกแบบการทำงานของโปรแกรมภาษาบล็อก จะมีโปรแกรมการทำงานดังนี้

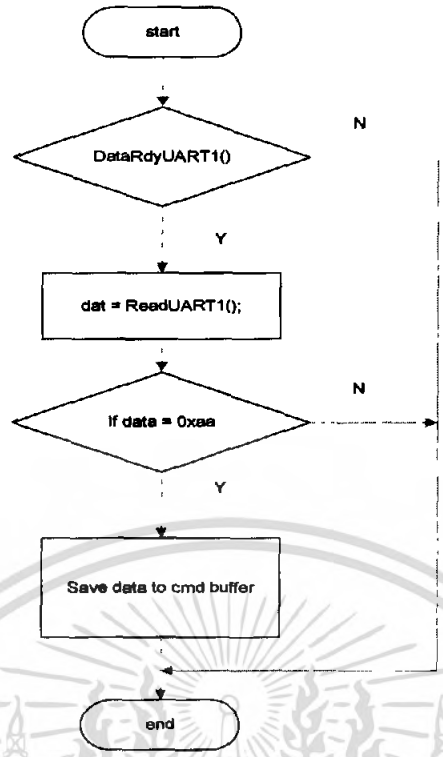
3.3.1 Flow Chart แสดงการทำงานในส่วน Microcontroller



รูปที่ 3.5 แผนผังแสดงการทำงานในส่วน Microcontroller

อธิบายแผนผังการทำงาน

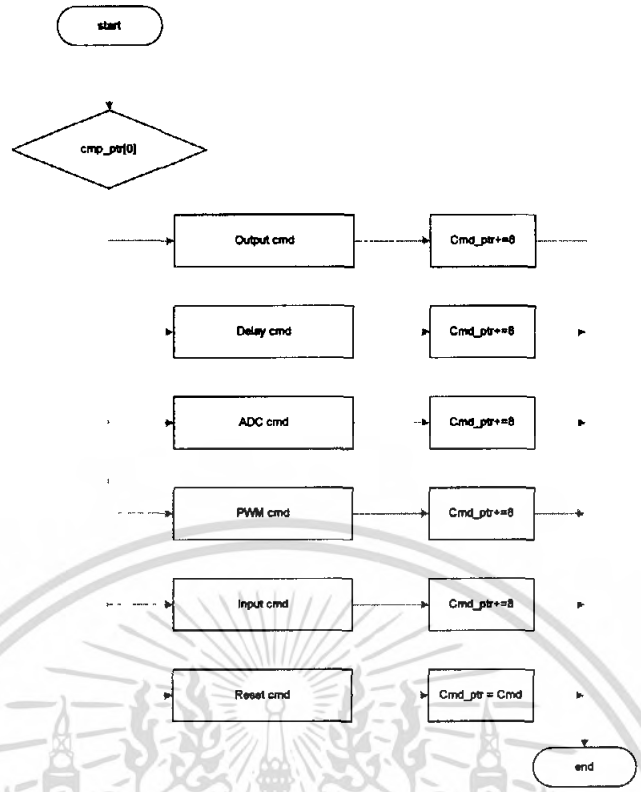
ในเมนูโปรแกรมหลังจากที่ทำการเริ่มทำงานแล้ว จะทำการเซ็ทค่าเริ่มต้นต่างๆ หลังจากนั้นจะทำทำการรับค่าสั่งต่างๆเข้ามาบรรจุใน Process loader ซึ่งจะเป็นการเช็คค่าคอนดิชัน คือถ้าค่าตรงก็จะทำในส่วนที่เป็นจริง แต่ถ้าค่าเป็นเท็จก็จะทำการออกไปรับค่ามาให้เพื่อจะนำมาบรรจุในส่วนของ Process loader อีกครั้งทำอย่างนี้ไปเรื่อย ๆ จนกว่าจะคอนดิชันเป็นจริง และถ้าเป็นจริงแล้วจะนำค่านั้นบรรจุใน Process main แล้วกระทำตามคำสั่งที่ได้รับมาเป็นลำดับต่อไปเมื่อเสร็จสิ้นก็จะไปรอรับค่าจาก Process loader และกระทำอย่างนี้ต่อไป



รูปที่ 3.6 แผนผังแสดงการทำงานย่อย Process Loader

อธิบายแผนผังการทำงาน

เมื่อเริ่มเข้ามาในส่วนนี้จะทำการรอรับค่าที่จะส่งมาจากคอมพิวเตอร์ ถ้าไม่มีค่าส่งมาก็จะข้ามโปรแกรมในส่วนนี้ไป แต่ถ้ามีจะนำมาเก็บในตัวแปรdat เพื่อเปรียบเทียบว่าตรงตามคอนดิชันที่กำหนดไว้หรือไม่ ถ้าตรงจะทำทำการเซฟข้อมูลชุดนั้นเก็บเอาไว้ แต่ถ้าไม่ตรงจะทำการข้ามในโปรแกรมส่วนย่อยนี้ไปก็จะจบในส่วนโปรแกรมย่อยนี้

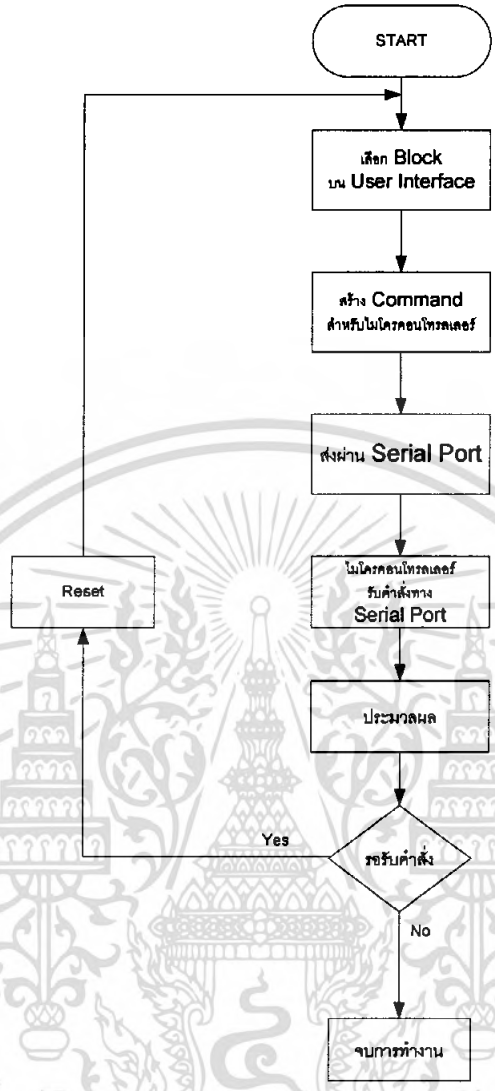


รูปที่ 3.7 แผนผังแสดงการทำงานย่อย Process Main

อธิบายแผนผังการทำงาน

เมื่อเริ่มเข้ามาทำงานในส่วนโปรแกรมย่อยนี้แล้ว จะนำค่าที่ได้จากตัวแปรที่เก็บไว้มาเปรียบเทียบว่าตรงกับฟังก์ชันย่อยชนิดไหน แล้วก็จะทำงานตามฟังก์ชันย่อยนั้น แล้วออกจากโปรแกรมในส่วนนี้

3.3.2 Flow Chart การทำงานในส่วน Microcontroller ร่วมกับ User Interface

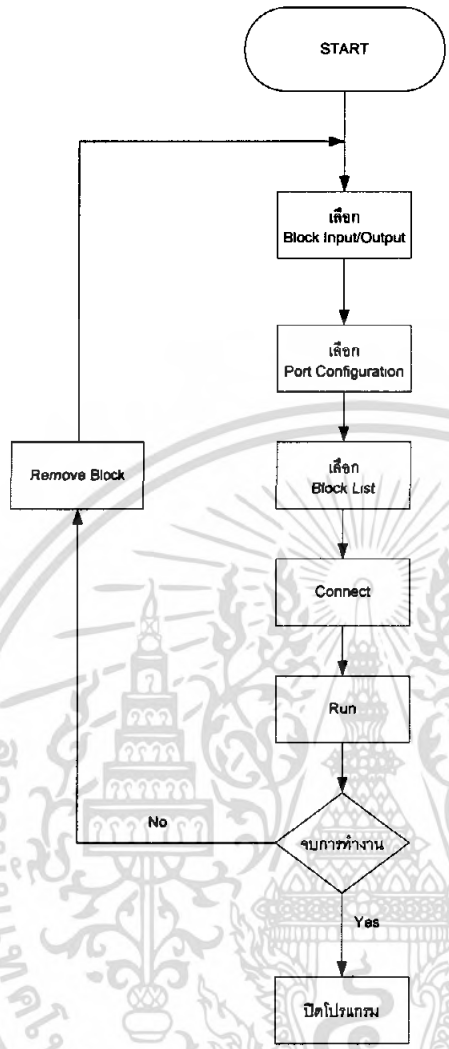


รูปที่ 3.8 แผนผังแสดงการทำงานในส่วน Microcontroller ร่วมกับ User Interface

อธิบายแผนผังการทำงานของภาษาบล็อก

เริ่มต้นด้วยการเลือกบล็อกที่ต้องการออกแบบระบบบนUser Interface โดยมีการสร้างคำสั่งให้กับไมโครคอนโทรลเลอร์ ซึ่งทั้งสองส่วนระหว่างไมโครคอนโทรลเลอร์ต้องส่งและรับคำสั่งเดียวกัน ซึ่งจะติดต่อกันผ่าน Serial Port หลังจากนั้นไมโครคอนโทรลเลอร์จะรับคำสั่งที่ได้จาก Serial Port ไปประมวลผล และรอรับคำสั่งต่อไปว่าจะให้หยุดการทำงานและออกจากโปรแกรม. หรือว่ารับคำสั่งใหม่ที่ได้จากกระบวนการอื่นอีก โดยเมื่อมีการสร้างกระบวนการใหม่นั้นต้องมีการรีเซ็ตค่าจากบอร์ดทดลองก่อน จึงจะได้ผลการทดลองออกมาถูกต้องตามต้องการ

3.3.3 Flow Chart การทำงานของโปรแกรมวิชวลเบสิก



รูปที่ 3.9 แผนผังแสดงการทำงานของโปรแกรมวิชวลเบสิก

อธิบายแผนผังการทำงานของโปรแกรมวิชวลเบสิก

เริ่มต้นด้วยการเลือก Block Input /Output จาก Block Window ที่ต้องการ เลือก Port Configuration จากนั้นเลือก Port Block List ซึ่งจะเป็นบล็อกที่แสดงการเชื่อมต่อจาก Block Input หลังจากนั้นกดปุ่ม Connect เพื่อทำการเชื่อมต่อกับไมโครคอนโทรลเลอร์ผ่าน Serial port กดปุ่ม Run เพื่อส่งข้อมูลไปยังไมโครคอนโทรลเลอร์ เมื่อผู้ใช้งานต้องการสร้างกระบวนการอื่นอีกให้กดปุ่ม Remove all เพื่อลบบล็อกของกระบวนการเดิมแล้วเลือกบล็อกใหม่ แต่ถ้าเสร็จสิ้นกระบวนการแล้วให้ออกจากโปรแกรมโดยการกดปุ่ม Exit

บทที่ 4

การทดลองและผลการทดลอง

4.1 การทดลอง

เมื่อทำการออกแบบระบบที่ต้องการใช้งานลงบน User Interface แล้ว จากนั้นชุดคำสั่งที่ถูกออกแบบไว้จะถูกส่งไปยังไมโครคอนโทรลเลอร์ ไมโครคอนโทรลเลอร์จะทำการประมวลผลชุดคำสั่งเหล่านั้นแล้วแสดงผลลัพธ์ออกมาทางบอร์ดทดลอง

4.2 ผลการทดลอง

ในการทดลองนี้จะเป็นการทดลองเพื่อแสดงความสามารถของภาษาบล็อกที่สามารถควบคุมการทำงานของไมโครคอนโทรลเลอร์ด้วยโปรแกรม Visual Basic โดยสามารถดูได้จากส่วนต่างๆต่อไปนี้คือ

1. หน้าจอใช้งานที่ใช้ในการทดลองมี

- LED
- DELAY
- UART
- ADC
- PWM

2. การใช้บล็อก

- เลือกบล็อกโดยการคลิกบล็อกที่ใช้งาน
- การกำหนดค่าให้กับบล็อก
- การเก็บบันทึกฟังก์ชั่น
- การเรียกใช้งานฟังก์ชั่น
- การ Remove all บล็อก
- การ Save และ Load การทำงานทั้งหมด

3. การเชื่อมต่อ

การเชื่อมต่อระหว่าง Visual Basic กับ ไมโครคอนโทรลเลอร์ โดยการกดปุ่ม Connect และส่งข้อมูลไปยังไมโครคอนโทรลเลอร์โดยการกดปุ่ม Run

4. ไมโครคอนโทรลเลอร์

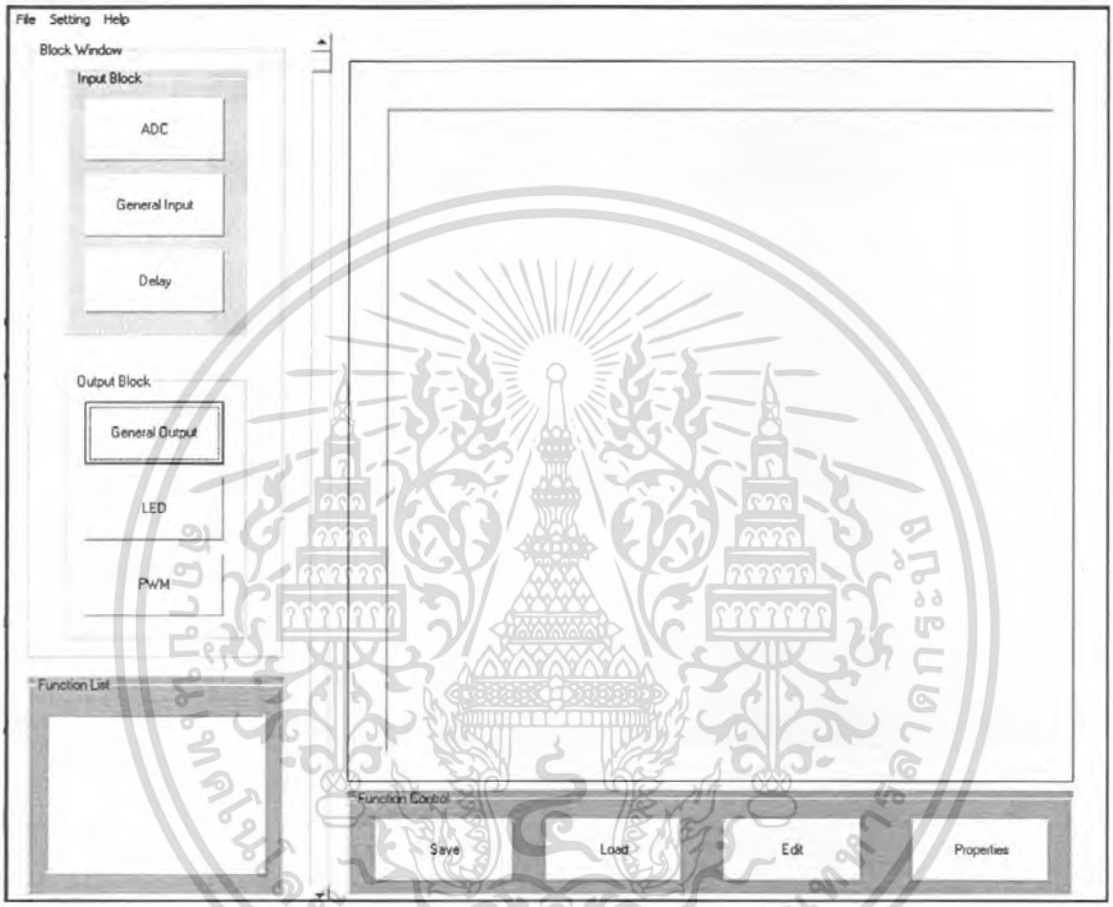
ไมโครคอนโทรลเลอร์จะแสดงผลการทำงานจากการรับคำสั่งจากผู้ใช้งานทาง User Interface

5. ตัวอย่างการทำงาน

4.2.1 ลำดับการทดลอง

4.2.1.1 หน้าจอใช้งาน

หน้าจอใช้งานที่ใช้ในการทดลองมีดังต่อไปนี้

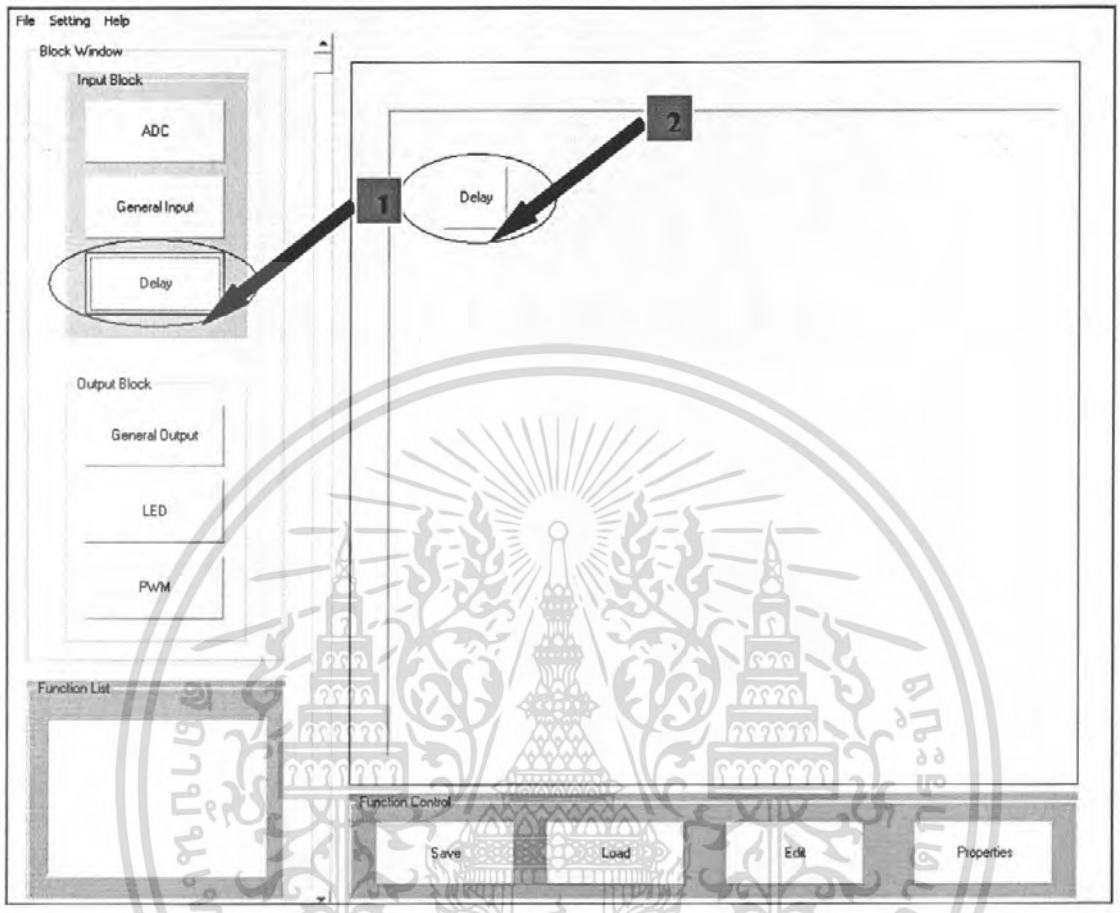


รูปที่ 4.1 แสดงหน้าจอใช้งาน

จากรูปที่ 4.1 แสดงหน้าจอ User Interface ที่ใช้ในการออกแบบระบบตามที่ผู้ใช้งานต้องการ โดยที่ด้านซ้ายของหน้าต่างจะแสดงถึง Control Form ต่าง ๆ เพื่อใช้ในการออกแบบระบบ

4.2.1.2 การเลือกบล็อก

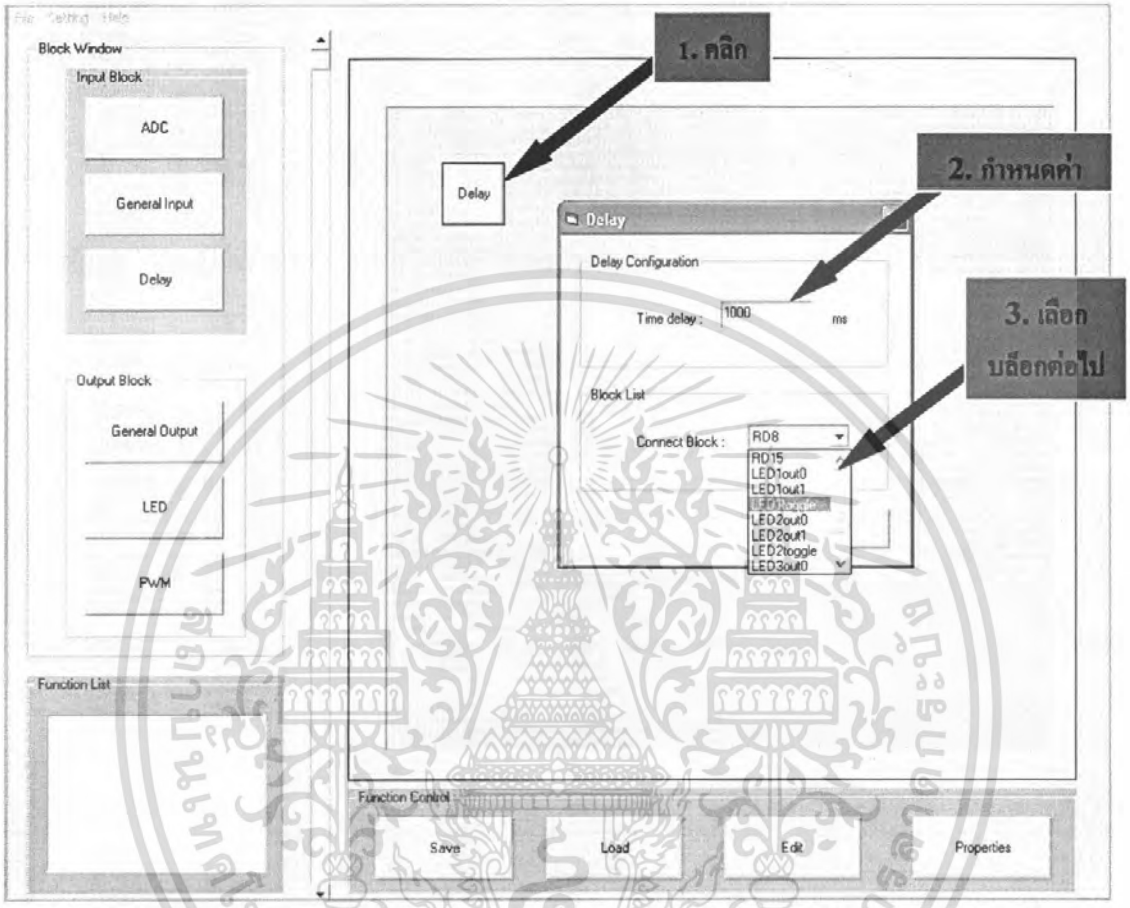
- การเลือกบล็อกทำได้โดยการคลิกบล็อกที่ Block Window ทางด้านซ้ายของ User Interface จากนั้นบล็อกที่เลือกจะมาปรากฏอยู่บน Control Area ทางด้านขวา



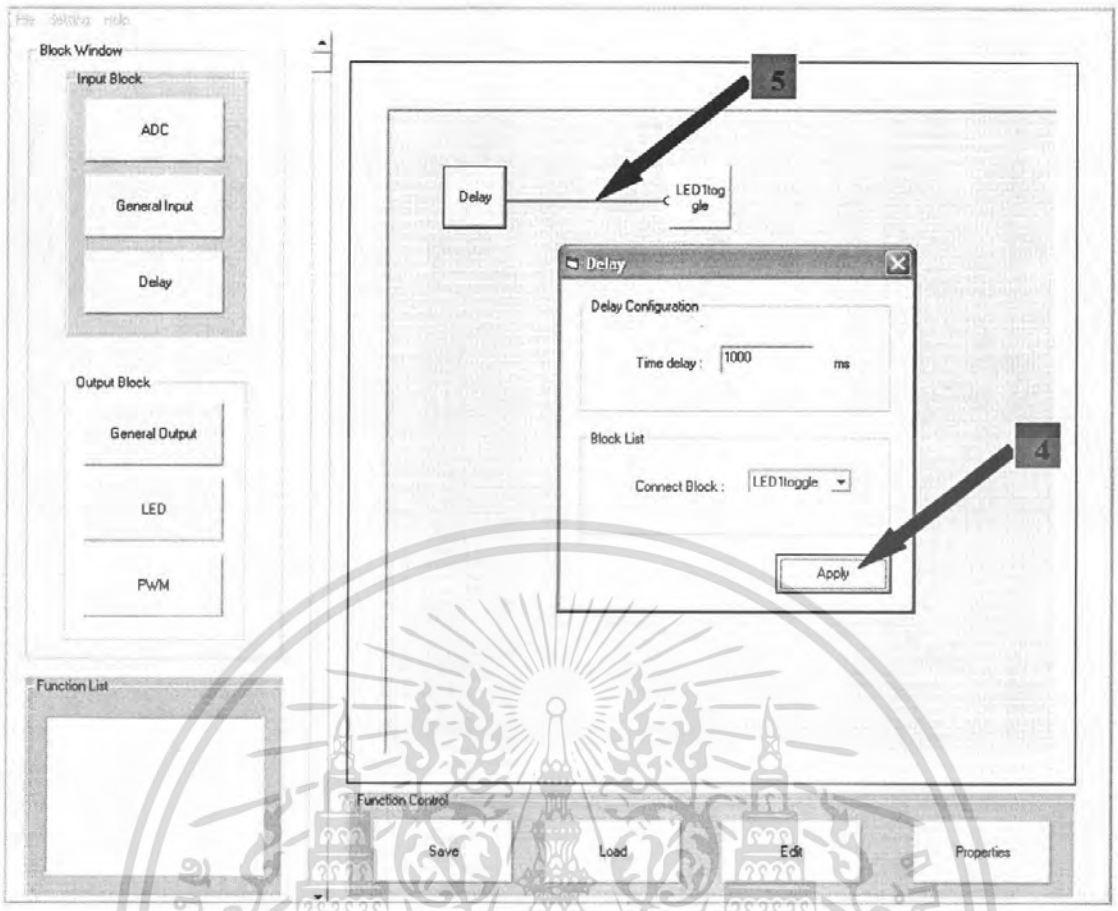
รูปที่ 4.2 ขั้นตอนการเลือกบล็อก

- การกำหนดค่าให้กับบล็อก

การเชื่อมต่อกันระหว่างบล็อก โดยมีเส้นเป็นตัวเชื่อมต่อจะบอกถึงทิศทางในการทำงานที่มีความสัมพันธ์กันทางด้านอินพุต/เอาท์พุต



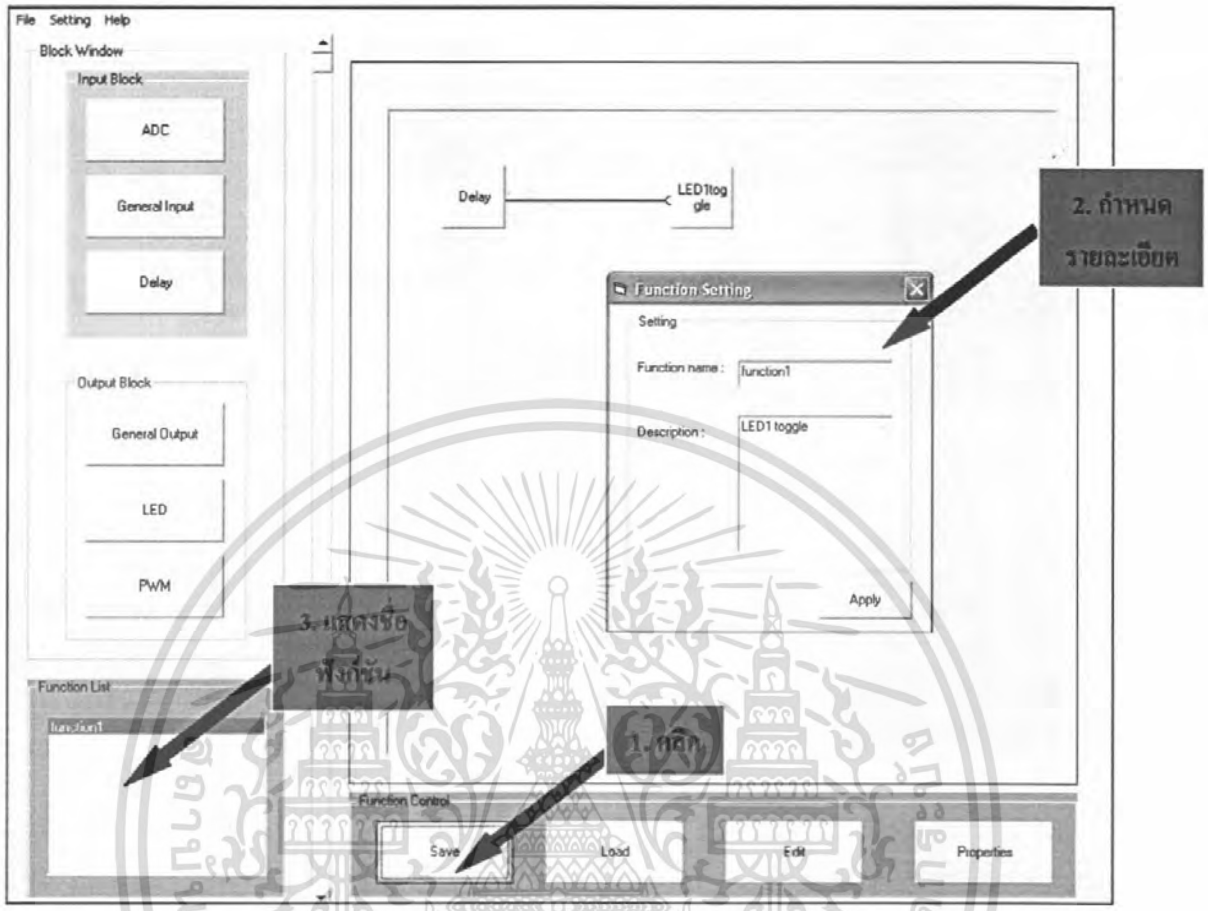
รูปที่ 4.3 แสดงการเซตค่าเพื่อเลือก อินพุต/เอาท์พุต ต่อให้เพื่อเส้นแสดง



รูปที่ 4.4 แสดงเส้นเชื่อมต่อบล็อก

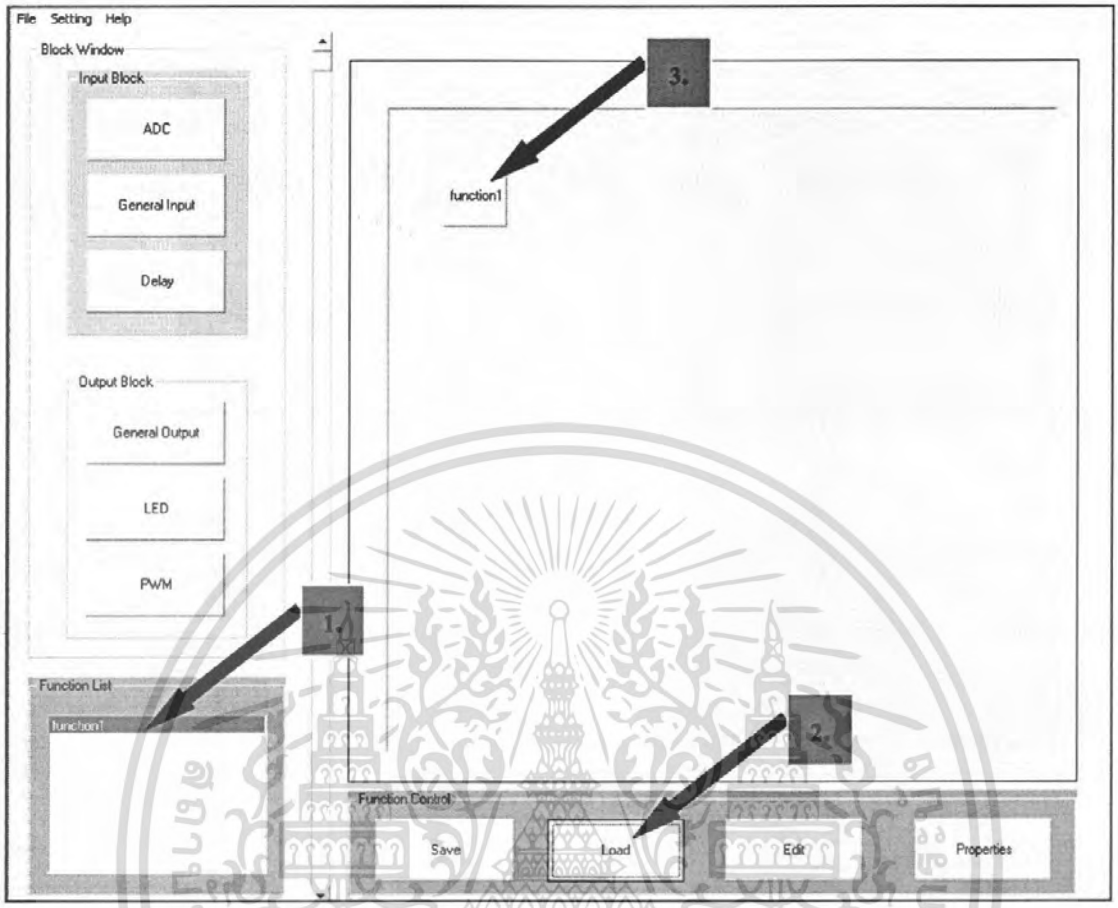
1. คลิกบล็อกที่เลือกมาเพื่อกำหนดค่าให้กับบล็อก
2. กำหนดค่าให้กับบล็อกนั้น
3. เลือกบล็อกต่อไปที่ต้องการจะเชื่อมต่อเส้นกับบล็อกที่ 1 จาก Block List
4. กดปุ่ม Apply
5. เส้นเชื่อมต่อกับบล็อกต่อไปที่เลือกจะปรากฏขึ้น

- การเก็บบันทึกฟังก์ชัน



รูปที่ 4.5 แสดงการเก็บบันทึกฟังก์ชั่น

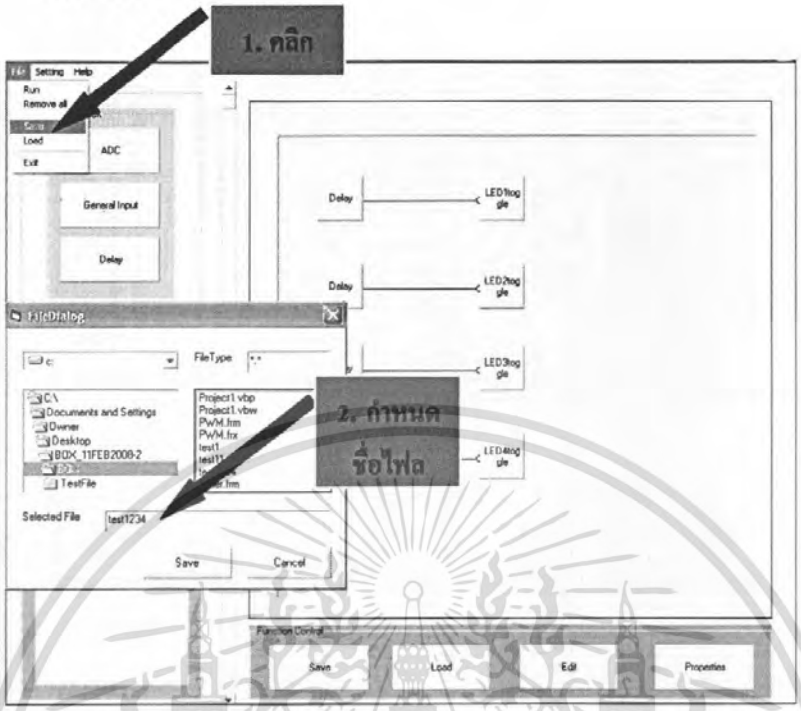
- การเรียกใช้งานฟังก์ชัน



รูปที่ 4.6 แสดงการเรียกใช้งานฟังก์ชัน

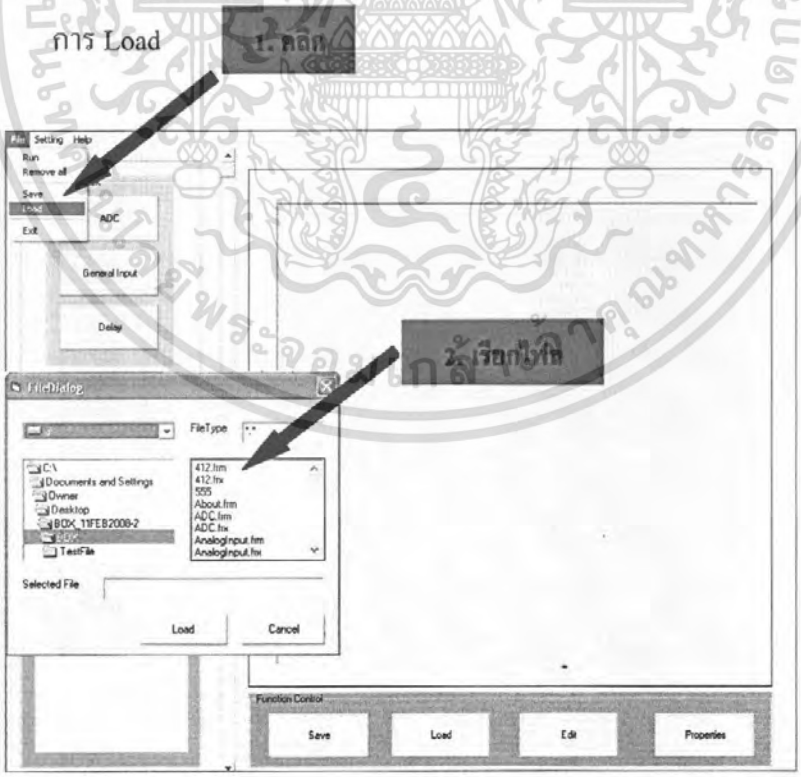
- การ Save และ Load การทำงานทั้งหมด

การ Save



รูปที่ 4.7 หน้าจอแสดงการ Save

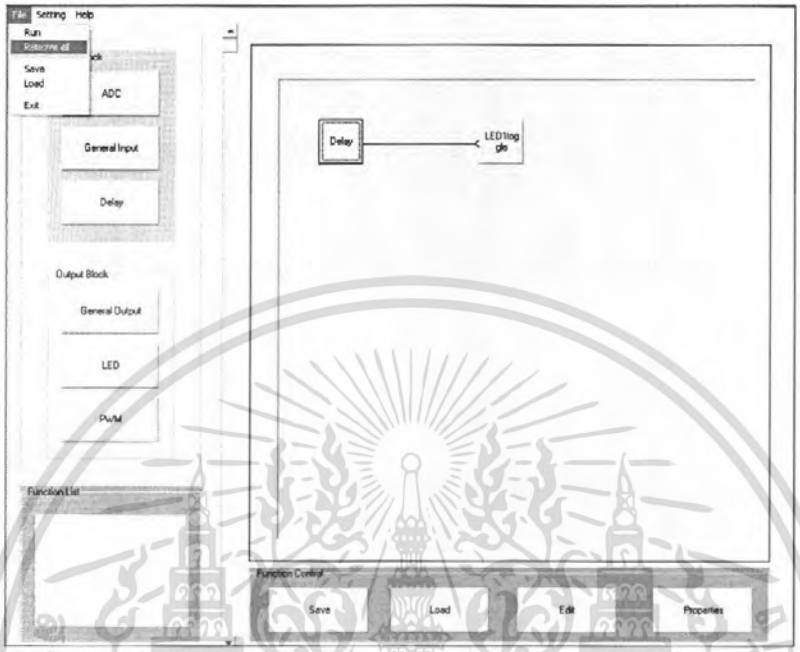
การ Load



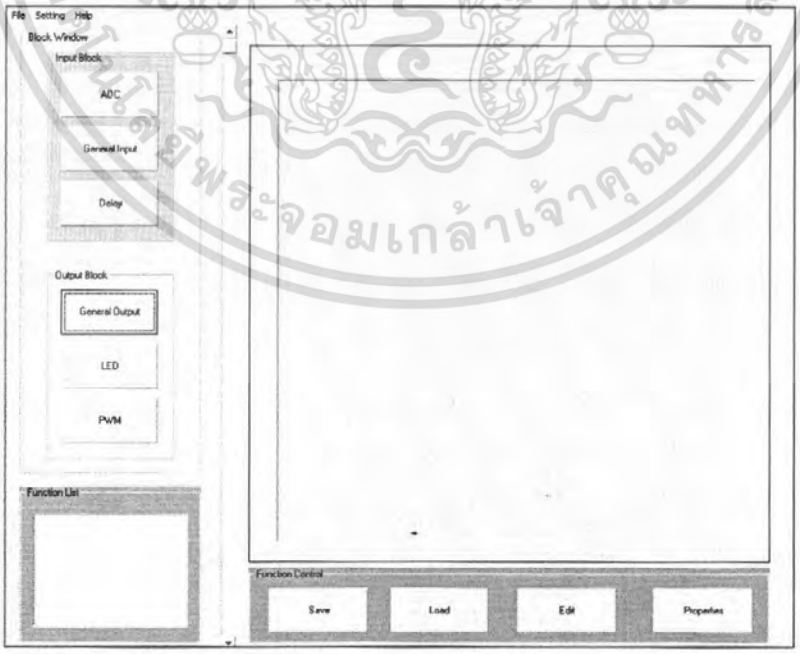
รูปที่ 4.8 หน้าจอแสดงการ Load

- การ Remove บล็อก

การ Remove บล็อกที่ไม่ต้องการทำได้ โดยที่ต้องการเข้าไปที่เมนู File แล้วกดปุ่ม Remove all จากนั้นบล็อกที่ไม่ต้องการทั้งหมดก็จะหายไป



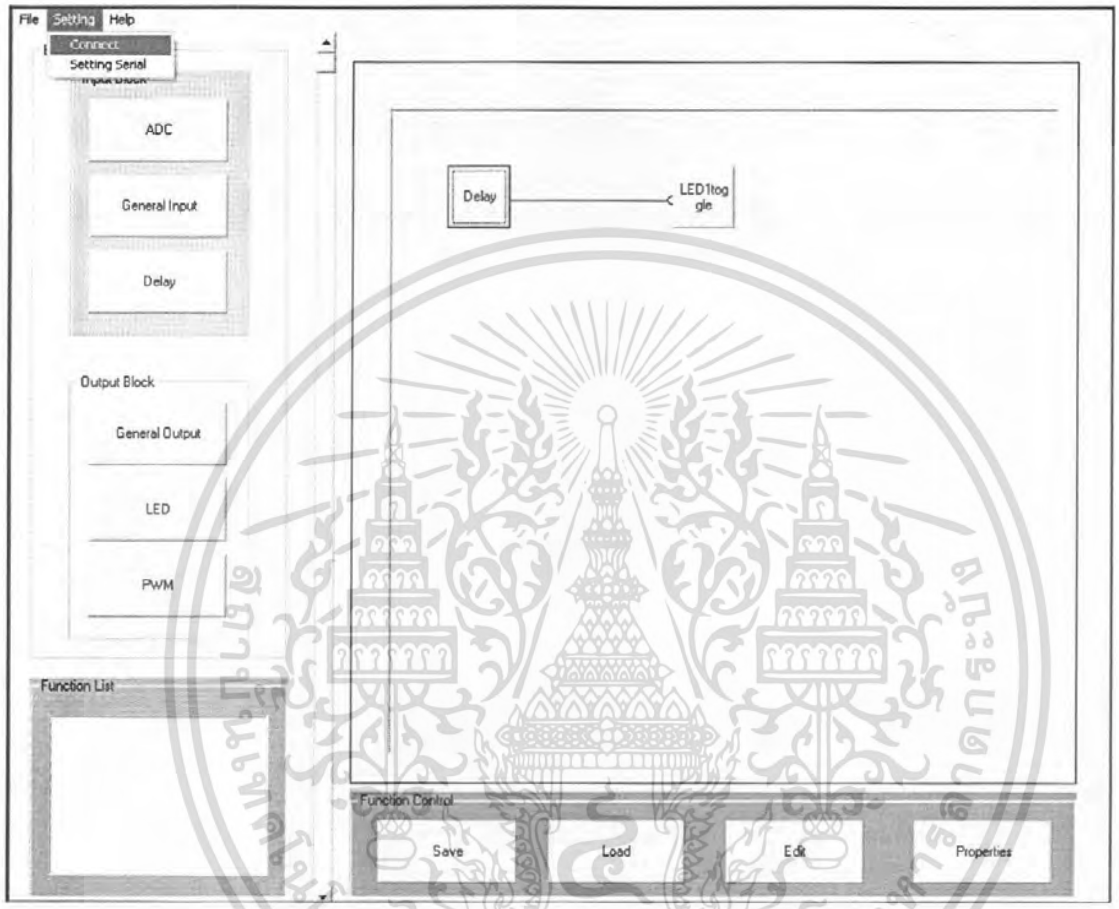
รูปที่ 4.9 แสดงการ Remove บล็อก



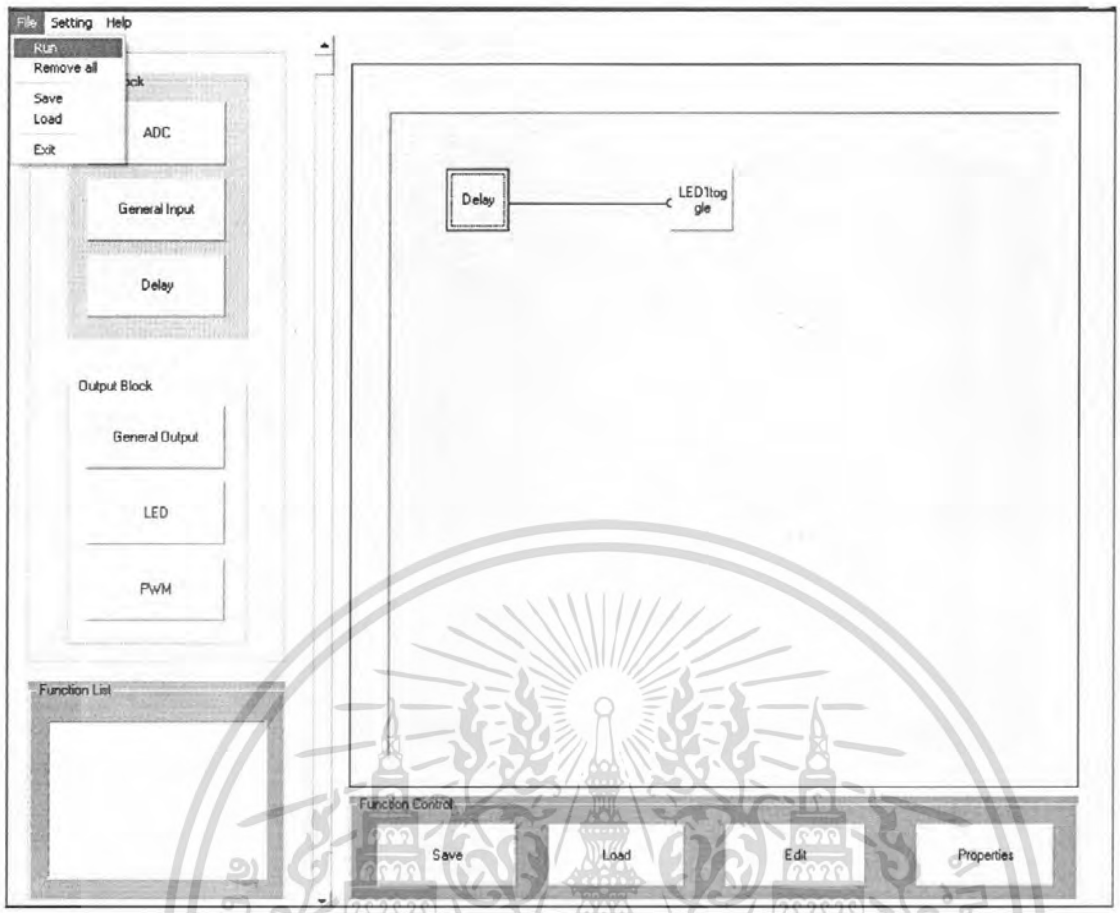
รูปที่ 4.10 หน้าจอที่เกิดจากการ Remove บล็อก

4.2.1.3 การเชื่อมต่อ

การเชื่อมต่อระหว่าง Visual Basic กับไมโครคอนโทรลเลอร์ dsPIC30F6010A นั้นจะใช้โมดูล UART เพื่อสื่อสารข้อมูลอนุกรมระหว่างไมโครคอนโทรลเลอร์ dsPIC30F6010A กับคอมพิวเตอร์ผ่านทางพอร์ตอนุกรม RS232



รูปที่ 4.11 แสดงการเชื่อมต่อกับไมโครคอนโทรลเลอร์ผ่านทาง Serial Port



รูปที่ 4.12 แสดงการส่งข้อมูลไปยังไมโครคอนโทรลเลอร์

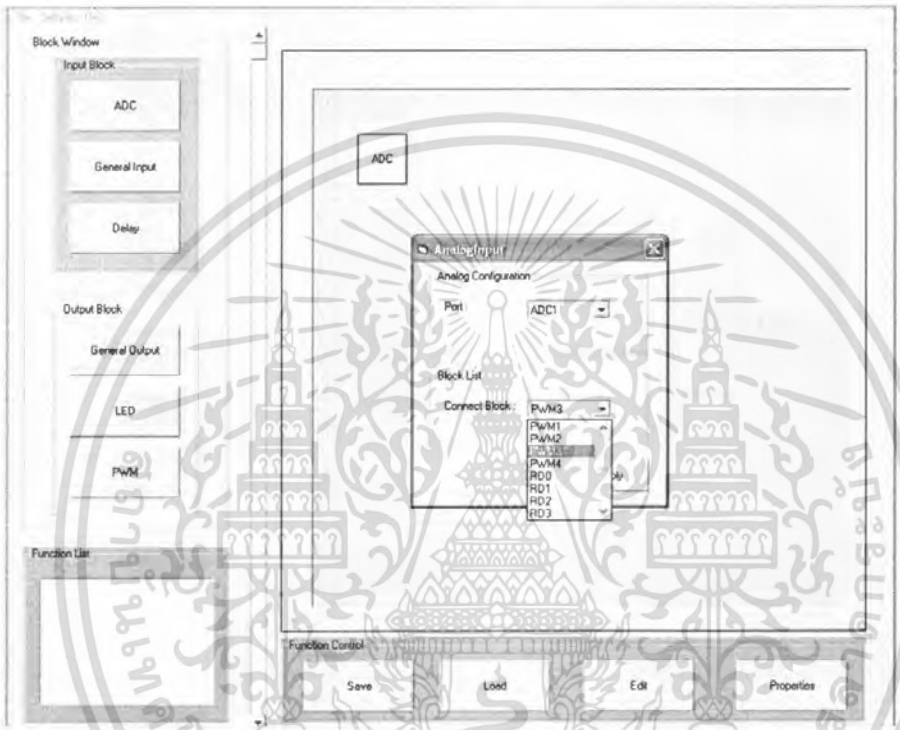
4.2.1.4 ไมโครคอนโทรลเลอร์

หลังจากที่มีการส่งข้อมูลผ่านพอร์ตอนุกรมแล้ว ไมโครคอนโทรลเลอร์จะทำหน้าที่ในการประมวลผลการทำงานตามที่ผู้ใช้งานได้ออกแบบไว้ใน User Interface จากนั้นจึงจะสามารถแสดงผลของการออกแบบได้

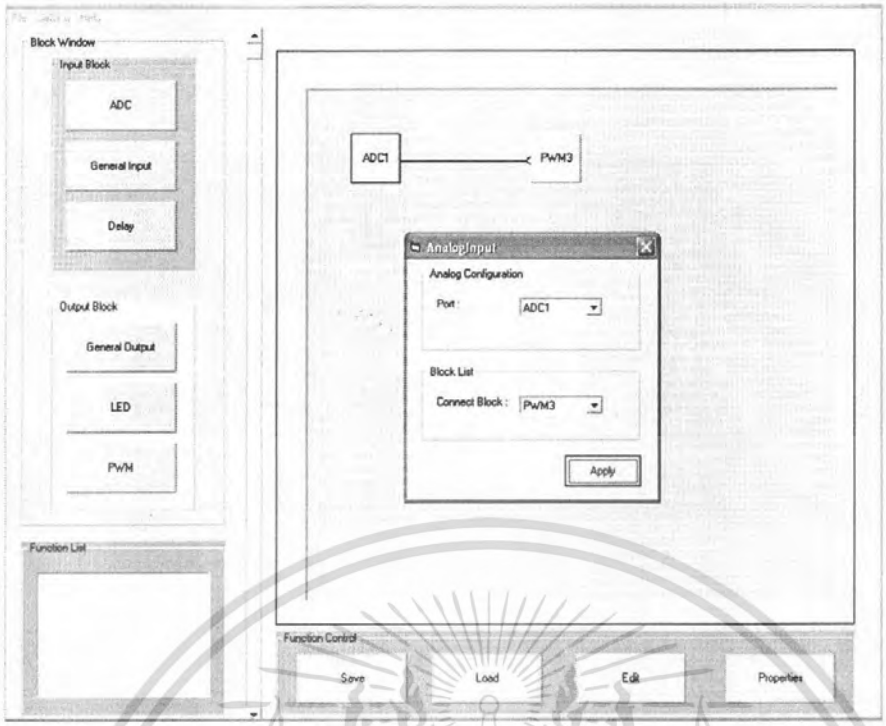
4.2.1.5 ตัวอย่างของการทำงานภาษาบล็อก

ตัวอย่างที่ 1 การควบคุมความสว่างของ LED โดยใช้บล็อก ADC และ PWM

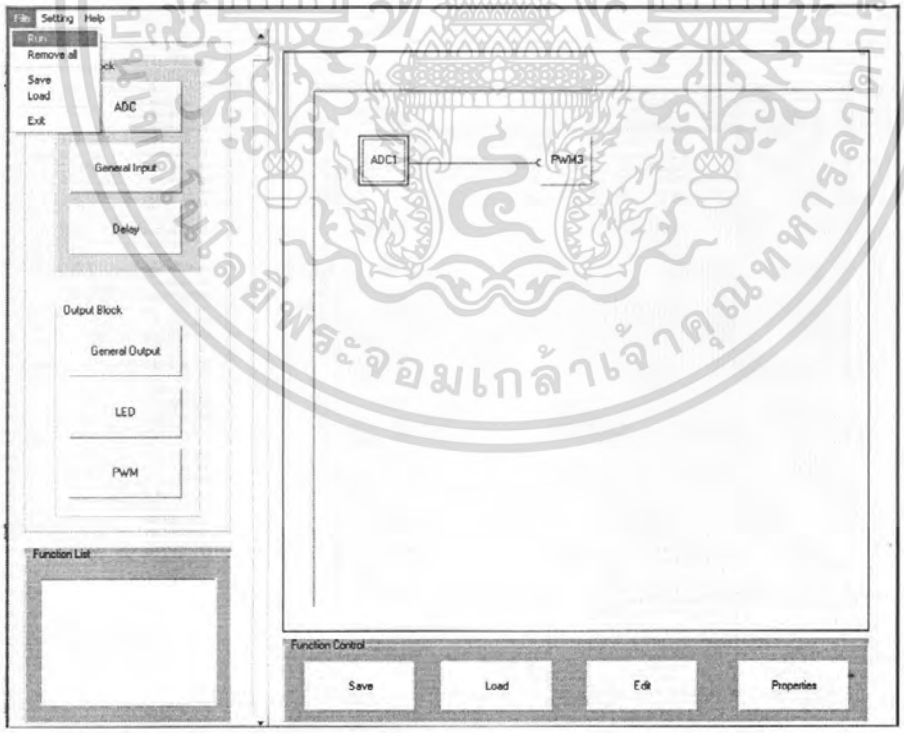
ในการควบคุมความสว่างของ LED จะต้องทำการป้อนค่าเพื่อปรับปริมาณความสว่างของหลอดไฟ โดยปรับความสว่างของ LED ตามการป้อนค่าของ ADC ซึ่งค่าความสว่างจะแปรผันตามค่าของแรงดัน เมื่อต้องการให้สว่างมากก็ปรับค่าแรงดันให้มากและลดค่าความสว่างได้ตามค่าของแรงดันตามลำดับ ซึ่งผลการทดลองจะแสดงผลออกมาได้ที่ LED และ ขาเอาท์พุท



รูปที่ 4.13 แสดงการเลือกพอร์ต PWM



รูปที่ 4.14 แสดงการเลือกพอร์ตที่ใช้และเอาท์พุตที่ต้องการให้แสดงผล

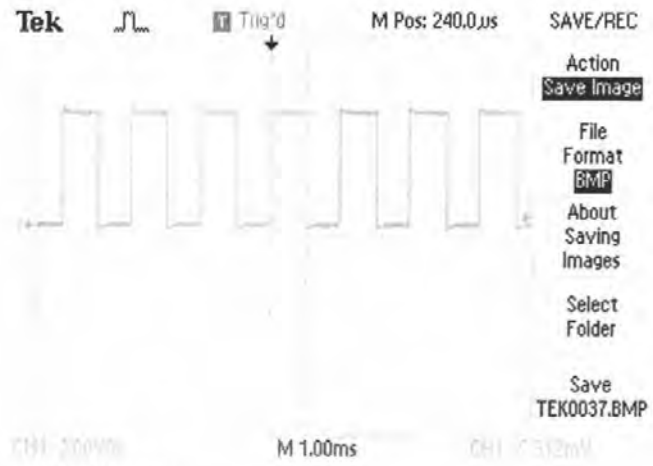


รูปที่ 4.15 แสดงการส่งข้อมูลไปยังไมโครคอนโทรลเลอร์

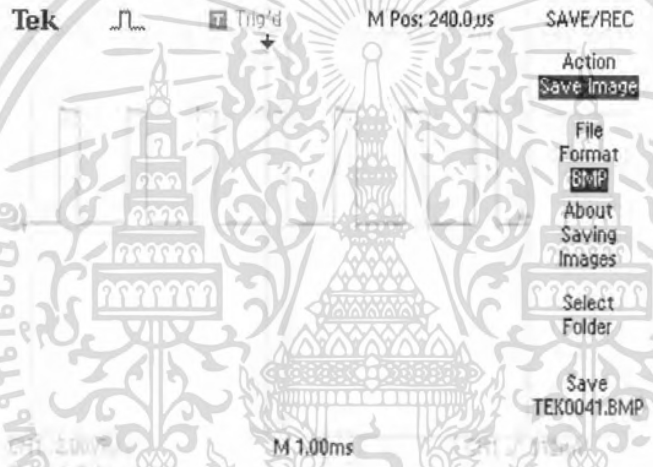
ผลการทดลองการควบคุมความสว่างของ LED โดยใช้บล็อกADCและPWMนั้น จะมีการนำสัญญาณที่ได้จากขาเอาท์พุทมาวัดสัญญาณผ่านออสซิสโลสโคป จากกราฟจะเห็นได้ว่า เมื่อเพิ่ม ค่า Value มากขึ้น ค่า Duty cycleจะมีค่ามากขึ้นตาม โดยดูได้จากการเปลี่ยนแปลงค่า Value กับ Duty cycle ดังกราฟต่อไปนี้

รูปที่ 4.16 แสดงค่า VALUE ที่ 0%

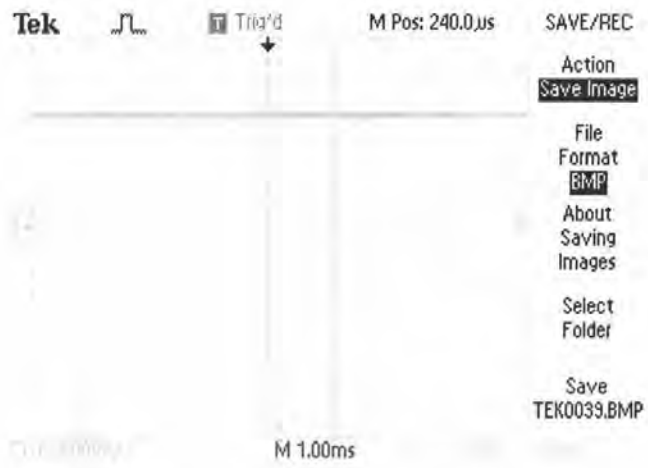
รูปที่ 4.17 แสดงค่า VALUE ที่ 25%



รูปที่ 4.18 แสดงค่า VALUE ที่ 50%



รูปที่ 4.19 แสดงค่า VALUE ที่ 75%

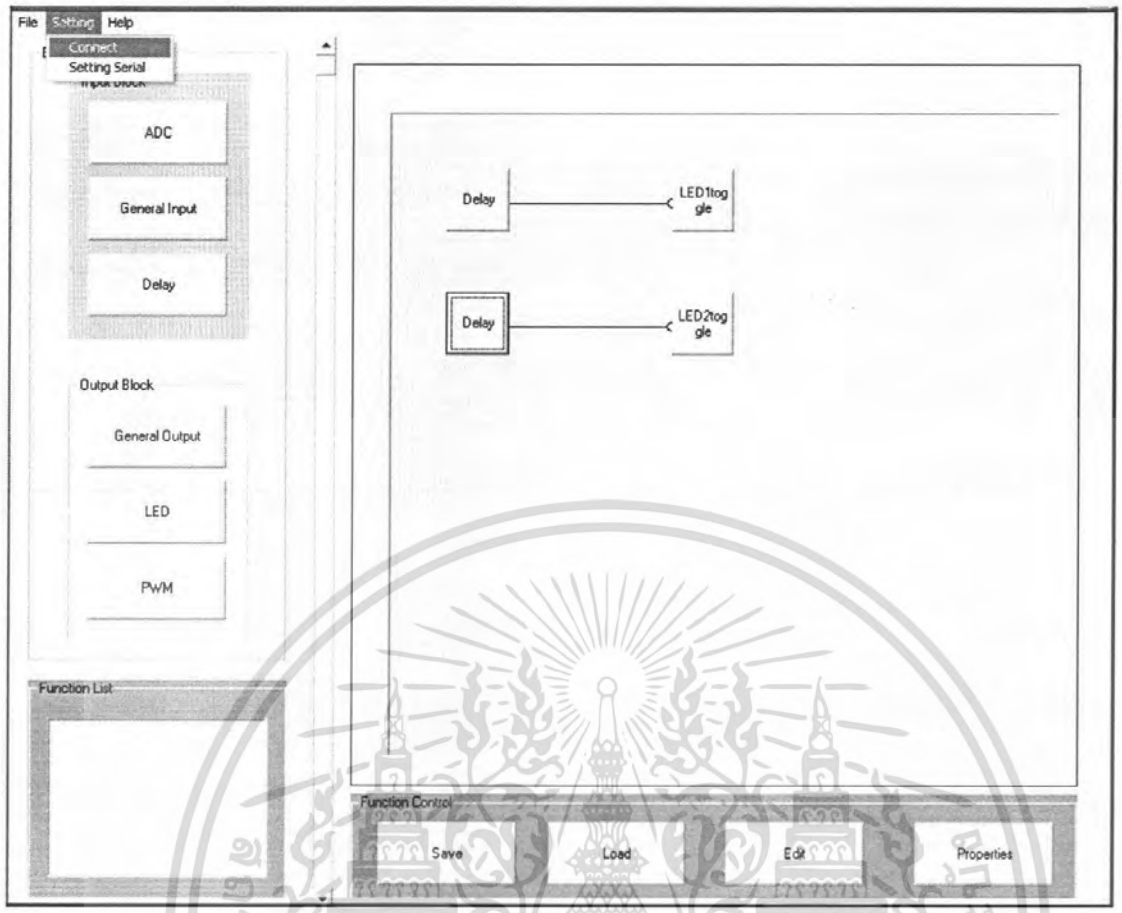


รูปที่ 4.20 แสดงค่า VALUE ที่ 100%

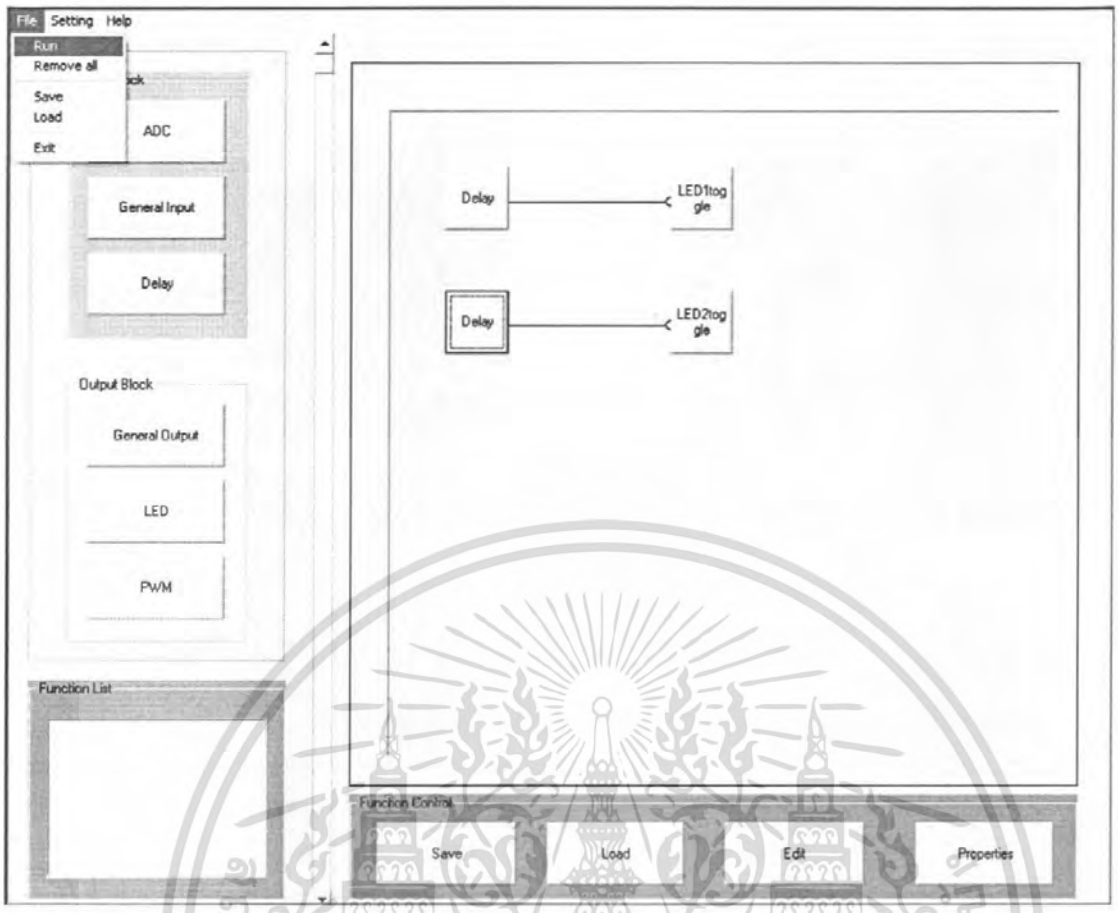
ตารางที่ 4.1 แสดงความสัมพันธ์ระหว่างค่า Value กับ Duty Cycle

Volt(V)	Value (%)	Duty Cycle (%)
0	0	0
1.125	25	28.5
2.25	50	51.4
3.375	75	80
4.5	100	100

ตัวอย่างที่ 2 ไฟกระพริบในการควบคุมการกระพริบของหลอด LED ต้องมีการเลือกบล็อก Delay เพื่อหน่วงเวลาให้กับ LED โดยต้องมีการกำหนด Delay ให้เท่ากับจำนวนหลอดไฟ LED ที่ต้องการในการทดลองนี้กำหนดให้ไฟกระพริบ 2 หลอด



รูปที่ 4.21 แสดงการเชื่อมต่อกับไมโครคอนโทรลเลอร์



รูปที่ 4.22 แสดงการส่งข้อมูลไปยังไมโครคอนโทรลเลอร์

บทที่ 5
บทสรุป ปัญหาและอุปสรรคในการทำงานและแนวทางการแก้ไข

5.1 บทสรุป

จากการศึกษาภาษาบล็อกจะเป็นการศึกษาเกี่ยวกับการติดต่อสื่อสารการทำงานระหว่างคอมพิวเตอร์และไมโครคอนโทรลเลอร์ dsPIC30F6010A โดยเมื่อเราทำการออกแบบระบบที่ต้องการใช้งานลงบน User Interface แล้วก็จะทำการส่งชุดคำสั่งบล็อกที่ถูกออกแบบออกไปยังไมโครคอนโทรลเลอร์ผ่านทางพอร์ตอนุกรม RS232 จากนั้นไมโครคอนโทรลเลอร์ก็จะทำการประมวลผลการทำงานของระบบที่ออกแบบไว้ ซึ่งผู้ใช้งานจะสามารถดูผลลัพธ์ของการทำงานได้จากการแสดงผลของบอร์ดทดลอง

แต่อย่างไรก็ตามโปรแกรมนี้เป็นโปรแกรมที่รองรับกลุ่มผู้ใช้งานที่อาจจะไม่มีความรู้ทางด้านการใช้งานไมโครคอนโทรลเลอร์เนื่องจากผู้ใช้งานไม่ต้องเขียนโปรแกรมเพื่อควบคุมการทำงานให้กับไมโครคอนโทรลเลอร์โดยตรง ซึ่งจากผลการศึกษาภาษาบล็อกก็จะเห็นได้ว่าเป็นโปรแกรมที่สามารถตอบสนองต่อการใช้งานได้ดีเป็นไปตามที่คณะผู้จัดทำได้วางแผนไว้

5.2 ปัญหาและอุปสรรคในการทำงาน

ในการทำโครงงานชิ้นนี้มีปัญหาและอุปสรรคหลายอย่างที่เกิดขึ้น คือ

1. เป็นโครงงานที่เกี่ยวข้องกับการเขียนโปรแกรมคอมพิวเตอร์ ซึ่งขาดความเข้าใจในการเขียนโปรแกรมของไมโครคอนโทรลเลอร์ dsPIC30F6010A จึงจำเป็นต้องใช้เวลาในการศึกษาทำให้การทำงานเป็นไปอย่างล่าช้า

2. เป็นโครงงานที่เกี่ยวข้องกับการเขียนโปรแกรมเกี่ยวกับ Visual Basic ซึ่งเป็นโปรแกรมที่ยังไม่เคยใช้มาก่อน ซึ่งขาดความเข้าใจในการเขียนโปรแกรม Visual Basic จึงจำเป็นต้องใช้เวลาในการศึกษา ทำให้การทำงานเป็นไปอย่างล่าช้า

3. อุปกรณ์ที่ใช้ในการทดลองมีปัญหาบ่อยครั้ง ซึ่งทำให้การทดลองที่ได้เกิดความผิดพลาดขึ้นได้และทำให้การทดลองล่าช้า

4. อุปกรณ์ในการโหลดโปรแกรม PICKIT ลงบนไมโครคอนโทรลเลอร์มีปัญหาบ่อย ต้องถอดและเสียบสายจนกระทั่งใช้งานได้

5. เนื่องจากการวางแผนงานที่ไม่ดี

5.3 แนวทางการแก้ไข

1.ศึกษาการเขียนโปรแกรมไมโครคอนโทรลเลอร์ dsPIC30F6010A จากหนังสือและทางอินเตอร์เน็ต และขอความช่วยเหลือจากผู้รู้ และผู้มีประสบการณ์และต้องลงมือปฏิบัติจึงจะทำให้เกิดความชำนาญมากขึ้น

2. ศึกษาการเขียนโปรแกรม Visual Basic จากหนังสือและทางอินเตอร์เน็ต และขอความช่วยเหลือจากผู้รู้และผู้มีประสบการณ์และต้องลงมือปฏิบัติจึงจะทำให้เกิดความชำนาญมากขึ้น

3. ทำการทดลองทุกครั้งต้องเช็คอุปกรณ์อยู่เสมอและต้องตรวจสอบผลที่ได้

4. ถอดสายที่ต่อกับคอมพิวเตอร์ออกมาแล้วเสียบสายใหม่หรือรีสตาร์ทคอมพิวเตอร์เพื่อให้ใช้งานได้

5. ควรมีการวางแผนการทำงานที่ดี จึงจะทำให้งานออกมาตรงตามกำหนด

บรรณานุกรม

[1] สัจจะ จรัสรุ่งรวีวร , "คู่มือการเขียนโปรแกรมและใช้งาน Visual Basic 6" , สำนักพิมพ์อินโฟเพรส , บริษัทดวงกมลสมัย จำกัด

[2] นคร ภักดีชาติ , ชัยวัฒน์ ลิ้มพรจิตรวิไล, " หนังสือคู่มือการทดลองเบื้องต้น dsPIC Microcontroller ด้วยโปรแกรมภาษา C กับ MPLAB C30 " , บริษัท อินโนเวทีฟ เอ็กซ์เพอริเมนต์ จำกัด

[3] ประภาพร ช่างไม้ , "คู่มือการเขียนโปรแกรมภาษา C ฉบับผู้เริ่มต้น" , บริษัท Infopress Developer Book

[4] อภิชาต ภู่พลับ , "เริ่มต้นเขียนโปรแกรมติดต่อและควบคุมฮาร์ดแวร์ด้วย Visual Basic" , บริษัท Infopress Developer Book

[5] พิชิต สันติกุลานนท์, ฉันทวุฒิ พืชผล , คู่มือเรียน VISUAL BASIC 6 , บริษัทโปรวิชัน จำกัด

[6] พร้อมเลิศ หล่อวิจิตร , "Advanced Visual Basic ฉบับ Object &Component" , บริษัทโปรวิชัน จำกัด

[7] ธีระศักดิ์ สุโชตินันท์ , ประยุทธ อินแบน, "โปรแกรมเมอร์มือใหม่ หัดเขียนโปรแกรม Microsoft Visual Basic6 Enterprise Edition", สำนักพิมพ์แห่งจุฬาลงกรณ์มหาวิทยาลัย

[8] ธนพล ฉันจรัสวิชัย , "ปฏิบัติการ Visual Basic สำหรับ Common Windows" , บริษัท ซีเอ็ดยูเคชั่น จำกัด (มหาชน)

[9] กิตติ ภักดีวัฒนะกุล , จำลอง ครูอุตสาหะ , "Visual Basic6 ฉบับโปรแกรมเมอร์" , บริษัทเคทีพีคอมพ์แอนด์คอนซัลท์ จำกัด

[10] www.kerhuel.eu

ภาคผนวก

ภาคผนวก ก

MICROCHIP

dsPIC30F6010A/6015

dsPIC30F6010A/6015 Enhanced Flash 16-bit Digital Signal Controller (DSC)

Note: This data sheet summarizes features of this group of dsPIC30F devices and is not intended to be a complete reference source. For more information on the CPU, peripherals, register descriptions and general device functionality, refer to the *dsPIC30F Family Reference Manual* (DS70046). For more information on the device instruction set and programming, refer to the *dsPIC30F Programmer's Reference Manual* (DS70030).

High-Performance Modified RISC CPU:

- Modified Harvard architecture
- C compiler optimized instruction set architecture with flexible Addressing modes
- 84 base instructions
- 24-bit wide instructions, 16-bit wide data path
- 144 Kbytes on-chip Flash program space (Instruction words)
- 8 Kbytes of on-chip data RAM
- 4 Kbytes of nonvolatile data EEPROM
- Up to 30 MIPS operation:
  - DC to 40 MHz external clock input
  - 4 MHz-10 MHz oscillator input with PLL active (4x, 8x, 16x)
  - 7.37 MHz internal RC with PLL active (4x, 8x, 16x)
- 44 interrupt sources
  - 5 external interrupt sources
  - 8 user selectable priority levels for each interrupt source
  - 4 processor trap sources
- 16 x 16-bit working register array

DSP Engine Features:

- Dual data fetch
- Accumulator write back for DSP operations
- Modulo and Bit-Reversed Addressing modes
- Two, 40-bit wide accumulators with optional saturation logic
- 17-bit x 17-bit single-cycle hardware fractional/integer multiplier
- All DSP instructions single cycle
- ±16-bit single-cycle shift

Peripheral Features:

- High-current sink/source I/O pins: 25 mA/25 mA
- Timer module with programmable prescaler:
  - Five 16-bit timers/counters; optionally pair 16-bit timers into 32-bit timer modules
- 16-bit Capture input functions
- 16-bit Compare/PWM output functions
- 3-wire SPI™ modules (supports 4 Frame modes)
- I^2C™ module supports Multi-Master/Slave mode and 7-bit/10-bit addressing
- 2 UART modules with FIFO Buffers
- 2 CAN modules, 2.0B compliant (dsPIC306010A)
- 1 CAN module, 2.0B compliant (dsPIC306015)

Motor Control PWM Module Features:

- 8 PWM output channels
  - Complementary or Independent Output modes
  - Edge and Center-Aligned modes
- 4 duty cycle generators
- Dedicated time base
- Programmable output polarity
- Dead-Time control for Complementary mode
- Manual output control
- Trigger for A/D conversions

Quadrature Encoder Interface Module Features:

- Phase A, Phase B and Index Pulse input
- 16-bit up/down position counter
- Count direction status
- Position Measurement (x2 and x4) mode
- Programmable digital noise filters on inputs
- Alternate 16-bit Timer/Counter mode
- Interrupt on position counter rollover/underflow

dsPIC30F6010A/6015

Analog Features:

- 10-bit Analog-to-Digital Converter (A/D) with 4 S/H Inputs:
  - 1 Msps conversion rate
  - 16 input channels
  - Conversion available during Sleep and Idle
- Programmable Brown-out Detection and Reset generation

Special Microcontroller Features:

- Enhanced Flash program memory:
  - 10,000 erase/write cycle (min.) for industrial temperature range, 100K (typical)
- Data EEPROM memory:
  - 100,000 erase/write cycle (min.) for industrial temperature range, 1M (typical)
- Self-reprogrammable under software control
- Power-on Reset (POR), Power-up Timer (PWRT) and Oscillator Start-up Timer (OST)
- Flexible Watchdog Timer (WDT) with on-chip, low-power RC oscillator for reliable operation
- Fail-Safe Clock Monitor operation detects clock failure and switches to on-chip, low-power RC oscillator
- Programmable code protection
- In-Circuit Serial Programming™ (ICSP™)
- Selectable Power Management modes
  - Sleep, Idle and Alternate Clock modes

CMOS Technology:

- Low-power, high-speed Flash technology
- Wide operating voltage range (2.5V to 5.5V)
- Industrial and Extended temperature ranges
- Low-power consumption

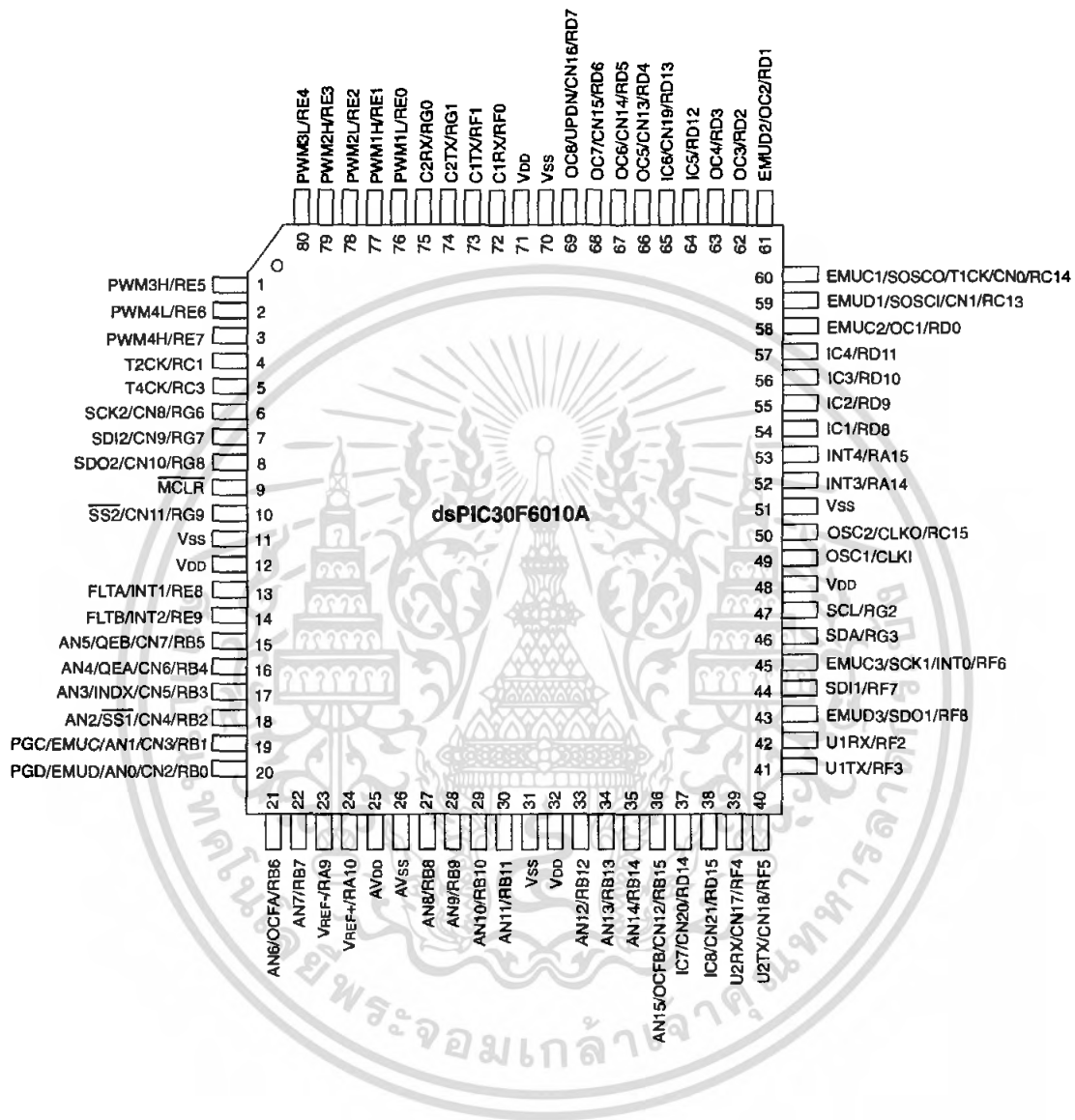
dsPIC30F Motor Control and Power Conversion Family*

Device	Pins	Program Mem. Bytes/ Instructions	SRAM Bytes	EEPROM Bytes	Timer 16-bit	Input Cap	Output Comp/Std PWM	Motor Control PWM	A/D 10-bit 1 Msps	Quad Enc	UART	SPI™	I²C™	CAN
dsPIC30F2010	28	12K/4K	512	1024	3	4	2	6 ch	6 ch	Yes	1	1	1	-
dsPIC30F3010	28	24K/8K	1024	1024	5	4	2	6 ch	6 ch	Yes	1	1	1	-
dsPIC30F4012	28	48K/16K	2048	1024	5	4	2	6 ch	6 ch	Yes	1	1	1	1
dsPIC30F3011	40/ 44	24K/6K	1024	1024	5	4	4	6 ch	9 ch	Yes	2	1	1	-
dsPIC30F4011	40/ 44	48K/16K	2048	1024	5	4	4	6 ch	9 ch	Yes	2	1	1	1
dsPIC30F5015	64	66K/22K	2048	1024	5	4	4	8 ch	16 ch	Yes	1	2	1	1
dsPIC30F5016	80	66K/22K	2048	1024	5	4	4	8 ch	16 ch	Yes	1	2	1	1
dsPIC30F6010A	80	144K/48K	8192	4096	5	8	8	8 ch	16 ch	Yes	2	2	1	2
dsPIC30F6015	64	144K/48K	8192	4096	5	8	8	8 ch	16 ch	Yes	2	2	1	1

* This table provides a summary of the dsPIC30F6010A/6015 peripheral features. Other available devices in the dsPIC30F Motor Control and Power Conversion Family are shown for feature comparison.

Pin Diagram

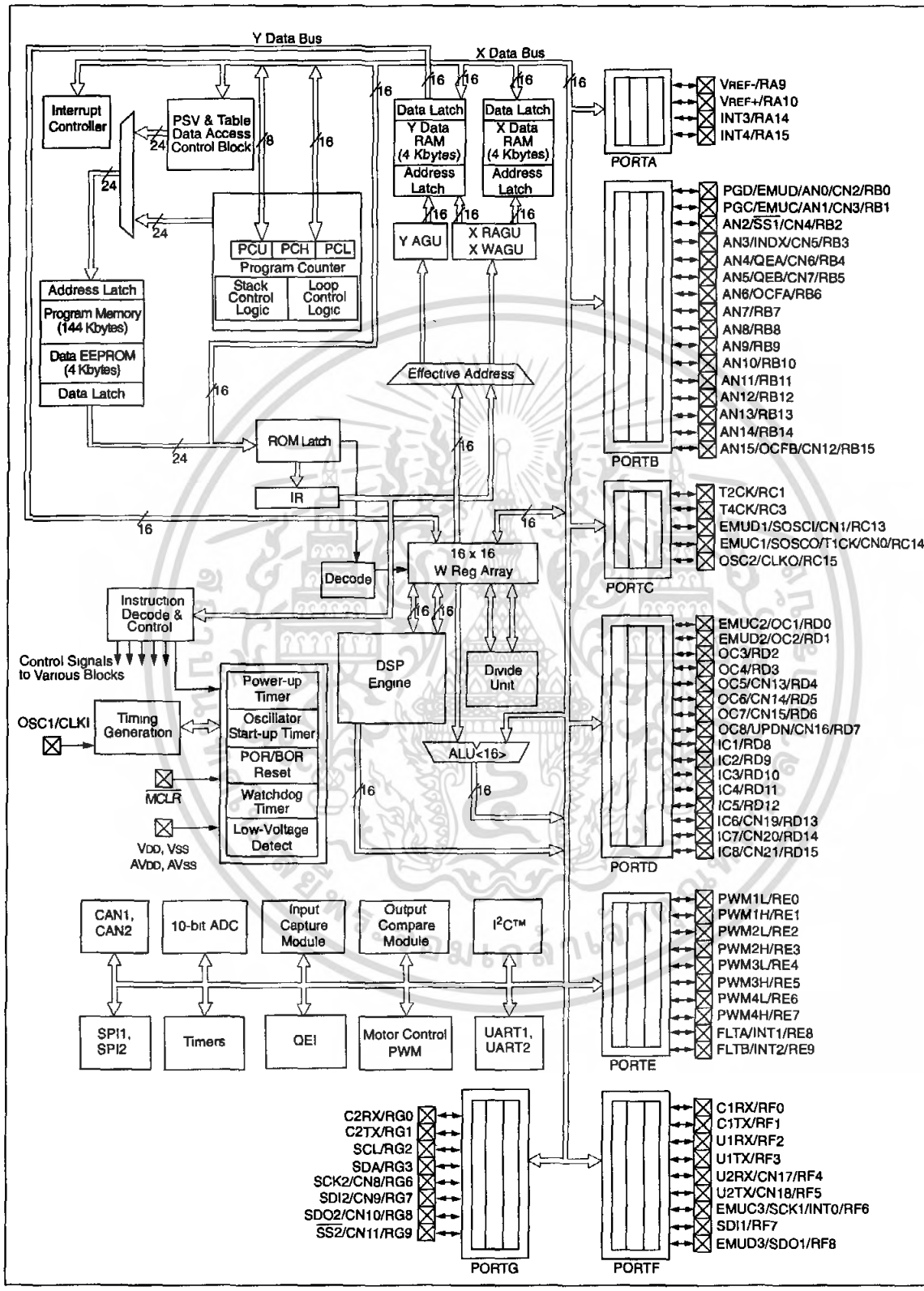
80-Pin TQFP



Note: Pinout subject to change.

dsPIC30F6010A/6015

FIGURE 1-1: dsPIC30F6010A BLOCK DIAGRAM



dsPIC30F6010A/6015

Table 1-1 provides a brief description of the device I/O pinout and the functions that are multiplexed to a port pin. Multiple functions may exist on one port pin. When multiplexing occurs, the peripheral module's functional requirements may force an override of the data direction of the port pin.

TABLE 1-1: dsPIC30F6010A/6015 I/O PIN DESCRIPTIONS

Pin Name	Pin Type	Buffer Type	Description
AN0-AN15	I	Analog	Analog input channels. AN0 and AN1 are also used for device programming data and clock inputs, respectively.
AVDD	P	P	Positive supply for analog module.
AVSS	P	P	Ground reference for analog module.
CLKI CLKO	I O	ST/CMOS —	External clock source input. Always associated with OSC1 pin function. Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode. Optionally functions as CLKO in RC and EC modes. Always associated with OSC2 pin function.
CN0-CN23	I	ST	Input change notification inputs. Can be software programmed for internal weak pull-ups on all inputs.
C1RX C1TX C2RX C2TX	I O I O	ST — ST —	CAN1 bus receive pin CAN1 bus transmit pin. CAN2 bus receive pin. CAN2 bus transmit pin.
EMUD EMUC EMUD1 EMUC1 EMUD2 EMUC2 EMUD3 EMUC3	I/O I/O I/O I/O I/O I/O I/O I/O	ST ST ST ST ST ST ST ST	ICD Primary Communication Channel data input/output pin. ICD Primary Communication Channel clock input/output pin. ICD Secondary Communication Channel data input/output pin. ICD Secondary Communication Channel clock input/output pin. ICD Tertiary Communication Channel data input/output pin. ICD Tertiary Communication Channel clock input/output pin. ICD Quaternary Communication Channel data input/output pin. ICD Quaternary Communication Channel clock input/output pin.
IC1-IC8	I	ST	Capture inputs 1 through 8.
INDX QEA QEB UPDN	I I I O	ST ST ST CMOS	Quadrature Encoder Index Pulse input. Quadrature Encoder Phase A input in QEI mode. Auxiliary Timer External Clock/Gate input in Timer mode. Quadrature Encoder Phase A input in QEI mode. Auxiliary Timer External Clock/Gate input in Timer mode. Position Up/Down Counter Direction State.
INT0 INT1 INT2 INT3 INT4	I I I I I	ST ST ST ST ST	External interrupt 0. External interrupt 1. External interrupt 2. External interrupt 3. External interrupt 4.

Legend: CMOS = CMOS compatible input or output; Analog = Analog input; ST = Schmitt Trigger input with CMOS levels; O = Output; I = Input; P = Power

TABLE 1-1: dsPIC30F6010A/6015 I/O PIN DESCRIPTIONS (CONTINUED)

Pin Name	Pin Type	Buffer Type	Description
FLTA	I	ST	PWM Fault A input.
FLTB	I	ST	PWM Fault B input.
PWM1L	O	—	PWM 1 Low output.
PWM1H	O	—	PWM 1 High output.
PWM2L	O	—	PWM 2 Low output.
PWM2H	O	—	PWM 2 High output.
PWM3L	O	—	PWM 3 Low output.
PWM3H	O	—	PWM 3 High output.
PWM4L	O	—	PWM 4 Low output.
PWM4H	O	—	PWM 4 High output.
$\overline{MCLR}$	I/P	ST	Master Clear (Reset) input or programming voltage input. This pin is an active-low Reset to the device.
OCFA	I	ST	Compare Fault A input (for Compare channels 1, 2, 3 and 4).
OCFB	I	ST	Compare Fault B input (for Compare channels 5, 6, 7 and 8).
OC1-OC8	O	—	Compare outputs 1 through 8.
OSC1	I	ST/CMOS	Oscillator crystal input. ST buffer when configured in RC mode; CMOS otherwise.
OSC2	I/O	—	Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode. Optionally functions as CLKO in RC and EC modes.
PGD	I/O	ST	In-Circuit Serial Programming™ data input/output pin.
PGC	I	ST	In-Circuit Serial Programming clock input pin.
RA9-RA10	I/O	ST	PORTA is a bidirectional I/O port.
RA14-RA15	I/O	ST	
RB0-RB15	I/O	ST	PORTB is a bidirectional I/O port.
RC1	I/O	ST	PORTC is a bidirectional I/O port.
RC3	I/O	ST	
RC13-RC15	I/O	ST	
RD0-RD15	I/O	ST	PORTD is a bidirectional I/O port.
RE0-RE9	I/O	ST	PORTE is a bidirectional I/O port.
RF0-RF8	I/O	ST	PORTF is a bidirectional I/O port.
RG0-RG3	I/O	ST	PORTG is a bidirectional I/O port.
RG6-RG9	I/O	ST	
SCK1	I/O	ST	Synchronous serial clock input/output for SPI™ #1.
SDI1	I	ST	SPI #1 Data In.
SDO1	O	—	SPI #1 Data Out.
SS1	I	ST	SPI #1 Slave Synchronization.
SCK2	I/O	ST	Synchronous serial clock input/output for SPI #2.
SDI2	I	ST	SPI #2 Data In.
SDO2	O	—	SPI #2 Data Out.
SS2	I	ST	SPI #2 Slave Synchronization.
SCL	I/O	ST	Synchronous serial clock input/output for I^2C™.
SDA	I/O	ST	Synchronous serial data input/output for I^2C.
SOSCO	O	—	32 kHz low-power oscillator crystal output.
SOSCI	I	ST/CMOS	32 kHz low-power oscillator crystal input. ST buffer when configured in RC mode; CMOS otherwise.
T1CK	I	ST	Timer1 external clock input.
T2CK	I	ST	Timer2 external clock input.
T4CK	I	ST	Timer4 external clock input.

Legend: CMOS = CMOS compatible input or output; Analog = Analog input; ST = Schmitt Trigger input with CMOS levels; O = Output; I = Input; P = Power

TABLE 1-1: dsPIC30F6010A/6015 I/O PIN DESCRIPTIONS (CONTINUED)

Pin Name	Pin Type	Buffer Type	Description
U1RX	I	ST	UART1 Receive.
U1TX	O	—	UART1 Transmit.
U1ARX	I	ST	UART1 Alternate Receive.
U1ATX	O	—	UART1 Alternate Transmit.
U2RX	I	ST	UART2 Receive.
U2TX	O	—	UART2 Transmit.
VDD	P	—	Positive supply for logic and I/O pins.
VSS	P	—	Ground reference for logic and I/O pins.
VREF+	I	Analog	Analog Voltage Reference (High) input.
VREF-	I	Analog	Analog Voltage Reference (Low) input.

Legend: CMOS = CMOS compatible input or output Analog = Analog input
ST = Schmitt Trigger input with CMOS levels O = Output
I = Input P = Power

ภาคผนวก ข

```
//-------------------------------------------------------------------------------------------------------------//
// Program              : Block Language
// Description          : Generate PWM signal by Output Comparator at pin RD0(OC1)
// Frequency            : 9.6 MHz at PLL 4x
// Filename             : oc_pwm.c
// C compiler           : C30 Compiler by Microchip Technology
//-------------------------------------------------------------------------------------------------------------//
#include<p30f6010a.h>
#include<ports.h>                       //  Module function for Interrupt configuration port
#include<outcompare.h>                  //  Module function for  output comparator
#include<adc10.h>                       // Module function for 10 bit ADC
#include<uart.h>                        // Module function for uart
#include "Delay.h"
#include "GenericTypeDefs.h"

_FOSC(FRC_PLL4)                                 // internal Osc 7.37MHz + 4X PLL
_FWDT(WDT_OFF)                                  // Disable Watchdog timer
_FBORPOR(MCLR_EN & PBOR_OFF & PWRT_OFF)

unsigned int duty;                              // Keep dutycycle for PWM signal
unsigned char ANString[8];
char cmd[8*128];
int temp_val[10];
int cmd_count = 0;
char *cmd_ptr;
int run  = 0;

void adc_init()
{
            ADCHS                       // Input to AN0 (potentiometer)
            ADPCFGbits.PCFG0 = 0;   // Disable digital input on AN5 (potentiometer)
            ADPCFGbits.PCFG1 = 0;   // Disable digital input on AN4 (TC1047A temp sensor)
            ADPCFGbits.PCFG2 = 0;   // Disable digital input on AN5 (potentiometer)
            ADPCFGbits.PCFG3 = 0;   // Disable digital input on AN4 (TC1047A temp sensor)


     ADCON1 = 0x00EC;           // SIMSAM bit = 1 implies ...
     ADCHS = 0x0007;             // Connect AN7 as CH0 input
     ADCSSL = 0;
     ADCON3 = 0x0302;           // Auto Sampling 3 Tad, Tad = internal 2 Tcy
     ADCON2 = 0x030C;           // CHPS = 1x implies simultaneous ...
     ADCON1bits.ADON = 1;    // turn ADC ON
}

void outputcompare_init()
{
        unsigned int pulse_start ;
        CloseOC1();
   CloseOC2();
   CloseOC3();
   CloseOC4();

        pulse_start = 0;
        duty = 0;


        OpenOC1(OC_IDLE_CON &
```

```
        OC_TIMER3_SRC &
        OC_CONTINUE_PULSE,
        duty, pulse_start);

        OpenOC2(OC_IDLE_CON &
        OC_TIMER3_SRC &
        OC_CONTINUE_PULSE,
        duty, pulse_start);

        OpenOC3(OC_IDLE_CON &
        OC_TIMER3_SRC &
        OC_PWM_FAULT_PIN_DISABLE,
        duty, pulse_start);

        OpenOC4(OC_IDLE_CON &
        OC_TIMER3_SRC &
        OC_PWM_FAULT_PIN_DISABLE,
        duty, pulse_start);

        PR3 = 1023*10 ;
        T3CON = 0x8000;
}

void uart1_init()
{
        unsigned int baudvalue;          // Keep baud rate value for Load into U1BRG
        unsigned int U1MODEvalue;        // Keep value for Load into U1MODE
        unsigned int U1STAvalue;         // Keep value for Load into U1STA

        CloseUART1();                    // Disable UART1

        baudvalue = 47;                  // Baud rate 9600 bps

        U1MODEvalue = UART_EN &                  // Enable UART1
                      UART_IDLE_CON &            // UART1 working in idle mode
                      UART_DIS_WAKE &            // Disable wake-up on start UART
                      UART_DIS_LOOPBACK &        // Disable loop back mode
                      UART_DIS_ABAUD &           // Disable autobaud mode
                      UART_NO_PAR_8BIT &         // Set data 8 bit ,no parity bit
                      UART_1STOPBIT;             // Set 1 stop bit

        U1STAvalue    = UART_TX_PIN_NORMAL &     // TX Break bit normal
                        UART_TX_ENABLE &         // Enable TX for UART
                        UART_ADR_DETECT_DIS &    // Disable detect address mode
                        UART_RX_OVERRUN_CLEAR    // Reset buffer over run

   OpenUART1(U1MODEvalue, U1STAvalue, baudvalue); // Execute configuration for UART1
}

void uitoa(unsigned short Value, unsigned char *Buffer)
{
        unsigned char i;
        unsigned short Digit;
        unsigned short Divisor;
        unsigned char Printed = 0;

        if(Value)
        {
                for(i = 0, Divisor = 10000; i < 5; i++)
                {
                        Digit = Value/Divisor;
```

```
                        if(Digit || Printed)
                        {
                                *Buffer++ = '0' + Digit;
                                Value -= Digit*Divisor;
                                Printed = 1;
                        }
                        Divisor /= 10;
                }
        }
        else
        {
                *Buffer++ = '0';
        }

        *Buffer = '\0';
}

void Process_loader(void)
{
        unsigned char  dat;             // Buffer for receive character
  int len;
  int i;
    if(DataRdyUART1())
                {

      dat = ReadUART1();            // Load data into buffer variable
      if(dat == 0xAA)
                        {
        while(!DataRdyUART1());
        dat = ReadUART1();          // len hi byte

        while(!DataRdyUART1());
        len = ((dat<<8)&0xFF00)|(ReadUART1()&0x00FF);       // len low byte
        run = 0;
        for(i=0;i<len;i++){
              while(!DataRdyUART1());
          cmd[cmd_count++] = ReadUART1();
        }
        cmd_count = 0;
        run = 1;
      }

    }

}

void Process_main(void)
{
  SHORT_VAL delay;
  unsigned int pwm,old_pwm[4],adc_result[4],i;
    putcUART1(cmd_ptr[0]);
   while(BusyUART1());
  switch(cmd_ptr[0]){
    case 0x00 :                 //output or delay
      if(cmd_ptr[1]==0)
    {
    switch(cmd_ptr[2])
          {
           case 0: CloseOC1(); TRISDbits.TRISD0 = 0; LATDbits.LATD0 = cmd_ptr[3]; break;
           case 1: CloseOC2(); TRISDbits.TRISD1 = 0; LATDbits.LATD1 = cmd_ptr[3]; break;
           case 2: CloseOC3(); TRISDbits.TRISD2 = 0; LATDbits.LATD2 = cmd_ptr[3]; break;
```

```
            case 3: CloseOC4(); TRISDbits.TRISD3 = 0; LATDbits.LATD3 = cmd_ptr[3]; break;
            case 4: CloseOC5(); TRISDbits.TRISD4 = 0; LATDbits.LATD4 = cmd_ptr[3]; break;
            case 5: CloseOC6(); TRISDbits.TRISD5 = 0; LATDbits.LATD5 = cmd_ptr[3]; break;
           case 6: CloseOC7(); TRISDbits.TRISD6 = 0; LATDbits.LATD6 = cmd_ptr[3]; break;
           case 7: CloseOC8(); TRISDbits.TRISD7 = 0; LATDbits.LATD7 = cmd_ptr[3]; break;
          }
}else{
     switch(cmd_ptr[2])
           {
case 0: CloseOC1(); TRISDbits.TRISD0 = 0; LATDbits.LATD0 = temp_val[cmd_ptr[3]]; break;
case 1: CloseOC2(); TRISDbits.TRISD1 = 0; LATDbits.LATD1 = temp_val[cmd_ptr[3]]; break;
case 2: CloseOC3(); TRISDbits.TRISD2 = 0; LATDbits.LATD2 = temp_val[cmd_ptr[3]]; break;
case 3: CloseOC4(); TRISDbits.TRISD3 = 0; LATDbits.LATD3 = temp_val[cmd_ptr[3]]; break;
case 4: CloseOC5(); TRISDbits.TRISD4 = 0; LATDbits.LATD4 = temp_val[cmd_ptr[3]]; break;
case 5: CloseOC6(); TRISDbits.TRISD5 = 0; LATDbits.LATD5 = temp_val[cmd_ptr[3]]; break;
case 6: CloseOC7(); TRISDbits.TRISD6 = 0; LATDbits.LATD6 = temp_val[cmd_ptr[3]]; break;
case 7: CloseOC8(); TRISDbits.TRISD7 = 0; LATDbits.LATD7 = temp_val[cmd_ptr[3]]; break;
           }
       }
     cmd_ptr += 8;      // point to next command;
break;

     case 0x01 :        //delay

          delay.v[0] =  cmd_ptr[2];
          delay.v[1] =  cmd_ptr[1];
          DelayMs(delay.Val);
          cmd_ptr += 8;              // point to next command;
     break;

     case 0x02 :
     cmd_ptr = &cmd[0];     // go to first command
     break;

     case 0x03 :              //input

         switch(cmd_ptr[2])
         {
      case 0: TRISDbits.TRISD8 = 1;  temp_val[cmd_ptr[3]] = PORTDbits.RD8;   break;
      case 1: TRISDbits.TRISD9 = 1;  temp_val[cmd_ptr[3]] = PORTDbits.RD9;   break;
      case 2: TRISDbits.TRISD10 = 1; temp_val[cmd_ptr[3]] = PORTDbits.RD10; break;
      case 3: TRISDbits.TRISD11 = 1; temp_val[cmd_ptr[3]] = PORTDbits.RD11; break;
      case 4: TRISDbits.TRISD12 = 1; temp_val[cmd_ptr[3]] = PORTDbits.RD12; break;
      case 5: TRISDbits.TRISD13 = 1; temp_val[cmd_ptr[3]] = PORTDbits.RD13; break;
      case 6: TRISDbits.TRISD14 = 1; temp_val[cmd_ptr[3]] = PORTDbits.RD14; break;
      case 7: TRISDbits.TRISD15 = 1; temp_val[cmd_ptr[3]] = PORTDbits.RD15; break;
           }

     cmd_ptr += 8;              // point to next command;
     break;

     case 0x04 :              //PWM
        if(cmd_ptr[1]==0)
        {
         pwm = (1023)*(cmd_ptr[3]/10);
         PR3 = 1023*10 ;
         T3CON = 0x8000;
         switch(cmd_ptr[2])
           {
            case 0: TRISDbits.TRISD0 = 0;
               OpenOC1(OC_IDLE_CON &
```

```
                OC_TIMER3_SRC &
                OC_PWM_FAULT_PIN_DISABLE,
                0, 0);
                  SetDCOC1PWM(pwm);
                  break;
              case 1: TRISDbits.TRISD1 = 0;
                        OpenOC2(OC_IDLE_CON &
                        OC_TIMER3_SRC &
                        OC_PWM_FAULT_PIN_DISABLE,
                        0, 0);
                        SetDCOC2PWM(pwm);
                        break;

              case 2: TRISDbits.TRISD2 = 0;
                        OpenOC3(OC_IDLE_CON &
                        OC_TIMER3_SRC &
                        OC_PWM_FAULT_PIN_DISABLE,
                        0, 0);
                        SetDCOC3PWM(pwm);
                        break;

               case 3: TRISDbits.TRISD3 = 0;
                        OpenOC4(OC_IDLE_CON &
                        OC_TIMER3_SRC &
                        OC_PWM_FAULT_PIN_DISABLE,
                        0, 0);
                        SetDCOC4PWM(pwm);
                        break;
          }
                cmd_ptr[2] = 0xFF;
}else if(cmd_ptr[1]==1)
        {
        switch(cmd_ptr[2])
            {
                case 0: OpenOC1(OC_IDLE_CON &
                        OC_TIMER3_SRC &
                        OC_PWM_FAULT_PIN_DISABLE,
                        0, 0);
                        if(old_pwm[0] != temp_val[cmd_ptr[3]] ){
                        SetDCOC1PWM(temp_val[cmd_ptr[3]]*10);
                        }
                        break;

                    case 1: OpenOC2(OC_IDLE_CON &
                        OC_TIMER3_SRC &
                        OC_PWM_FAULT_PIN_DISABLE,
                        0, 0);
                        if(old_pwm[1] != temp_val[cmd_ptr[3]] ){
                        SetDCOC2PWM(temp_val[cmd_ptr[3]]*10);
                        }
                        break;

                    case 2: OpenOC3(OC_IDLE_CON &
                        OC_TIMER3_SRC &
                        OC_PWM_FAULT_PIN_DISABLE,
                        0, 0);
                        SetDCOC3PWM(pwm);
                        if(old_pwm[2] != temp_val[cmd_ptr[3]] ){
                        SetDCOC3PWM(temp_val[cmd_ptr[3]]*10);
                        }
                        break;
```

```
                case 3: OpenOC4(OC_IDLE_CON &
                        OC_TIMER3_SRC &
                        OC_PWM_FAULT_PIN_DISABLE,
                        0, 0);
                        SetDCOC4PWM(pwm);
                        if(old_pwm[3] != temp_val[cmd_ptr[3]] ){
                        SetDCOC4PWM(temp_val[cmd_ptr[3]]*10);
                        }
                        break;
                }

                for(i=0;i<4;i++)
                {
                  old_pwm[i] = temp_val[cmd_ptr[3]];          // Save ADC value
                }

        }
                cmd_ptr += 8;
                break;

    case 0x05 :
                while(BusyADC10());         // Ensure for Sampling success
                ADCON1bits.DONE = 0;
                for(i=0;i<4;i++)                // Loop 4 time for read ADC keep to result array
                    {
                      adc_result[i] = ReadADC10(i);          // Keep value for ADC value
                    }
                temp_val[cmd_ptr[2]] = adc_result[cmd_ptr[1]];
                cmd_ptr += 8;
                break;

    case 0x06:

          break;
  default :
  break;

  }
}

//-------------------------------------------------------------------------------------------//
//---------------------------------- Main Program -------------------------------------------//
//-------------------------------------------------------------------------------------------//
int main(void)
{
   unsigned int result[4], old_result[4],i;          // Keep data
    char Txdata[] = "\r\nUart test input key...\r\n";   // Table character
outputcompare_init();
 adc_init();                                          // Initial ADC
uart1_init();                   // Initial UART1
cmd_ptr = &cmd[0];
        while(1)                // Infinite loop
        {
        Process_loader();
          if(run)
          {
            Process_main();
          }
```

```
        }
        return 0;
}

//-----------------------------------------------------------------------------------------------------//
// Program              : Block Language
// Description          : General Delay rouines
// Frequency            : 9.6 MHz at PLL 4x
// Filename             : Delay.c
// C compiler           : C30 Compiler by Microchip Technology
//-----------------------------------------------------------------------------------------------------//

#define __DELAY_C

#include "delay.h"
#if !defined(__18CXX) || defined(HI_TECH_C)
void DelayMs(unsigned short ms)
{
  unsigned char i;
  while(ms--)
  {
    i=4;
    while(i--)
    {
      Delay10us(25);
    }
  }
}
#endif  //#if !defined(__18CXX) || defined(HI_TECH_C)
#if defined(__C30__) || defined(__C32__)
void Delay10us(unsigned int dwCount)
{
        volatile unsigned int _dcnt;

        _dcnt = dwCount*((unsigned int)(0.00001/(1.0/INSTR_FREQ)/10));
        while(_dcnt--)
        {
                #if defined(__C32__)
                        Nop();
                        Nop();
                        Nop();
                #endif
        }
}
#endif
```

```
//-----------------------------------------------------------------------------------------------------------------//
// Program              : Block Language
// Description          : General Delay rouines
// Frequency            : 9.6 MHz at PLL 4x
// Filename             : Delay.h
// C compiler           : C30 Compiler by Microchip Technology
//-----------------------------------------------------------------------------------------------------------------//

#ifndef __DELAY_H
#define __DELAY_H

#define CLOCK_FREQ                  (29480000ul)     // Hz
#define INSTR_FREQ                  ((CLOCK_FREQ+2ul)/4ul)

#if defined(__18CXX) && !defined(HI_TECH_C)
        #include <delays.h>
#endif

#if !defined(INSTR_FREQ)
        #error INSTR_FREQ must be defined.
#endif

#if defined(__18CXX) && !defined(HI_TECH_C)
        #define Delay10us(us)           Delay10TCYx(((INSTR_FREQ/1000000)*us))
        #define DelayMs(ms)
        Delay1KTCYx((((INSTR_FREQ/1000)*ms)/1000))

#elif defined(__C30__) || defined(__C32__)
        void Delay10us(unsigned int dwCount);
        void DelayMs(unsigned short ms);

#else
        #define Delay10us(x)                    \
        {                                                       \
                unsigned long _dcnt;            \
                _dcnt=x*((unsigned long)(0.00001/(1.0/INSTR_FREQ)/6));          \
                while(_dcnt--);                         \
        }
        void DelayMs(WORD ms);
#endif


#endif
```

```
//-----------------------------------------------------------------------------//
// Program              : Block Language
// Description          : Generic Type Definitions
// Frequency            : 9.6 MHz at PLL 4x
// Filename             : GenericTypeDefs.h
// C compiler           : C30 Compiler by Microchip Technology
//-----------------------------------------------------------------------------//
#ifndef __GENERIC_TYPE_DEFS_H_
#define __GENERIC_TYPE_DEFS_H_
typedef enum _BOOL { FALSE = 0, TRUE } BOOL;        // Undefined size
typedef unsigned char           BYTE;                   // 8-bit unsigned
typedef unsigned short int      WORD;                   // 16-bit unsigned
typedef unsigned long           DWORD;                      // 32-bit unsigned
typedef unsigned long long      QWORD;                      // 64-bit unsigned
typedef signed char                 CHAR;                   // 8-bit signed
typedef signed short int  SHORT;                        // 16-bit signed
typedef signed long                 LONG;                   // 32-bit signed
typedef signed long long        LONGLONG;               // 64-bit signed
typedef union _BYTE_VAL
{
   BYTE Val;
   struct
   {
      unsigned char b0:1;
      unsigned char b1:1;
      unsigned char b2:1;
      unsigned char b3:1;
      unsigned char b4:1;
      unsigned char b5:1;
      unsigned char b6:1;
      unsigned char b7:1;
   } bits;
} BYTE_VAL;

typedef union _CHAR_VAL
{
   CHAR Val;
   struct
   {
      unsigned char b0:1;
      unsigned char b1:1;
      unsigned char b2:1;
      unsigned char b3:1;
      unsigned char b4:1;
      unsigned char b5:1;
      unsigned char b6:1;
      unsigned char b7:1;
   } bits;
} CHAR_VAL;

typedef union _WORD_VAL
{
   WORD Val;
   BYTE v[2];
   struct
   {
      BYTE LB;
```

```
        BYTE HB;
    } byte;
    struct
    {
        unsigned char b0:1;
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char b5:1;
        unsigned char b6:1;
        unsigned char b7:1;
        unsigned char b8:1;
        unsigned char b9:1;
        unsigned char b10:1;
        unsigned char b11:1;
        unsigned char b12:1;
        unsigned char b13:1;
        unsigned char b14:1;
        unsigned char b15:1;
    } bits;
} WORD_VAL;

typedef union _SHORT_VAL
{
    SHORT Val;
    BYTE  v[2];
    struct
    {
        BYTE LB;
        BYTE HB;
    } byte;
    struct
    {
        unsigned char b0:1;
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char b5:1;
        unsigned char b6:1;
        unsigned char b7:1;
        unsigned char b8:1;
        unsigned char b9:1;
        unsigned char b10:1;
        unsigned char b11:1;
        unsigned char b12:1;
        unsigned char b13:1;
        unsigned char b14:1;
        unsigned char b15:1;
    } bits;
} SHORT_VAL;

typedef union _DWORD_VAL
{
    DWORD Val;
        WORD w[2];
    BYTE v[4];
    struct
    {
        WORD LW;
```

```
    WORD HW;
  } word;
  struct
  {
    BYTE LB;
    BYTE HB;
    BYTE UB;
    BYTE MB;
  } byte;
  struct
  {
    unsigned char b0:1;
    unsigned char b1:1;
    unsigned char b2:1;
    unsigned char b3:1;
    unsigned char b4:1;
    unsigned char b5:1;
    unsigned char b6:1;
    unsigned char b7:1;
    unsigned char b8:1;
    unsigned char b9:1;
    unsigned char b10:1;
    unsigned char b11:1;
    unsigned char b12:1;
    unsigned char b13:1;
    unsigned char b14:1;
    unsigned char b15:1;
    unsigned char b16:1;
    unsigned char b17:1;
    unsigned char b18:1;
    unsigned char b19:1;
    unsigned char b20:1;
    unsigned char b21:1;
    unsigned char b22:1;
    unsigned char b23:1;
    unsigned char b24:1;
    unsigned char b25:1;
    unsigned char b26:1;
    unsigned char b27:1;
    unsigned char b28:1;
    unsigned char b29:1;
    unsigned char b30:1;
    unsigned char b31:1;
  } bits;
} DWORD_VAL;

typedef union _LONG_VAL
{
  LONG Val;
       WORD w[2];
  BYTE v[4];
  struct
  {
    WORD LW;
    WORD HW;
  } word;
  struct
  {
    BYTE LB;
    BYTE HB;
    BYTE UB;
```

```
        BYTE MB;
    } byte;
    struct
    {
        unsigned char b0:1;
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char b5:1;
        unsigned char b6:1;
        unsigned char b7:1;
        unsigned char b8:1;
        unsigned char b9:1;
        unsigned char b10:1;
        unsigned char b11:1;
        unsigned char b12:1;
        unsigned char b13:1;
        unsigned char b14:1;
        unsigned char b15:1;
        unsigned char b16:1;
        unsigned char b17:1;
        unsigned char b18:1;
        unsigned char b19:1;
        unsigned char b20:1;
        unsigned char b21:1;
        unsigned char b22:1;
        unsigned char b23:1;
        unsigned char b24:1;
        unsigned char b25:1;
        unsigned char b26:1;
        unsigned char b27:1;
        unsigned char b28:1;
        unsigned char b29:1;
        unsigned char b30:1;
        unsigned char b31:1;
    } bits;
} LONG_VAL;

typedef union _QWORD_VAL
{
    QWORD Val;
        DWORD d[2];
        WORD w[4];
    BYTE v[8];
    struct
    {
        DWORD LD;
        DWORD HD;
    } dword;
    struct
    {
        WORD LW;
        WORD HW;
        WORD UW;
        WORD MW;
    } word;
    struct
    {
        unsigned char b0:1;
        unsigned char b1:1;
```

```
unsigned char b2:1;
unsigned char b3:1;
unsigned char b4:1;
unsigned char b5:1;
unsigned char b6:1;
unsigned char b7:1;
unsigned char b8:1;
unsigned char b9:1;
unsigned char b10:1;
unsigned char b11:1;
unsigned char b12:1;
unsigned char b13:1;
unsigned char b14:1;
unsigned char b15:1;
unsigned char b16:1;
unsigned char b17:1;
unsigned char b18:1;
unsigned char b19:1;
unsigned char b20:1;
unsigned char b21:1;
unsigned char b22:1;
unsigned char b23:1;
unsigned char b24:1;
unsigned char b25:1;
unsigned char b26:1;
unsigned char b27:1;
unsigned char b28:1;
unsigned char b29:1;
unsigned char b30:1;
unsigned char b31:1;
unsigned char b32:1;
unsigned char b33:1;
unsigned char b34:1;
unsigned char b35:1;
unsigned char b36:1;
unsigned char b37:1;
unsigned char b38:1;
unsigned char b39:1;
unsigned char b40:1;
unsigned char b41:1;
unsigned char b42:1;
unsigned char b43:1;
unsigned char b44:1;
unsigned char b45:1;
unsigned char b46:1;
unsigned char b47:1;
unsigned char b48:1;
unsigned char b49:1;
unsigned char b50:1;
unsigned char b51:1;
unsigned char b52:1;
unsigned char b53:1;
unsigned char b54:1;
unsigned char b55:1;
unsigned char b56:1;
unsigned char b57:1;
unsigned char b58:1;
unsigned char b59:1;
unsigned char b60:1;
unsigned char b61:1;
unsigned char b62:1;
```

```
        unsigned char b63:1;
    } bits;
} QWORD_VAL;
#endif //__GENERIC_TYPE_DEFS_H_
```