การพัฒนาระบบสมองกลฝังตัวแบบทันเวลาบนเอฟพีจีเอ โดยใช้ไมโครเบลสและไมโครซีโอเอสทูว์

Development of Embedded Real-time System on FPGA Using MicroBlaze and MicroC/OS-II

พิณพงศ์ ศิลป์สกุลสุข

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2550

ปริญญานิพนธ์ปีการศึกษา 2550

ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การพัฒนาระบบสมองกลฝังตัวแบบทันเวลาบนเอฟพีจีเอโดยใช้ไมโครเบลสและไมโครซีโอเอสทูว์

Development of Embedded Real-time System on FPGA Using MicroBlaze and MicroC/OS-II

ผู้จัดทำ

1. นายพิณพงศ์ ศิลป์สกุลสุข รหัสนักศึกษา 47010517

อาจารย์ที่ปรึกษา

(ผู้ช่วยศาสตราจารย์อภิเนตร อูนากูล)

การพัฒนาระบบสมองกลฝังตัวแบบทันเวลาบนเอฟพีจีเอ โดยใช้ไมโครเบลสและไมโครซีโอเอสทูว์

นายพิณพงศ์ ศิลป์สกุลสุข 47010517

ผศ.อภิเนตร อูนากูล อาจารย์ที่ปรึกษา

ปีการศึกษา 2550

บทคัดย่อ

ในปัจจุบันนี้ฮาร์ดแวร์ที่มีความยืดหยุ่นและสามารถทำการโปรแกรมได้นั้นเริ่มเข้ามามีบทบาทสำคัญในการพัฒนาระบบสมองกลฝังตัวค่อนข้างมาก แต่อย่างไรก็ตาม ข้อมูลต่างๆ ที่จำเป็นต่อการเรียนรู้นั้นยังมีค่อนข้างที่จะจำกัดซึ่งทำให้การศึกษานั้นเป็นไปได้ค่อนข้างที่จะลำบาก ในโครงงานนี้เราได้ทำการนำเสนอแนวทางในการสร้างระบบร่วมระหว่างซอฟโพรเซสเซอร์คอร์ ไมโครเบลส กับเรียลไทม์โอเอส ไมโครซีโอเอสทูว์ บนเอฟพีจีเอ ซึ่งเราได้ทำการเรียบเรียงเอกสารสำหรับนำไปศึกษา และทำการพัฒนาไลบรารี่พื้นฐานต่างๆไว้บางส่วน

Development of Embedded Real-time System on FPGA

Using MicroBlaze and MicroC/OS-II

Mr. Pinnapong Silpsakulsuk 47010517

Asst. Prof. Apinetr Unakul Advisor

Academic Year 2007

ABSTRACT

In recent years, agile and reprogrammable hardware has become an important requirements in embedded system development. However, the there are limited resources with steep learning curve. In this project we introduced an integrated implementation of highly agile solution on FPGA with soft processor core using Xilinx MicroBlaze and with RTOS using uC/OS-II. We have developed relevant reference materials, user manual, and basic component library.

กิตติกรรมประกาศ

ปริญญานิพนธ์ฉบับนี้คงไม่อาจสำเร็จได้ด้วยดี หากไม่ได้รับความช่วยเหลือ และร่วมมือจากหลายๆ ฝ่ายด้วยกัน บุคคลแรกที่ต้องกล่าวถึง เพราะเป็นส่วนสำคัญที่ทำให้ปริญญานิพนธ์เสร็จลงได้ก็คือ ผศ. อภิเนตร อูนากูล ซึ่งเป็นที่ปรึกษาปริญญานิพนธ์ ที่ให้ความเอาใจใส่ แนะนำความรู้และความช่วยเหลือในทุกๆ ด้านเสมอมา ซึ่งต้องขอกราบขอบพระคุณเป็นอย่างยิ่ง นอกจากนี้ผู้ที่ต้องขอขอบพระคุณเป็นอย่างสูงอีกท่านหนึ่ง คือ คุณธนพร แสงไพฑูรย์ และบริษัทดีไซน์เกทเวย์ จำกัดที่เอื้อเฟื้อ อุปกรณ์ และเป็นผู้ที่คอยให้คำแนะนำดีๆ ในหลายๆ อย่างอีกท่านหนึ่ง

และต้องขอขอบพระคุณบุคคลสำคัญที่สุด ที่ทำให้ข้าพเจ้ามีวันนี้ ก็คือ บิดา มารดา อันเป็นที่เคารพรักยิ่ง ซึ่งได้เลี้ยงดูผู้เขียนมาเป็นอย่างดี พร้อมทั้งให้โอกาสในการศึกษาอย่างเต็มที่ และยังให้กำลังใจ เอาใจใส่เสมอมาในทุกๆ ด้าน อันหาที่เปรียบมิได้ ข้าพเจ้าขอระลึกในพระคุณอันสุดประมาณ และขอกราบขอบพระคุณมา ณ ที่นี้

พิณพงศ์ ศิลป์สกุลสุข

สารบัญ

หน้า

สารบัญ(ต่อ)

หน้า

สารบัญ(ต่อ)

หน้า

สารบัญ(ต่อ)

หน้า

สารบัญ(ต่อ)

หน้า

สารบัญรูป

หน้า

สารบัญรูป(ต่อ)

หน้า

สารบัญรูป(ต่อ)

หน้า

สารบัญรูป(ต่อ)

หน้า

สารบัญรูป(ต่อ)

หน้า

สารบัญตาราง

หน้า

บทที่1

บทนำ

1.1 ความเป็นมาของปัญหา

เนื่องด้วยในปัจจุบันนี้ ระบบสมองกลฝังตัวนั้นได้เข้ามามีบทบาทในชีวิตประจำวันของเรามากขึ้นทุกวัน และด้วยเทคโนโลยีในปัจจุบันนี้ หน่วยประมวลผลที่ใช้ในระบบสมองกลฝังตัวนั้นเริ่มมีประสิทธิภาพสูงขึ้นเรื่อยๆ จึงทำให้สามารถนำหน่วยประมวลผลเหล่านี้ไปทำงานที่มีความซับซ้อนได้มากขึ้น หรือแม้กระทั่งทำงานหลายๆอย่างไปพร้อมๆกัน ในงานบางชนิดนั้น ระบบมีความจำเป็นอย่างยิ่งที่จะต้องตอบสนองเหตุการณ์ต่างๆให้ทันภายในระยะเวลาที่กำหนด ระบบประเภทนี้เรียกว่าระบบทันเวลา หรือ เรียลไทม์ซิสเต็ม(real-time system) ซึ่งการออกแบบ และสร้างระบบประเภทเรียลไทม์ซิสเต็มนี้ค่อนข้างมีความซับซ้อนและมีสิ่งที่จะต้องระวังอยู่มาก ทางออกที่ดีทางหนึ่งนั้นคือการใช้งานระบบปฏิบัติการแบบทันเวลา หรือ เรียลไทม์โอเปอเรติ้งซิสเต็มนั่นเอง

ในระบบสมองกลฝังตัวนั้น หน่วยประมวลผลมีความจำเป็นที่จะต้องติดต่อกับอุปกรณ์ต่างๆ หลายชนิด วงจรรวมประเภทไมโครคอนโทรลเลอร์ในปัจจุบันจึงพยายามผนวกเอาอุปกรณ์ต่างๆเข้าไว้ในตัวมากขึ้นเรื่อยๆ และเรียกตัวเองว่าเป็นวงจรรวมชนิด ซิสเต็มออนชิพ(system on chip) หรือ เอสโอซี(SoC) แต่อย่างไรก็ตามอุปกรณ์เหล่านี้ก็ยังไม่เพียงพอกับความต้องการ เนื่องจากแต่ละระบบนั้นมีความต้องการอุปกรณ์ที่แตกต่างกันออกไปการเลือกใช้เอฟพีจีเอที่มีโพรเซสเซอร์คอร์ฝังอยู่จึงเป็นอีกทางออกหนึ่งที่น่าสนใจ เพราะเราสามารถออกแบบวงจรบนเอฟพีจีเอให้เป็นอุปกรณ์ที่เหมาะสมกับระบบของเราได้เอง แต่อย่างไรก็ตามเอฟพีจีเอประเภทที่มีโพรเซสเซอร์คอร์ฝังอยู่นี้ ในปัจจุบันยังมีราคาค่อนข้างสูง ไม่คุ้มค่ากับการนำมาใช้งานจริง อีกทางออกหนึ่งที่ค่อนข้างจะลงตัวคือการเลือกใช้ ซอฟท์โพรเซสเซอร์คอร์ ซึ่งใช้พื้นที่ของเอฟพีจีเอในการสังเคราะห์ตัวโพรเซสเซอร์นี้ขึ้นมา ถึงแม้ว่าประสิทธิภาพจะสู้โพรเซสเซอร์คอร์จริงๆไม่ได้ และเปลืองพื้นที่ของตัวเอฟพีจีเอไปบ้าง แต่หากนำเรื่องราคาเข้ามาเปรียบเทียบด้วยแล้ว การใช้ซอฟท์โพรเซสเซอร์คอร์เช่นนี้คุ้มแค่ต่อการนำมาใช้ในระบบจริงมากกว่าการใช้เอฟพีจีเอที่มีโพรเซสเซอร์คอร์จริงๆ ทางผู้จัดทำโครงงานจึงมีแนวคิดที่จะออกแบบระบบสมองกลฝังตัวแบบทันเวลาบนเอฟพีจีเอขึ้น โดยใช้ซอฟท์โพรเซสเซอร์คอร์และเรียลไทม์โอเปอเรติ้งซิสเต็มที่ได้กล่าวไว้ในข้างต้น พร้อมทั้งทำเอกสาร

อธิบายขั้นตอนการออกแบบโดยละเอียดเพื่อเป็นประโยชน์แก่ผู้ที่ต้องการนำระบบในลักษณะนี้ไปใช้งานจริง

1.2 วัตถุประสงค์ของโครงงาน

- เพื่อศึกษาและรวบรวมข้อมูลที่เกี่ยวกับวิธีการสร้างระบบแบบเรียลไทม์บนแพลตฟอร์มที่มีความยืดหยุ่นสูง
- เพื่อสร้างระบบเรียลไทม์ตัวอย่างบนแพลตฟอร์มที่มีความยืดหยุ่นสูง

1.3 ประโยชน์ที่คาดว่าจะได้รับ

- ระบบสมองกลฝังตัวแบบทันเวลาบนเอฟพีจีเอ
- เอกสารอธิบายขั้นตอนการออกแบบของระบบอย่างละเอียด

1.4 ขอบเขตของโครงงาน

- รวบรวมข้อมูลเกี่ยวกับการใช้งานไมโครซีโอเอสทูว์ และไมโครเบลส
- ทดลองสร้างระบบตัวอย่างโดยใช้ไมโครซีโอเอสทูว์ บนไมโครเบลส
- เรียบเรียงวิธีการในการสร้างระบบเรียลไทม์โดยใช้ไมโครซีโอเอสทูว์ และไมโครเบลส

1.5 ส่วนประกอบของรายงาน

- บทที่ 1 กล่าวถึงความเป็นมาของปัญหา วัตถุประสงค์ของโครงงาน ประโยชน์ที่คาดว่าจะได้รับ ขอบเขตของโครงงาน และส่วนประกอบของรายงานฉบับนี้
- บทที่ 2 กล่าวถึงทฤษฎีพื้นฐานเกี่ยวกับระบบทันเวลา หรือเรียลไทม์ซิสเต็ม
- บทที่ 3 กล่าวถึงสถาปัตยกรรมของซอฟท์โพรเซสเซอร์คอร์ ไมโครเบลส ที่ใช้ในโครงงาน
- บทที่ 4 เป็นเอกสารประกอบการทดลองการสร้างไมโครเบลสซิสเต็มอย่างง่าย
- บทที่ 5 เป็นเอกสารประกอบการทดลองการใช้งานจีพีไอโอบนไมโครเบลสซิสเต็ม
- บทที่ 6 เป็นเอกสารประกอบการทดลองการใช้งานไทม์เมอร์และอินเทอร์รัพท์บนไมโครเบลส
- บทที่ 7 เป็นเอกสารประกอบการทดลองการสร้างไอพีคอร์เพื่อใช้งานบนไมโครเบลส
- บทที่ 8 เป็นเอกสารประกอบการทดลองการนำไมโครซีโอเอสทูว์ไปใช้งานบนระบบไมโครเบลส
- บทที่ 9 เป็นเอกสารประกอบโปรแกรมประยุกต์ตัวอย่างที่สร้างบนไมโครซีโอเอสทูว์บนระบบไมโครเบลส
- บทที่ 10 เป็นสรุปผลการดำเนินงานของโครงงาน

บทที่10

บทวิจารณ์และบทสรุป

10.1 สรุปผลการทำงาน

1. สามารถสร้างระบบไมโครเบลสเพื่อใช้งานบนเอฟพีจีเอได้
2. สามารถ เพิ่มและลดอุปกรณ์ รวมถึงปรับแต่งระบบไมโครเบลสที่สร้างขึ้นได้
3. สามารถสร้างอุปกรณ์ของตนเองบนเอฟพีจีเอและนำไปเชื่อมต่อกับระบบไมโครเบลสได้
4. สามารถเขียนโปรแกรมบนระบบไมโครเบลสได้
5. สามารถนำไมโครซีโอเอสทูว์ไปใช้งานบนไมโครเบลสได้
6. สามารถเขียนโปรแกรมประยุกต์บนไมโครซีโอเอสทูว์ได้
7. สร้างโปรแกรมประยุกต์ตัวอย่างพร้อมทั้งเอกสารประกอบการออกแบบ

10.2 ปัญหาและอุปสรรคในการทำงาน

1. ไมโครซีโอเอสทูว์มีขนาดใหญ่และไม่สามารถนำไปใช้งานบนเอฟพีจีเอสปาร์ทาน3ได้เนื่องจากขนาดของบลอคแรมไม่เพียงพอ
2. การเราท์วงจรภายในเอฟพีจีเอที่ต้องทำงานบนความถี่สูงๆ ระดับ 100 เมกะเฮิร์ทซ์ ใช้เวลาในการเราท์ที่นานมาก และหากทดสอบแล้วเกิดความผิดพลาด มีการแก้ไขวงจรจะต้องทำการเราท์ใหม่

10.3 แนวทางแก้ไข

1. เปลี่ยนไปใช้เอฟพีจีเอที่มีขนาดของบลอคแรมสูงขึ้นเช่นเวอร์เท็กซ์2
2. ทำการเราท์วงจรที่ทำงานที่ความถี่ต่ำๆแล้วนำมาทดลองให้แน่ใจว่าทำงานได้ถูกต้องก่อนแล้วจึงเราท์วงจรที่ความถี่สูงๆครั้งเดียว

10.4 บทวิจารณ์โครงงาน

การออกแบบระบบเรียลไทม์โดยใช้ซอฟโพรเซสเซอร์คอร์บนเอฟพีจีเอนั้นสามารถทำได้ และนำไปใช้งานได้จริง อีกทั้งยังมีข้อดีในเรื่องของความยืดหยุ่นของระบบ เนื่องจากเราสามารถเพิ่ม ลด อุปกรณ์ภายในตัวชิพได้ตามต้องการ หรือแม้แต่จะสร้างอุปกรณ์ของตนเองเพิ่มเติมก็สามารถทำได้ อีกทั้งยังสามารถนำอุปกรณ์สำเร็จที่ผู้อื่นสร้างไว้เข้ามาเสริมได้อีกด้วย แต่ไม่เหมาะกับการนำไปใช้งานในระบบเล็กๆ เนื่องจากต้องอาศัยความรู้ความเข้าใจ และมีขั้นตอนในการพัฒนาที่ค่อนข้างจะซับซ้อนกว่าการใช้งานไมโครคอนโทรลเลอร์ทั่วไป และไม่เหมาะกับการนำไปใช้งานในระบบที่ต้องการประสิทธิภาพสูงเนื่องจากเอฟพีจีเอนั้นมีดีเลย์สูงกว่าวงจรรวมที่ออกแบบมาเฉพาะงาน

10.5 แนวทางการพัฒนา

ในงานอุตสาหกรรมจริงๆ โปรแกรมประยุกต์ที่เราทำการสร้างนั้นอาจมีขนาดใหญ่เกินกว่าที่จะเก็บชุดคำสั่งทั้งหมดไว้ในบลอคแรมได้ และในโครงงานนี้ยังไม่ได้กล่าวถึงการนำหน่วยความจำแบบแฟลชมาเชื่อมต่อเป็นหน่วยความจำภายนอกของเอฟพีจีเอ แล้วนำชุดคำสั่งไปเก็บไว้ในหน่วยความจำแบบแฟลชนั้นๆ และเขียนโปรแกรมบูทโหลดเดอร์เพื่อให้ไมโครเบลสไปเรียกชุดคำสั่งจากแฟลชขึ้นมาทำงาน ซึ่งค่อนข้างที่จะมีขั้นตอนที่ต้องศึกษาอีกพอสมควร

บทที่2
พื้นฐานเกี่ยวกับเรียลไทม์ซิสเต็ม

2.1 บทนำ

เรียลไทม์ซิสเต็ม(realtime system) เป็นระบบการทำงานที่ไม่สามารถคาดการได้ว่าช่วงเวลาของอินพุตที่เข้ามาจะเกิดขึ้นเมื่อไหร่ และต้องทำงานให้ได้ผลลัพธ์ออกไปให้ทันเวลา ซึ่งเรียลไทม์ซิสเต็มสามารถแบ่งออกได้เป็น 2 ประเภท คือ ซอฟต์เรียลไทม์ซิสเต็ม (Soft realtime system) และ ฮาร์ดเรียลไทม์ซิสเต็ม (Hard real time system) ซอฟต์เรียลไทม์ซิสเต็ม นั้นเป็นเรียลไทม์ซิสเต็มที่อาจจะมีความล่าช้าได้บ้าง ซึ่งจะไม่ส่งผลกระทบร้ายแรงต่อระบบเช่นระบบการอัพเดทตารางการบิน ตารางรถไฟ ซึ่งอาจจะล่าช้าได้หลายวินาทีโดยไม่เกิดความเสียหายอะไร ส่วนฮาร์ดเรียลไทม์ซิสเต็มนั้นระบบจะต้องทำงานให้ถูกต้องและต้องไม่เกินตามเวลาที่กำหนดไว้ ซึ่งความล่าช้าที่เกิดขึ้นในระบบนั้นเป็นสิ่งที่รับไม่ได้และจะส่งผลกระทบให้เกิดความเสียหายได้เช่นระบบรักษาอุณหภูมิของเครื่องยนต์ หากทำงานล่าช้าไปอาจทำให้เครื่องยนต์นั้นเกิดความเสียหายได้ โดยทั่วไปแล้วเรียลไทม์ซิสเต็มที่เราพบเห็นอยู่ทั่วไปนั้นก็จะมีทั้งซอฟต์เรียลไทม์และฮาร์ดเรียลไทม์

โดยส่วนใหญ่แล้วเรียลไทม์ซิสเต็มมักจะอยู่ในรูปแบบสมองกลฝังตัว (embedded) ซึ่งเสมือนกับว่ามีคอมพิวเตอร์รวมอยู่ในระบบ โดยที่ผู้ใช้งานไม่รู้สึกตัว ตัวอย่างของสมองกลฝังตัวมีดังนี้

- ด้านการสื่อสาร เช่น เราเตอร์(Router) สวิตช์ (Switch)

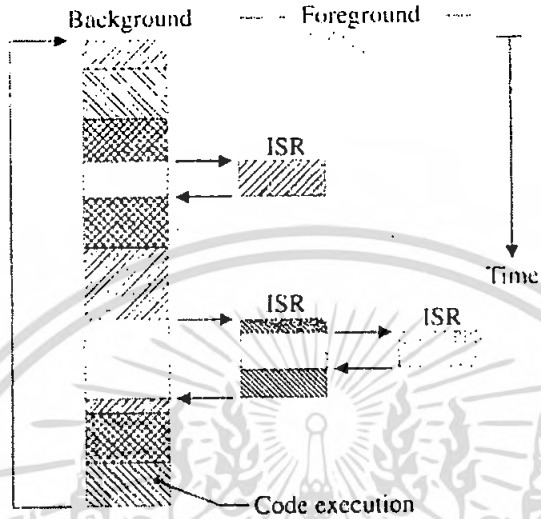
- ด้านอวกาศ เช่น ระบบจัดการการบิน ระบบเครื่องยนต์ เป็นต้น

โดยทั่วไปแล้วเรียลไทม์ซอฟแวร์แอพพลิเคชั่น(Realtime software application)จะออกแบบได้ยากกว่านอนเรียลไทม์แอพพลิเคชัน (non real time application)

2.2 ชนิดของเรียลไทม์ซิสเต็ม

2.2.1 ฟอร์กราวด์และแบคกราวด์ซิสเต็ม(Foreground / Background System)

ระบบขนาดเล็กที่มีความซับซ้อนน้อยมักจะออกแบบดังนี้



รูปที่ 2.1ฟอร์กราวด์/แบ็กกราวด์ซิสเต็ม (Foreground/Background System)

ระบบที่เรียกว่าฟอร์กราว/แบ็กกราวด์(foreground/background) หรือซูเปอร์ลูป (super-loop) ตัวแอพพลิเคชั่นจะประกอบด้วย infinite loop ซึ่งจะเรียกว่าโมดูล (module) หรือฟังก์ชั่น (function) เพื่อใช้ในการปฏิบัติการเป็นแบ็กกราวด์หรือเรียกว่าทาสก์เลเวล(task level) ส่วนอินเตอร์รัพต์เซอร์วิสรูทีน (interrupt service routines) หรือ ISRจะจัดการกับเหตุการณ์หรืออีเวนต์ (event) ที่เกิดขึ้น โดยจะทำงานเป็นฟอร์กราวด์ หรือจะเรียกอีกอย่างว่าอินเตอร์รัพต์เลเวล (interrupt level) การทำงานที่เกิดขึ้นนั้นจะถูกจัดการ โดยจะใช้ ISR เพื่อรับประกันว่าผลที่ได้จะทันเวลาที่ต้องการ

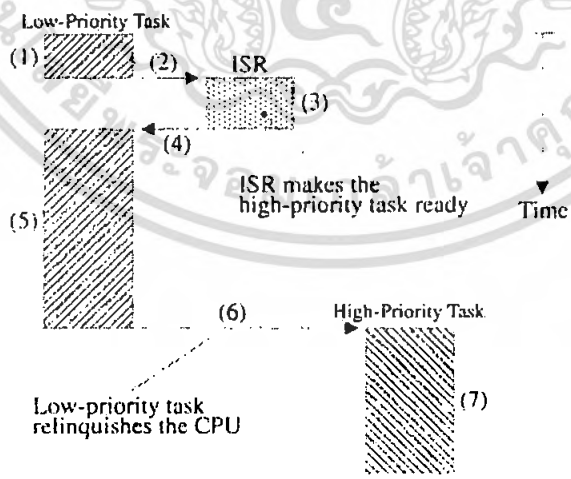
สำหรับข้อมูลของแบ็กกราวด์โมดูลที่ถูกจัดเตรียมโดย ISR นั้นจะไม่ถูกนำไปใช้งานจนกว่าแบ็กกราวด์รูทีนจะเริ่มทำงาน ซึ่งจะเรียกว่าการตอบสนองของทาสก์เลเวล (task level response) กรณีที่แย่ที่สุดของเวลาในการตอบสนองของทาสก์เลเวลขึ้นอยู่กับระยะเวลาที่แบ็กกราวด์ลูปทำงาน เพราะว่าเวลาในการทำงานของโค้ด(code)จะไม่คงที่ โดยขึ้นกับเวลาของลูปซึ่งไม่สามารถคาดการณ์ได้ ฉะนั้น ถ้ามีการเปลี่ยนแปลงโค้ด การทำงานของลูปจะมีผลต่อเวลา

2.2.2 นอนพรีเอมทีฟเคอร์เนล(Non-Preemptive Kernel)

นอนพรีเอมทีฟเคอร์เนล (Non-preemptive kernel) จะให้แต่ละทาสก์ที่ทำงานอยู่ภายในซีพียู ทำงานให้เสร็จก่อน ซึ่งการจัดลำดับแบบนอนพรีเอมทีฟ (Non-preemtive scheduling) จะเรียกว่าโคโอเปอร์เรทีฟมัลติทาสกิ้ง (cooperative multitasking) โดยทาสก์จะร่วมมือกับทาสก์อื่น ๆ ในการใช้งานซีพียูร่วมกัน ส่วนอีเวนท์ที่เกิดขึ้นจะถูกจัดการโดย ISR ซึ่ง ISR จะทำหน้าที่ให้ทาสก์ที่มีความสำคัญสูงกว่าและพร้อมที่จะทำงานได้ทำงานก่อน แต่หลังจากเสร็จงานที่ทำโดย ISR แล้ว จะย้อนกลับไปทำทาสก์ที่ทำก่อนเกิดอินเตอร์รัพต์ และเมื่อมีทาสก์ใหม่ที่มีความสำคัญสูงกว่าต้องการทำงาน จะต้องรอให้ทาสก์ที่กำลังทำงานอยู่ใน ซีพียูทำงานให้เสร็จก่อน จึงจะสามารถเข้าใช้งานซีพียูได้

ประโยชน์ของนอนพรีเอมทีฟเคอร์เนล คือ ที่ระดับทาสก์เลเวลนั้น นอนพรีเอมทีฟเคอร์เนลสามารถจะใช้นอนรีเอนทรานฟังก์ชั่น(non-reentrant function)ได้ ซึ่งทาสก์สามารถใช้นอนรีเอนทรานฟังก์ชั่นได้โดยไม่ต้องกลัวว่าจะเกิดการเปลี่ยนแปลงค่าโดยทาสก์อื่น เพราะว่าแต่ละทาสก์จะต้องทำงานให้เสร็จก่อนที่เลิกใช้งานซีพียู

ประโยชน์อีกอย่างหนึ่งของนอนพรีเอมทีฟเคอร์เนล คือ ไม่ต้องกังวลเรื่องการร่วมกันใช้ข้อมูล เพราะแต่ละทาสก์จะได้ทำงานให้เสร็จโดยไม่มีทาสก์อื่นเข้ามาใช้งานซีพียู และจะมีการใช้เซมาฟอร์ (semaphore) เพื่อช่วยในการจัดการเรื่องการเข้าใช้งานอุปกรณ์ I/O ต่าง ๆ ร่วมกัน ตัวอย่างเช่น การเข้าใช้งานปริ๊นเตอร์



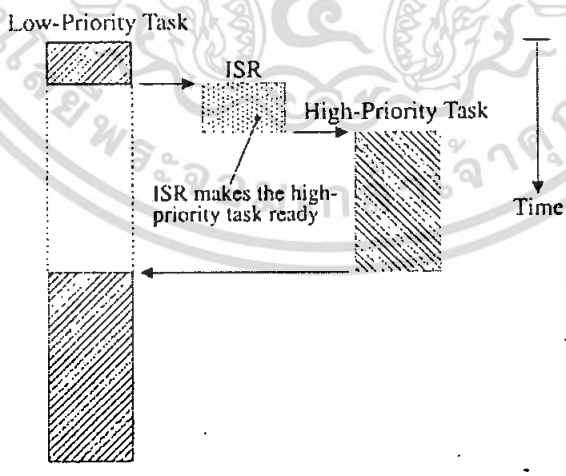
รูปที่ 2.2นอนพรีเอมทีฟเคอร์เนล (non preemtive kernel)

ที่หมายเลข 1 ขณะที่ทาสก์กำลังทำงานอยู่ มีการเกิดอินเตอร์รัพต์ขึ้น ถ้าอินเตอร์รัพต์ได้มีเปิดการใช้งานซีพียูจะทำการกระโดดเข้าไปทำงานใน ISR (หมายเลข 2) ซึ่ง ISR จะจัดการกับอีเวนต์ที่เกิดขึ้น (หมายเลข 3) และจะให้ทาสก์ที่มีความสำคัญสูงกว่าพร้อมที่จะทำงาน ซึ่งหลังจากทำงานในส่วนของ ISR เสร็จแล้วจะย้อนกลับมาทำงานในทาสก์เดิม (หมายเลข 4) และทำงานต่อไปเรื่อย ๆ (หมายเลข 5) หลังจากนั้นเมื่อทาสก์ทำงานเสร็จ มันจะทำการเรียกใช้งานในส่วนเคอร์เนลเพื่อเลิกใช้งานซีพียู (หมายเลข 6) และทาสก์ตัวใหม่จะทำงาน ซึ่งเป็นทาสก์ที่ทำงานเมื่อเกิดเหตุการณ์ขึ้นจาก ISR (หมายเลข 7)

อุปสรรคสำคัญของนอนพรีเอมทีฟเคอร์เนล คือ การเวลาในการตอบสนอง โดยทาสก์ที่มีความสำคัญสูงและพร้อมที่จะทำงาน อาจจะต้องคอยนาน เพราะว่าต้องคอยให้ทาสก์ที่กำลังทำงานอยู่ทำงานเสร็จก่อน โดยในระบบฟอร์กราวด์/แบ็กกราวด์ เวลาการตอบสนองของทาสก์เลเวลในนอนพรีเอมทีฟเคอร์เนลจะไม่สามารถคาดการณ์ได้ ทำให้ไม่รู้ว่าเมื่อไหร่ทาสก์ที่มีความสำคัญสูงจะได้ใช้งานซีพียูซึ่งก็ขึ้นอยู่กับแอพพลิเคชั่นที่สร้างขึ้น

2.2.3 พรีเอมทีฟเคอร์เนล(Preemptive Kernel)

พรีเอมทีฟเคอร์เนล (preemptive kernel) จะใช้งานเมื่อต้องการการตอบสนองเป็นหลัก โดยทาสก์ที่มีความสำคัญสูงและพร้อมที่จะทำงานจะได้ใช้งานซีพียูทันที โดยเมื่อทาสก์ตัวหนึ่งซึ่งมีความสำคัญสูงพร้อมที่จะทำงาน ทาสก์ตัวเดิมที่กำลังทำงานอยู่จะหยุดการทำงานชั่วคราวและให้ทาสก์ที่มีความสำคัญสูงจะเข้าใช้งานซีพียูทันที



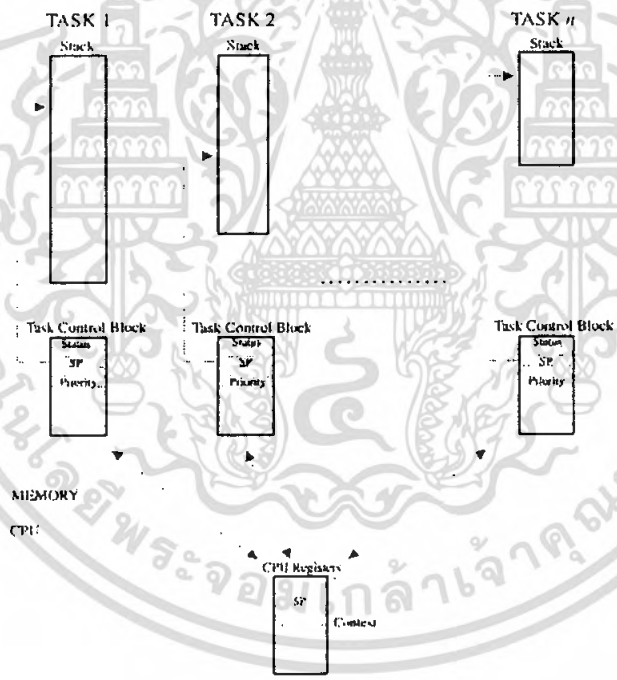
รูปที่ 2.3พรีเอมทีฟเคอร์เนล (preemtive kernel)

แอพพลิเคชั่นที่ใช้งานพรีเอมทีฟเคอร์เนลจะไม่ใช้นอนรีเอนทรานฟังก์ชั่น เพราะไม่สามารถป้องกันการเข้าใช้งานฟังก์ชั่นได้ โดยทาสก์ที่มีความสำคัญสูงและต่ำจะสามารถเข้าใช้งานฟังก์ชั่นทั่วไปได้ ทำให้ข้อมูลอาจเกิดการซ้อนทับกันได้ กรณีเมื่อทาสก์ที่มีความสำคัญสูง ต้องการใช้งานฟังก์ชั่นขณะที่ทาสก์ที่มีความสำคัญต่ำกำลังใช้งานอยู่

2.3 ทาสก์ และ มัลติทาสกิ้ง

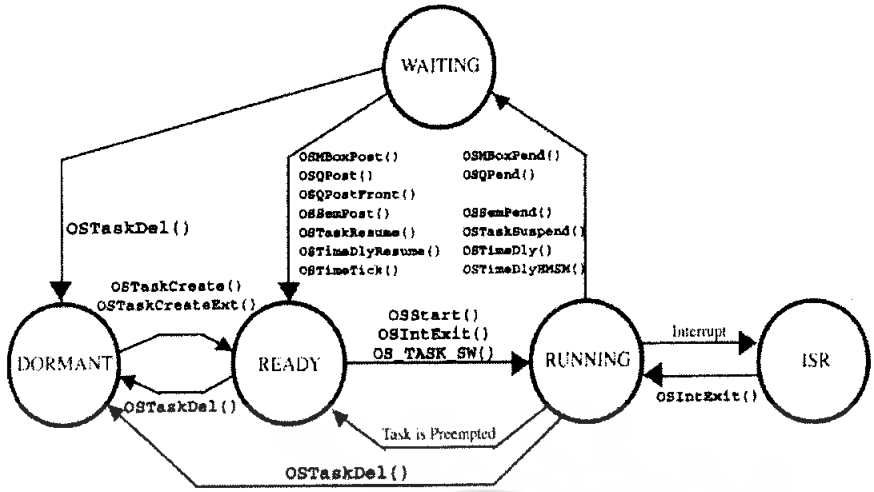
2.3.1 ทาสก์

ทาสก์ (task) คือ ตัวโปรแกรมที่คิดว่ามีซีพียูเป็นของตัวมันเองทั้งหมด การออกแบบการทำงานของเรียลไทม์แอพพลิเคชั่นจะเริ่มจากการแยกหน้าที่ให้แก่ทาสก์รับผิดชอบสำหรับการใช้แก้ปัญหาต่าง ๆ โดยในแต่ละทาสก์จะกำหนดค่าความสำคัญ (priority) , กำหนดคอนเท็กซ์(context)ภายในรีจิสเตอร์ (register)ของซีพียู และกำหนดพื้นที่สแตก (stack) ของแต่ละทาสก์



รูปที่ 2.4มัลติเพิลทาสก์ (Multiple Task)

ในแต่ละทาสก์มักจะอยู่ในสถานะซึ่งจะเป็น 1 ใน 5 สถานะ ได้แก่ DORMANT , READY , RUNNING , WAITING (สำหรับสถานการณ์ต่าง ๆ) และ ISR



รูปที่ 2.5สถานะของทาสก์

- สถานะ DORMANT จะทำหน้าที่สร้างทาสก์ขึ้นมา(แต่ยังไม่พร้อมที่จะทำงาน) และทำลายทาสก์

- ทาสก์จะเป็นสถานะ READY เมื่อทาสก์ที่สร้างมาสามารถพร้อมจะทำงาน แต่ความสำคัญมีค่าต่ำกว่าทาสก์ที่กำลังทำงานอยู่ในสถานะ RUNNING

- สถานะ RUNNING เป็นสถานะที่ทาสก์กำลังทำงานโดยเข้าใช้งานโปรเซสเซอร์

- ทาสก์จะเป็นสถานะ WAITING หรือ BLOCKING เมื่อทาสก์ต้องการให้อีเวนต์เกิดขึ้น เช่น การรอที่จะให้ I/O ทำงานเสร็จ , รอจะใช้งานแชร์รีซอสหรือการรอให้เกิดพัลส์ (pulse) เป็นต้น

- ทาสก์จะเป็นสถานะ ISR เมื่อมีอินเตอร์รัพต์เกิดขึ้น และซีพียูก็จะไปทำงานในส่วนของอินเตอร์รัพต์

2.3.2 มัลติทาสกิ้ง

มัลติทาสกิ้ง (Multitasking) คือกระบวนการจัดลำดับการทำงาน (scheduling) และสวิตชิ่ง (swithcing) ของ CPU (Central Processing Unit) ระหว่างหลาย ๆ ทาสก์โดยซีพียูตัวเดียวจะพิจารณาหลาย ๆ ลำดับของทาสก์ ซึ่งมัลติทาสกิ้งจะคล้ายกับฟอร์คราวด์/แบ็กกราวด์ด้วย โดยจะเป็นในรูปแบบหลาย ๆ แบ็กกราวด์ การทำมัลติทาสกิ้งเป็นการใช้ประโยชน์ของซีพียูให้คุ้มค่าที่สุดและยังเป็นการจัดเตรียมองค์ประกอบให้กับแอพพลิเคชั่นอีกด้วย ลักษณะสำคัญอีกประการหนึ่งของมัลติทาสกิ้งคือสามารถให้โปรแกรมเมอร์

ผู้พัฒนาแอพพลิเคชั่นสามารถจัดการความซับซ้อนที่มีในตัวเรียลไทม์แอพพลิเคชั่นได้โดยไม่ต้องคำนึงถึงแอพพลิเคชันตัวอื่น ทำให้แอพพลิเคชันโปรแกรมง่ายต่อการออกแบบและดูแล

2.3.3 คอนเท็กซ์สวิตชิ่ง (context switching)

เมื่อมัลติทาสกิ้งเคอร์เนลทำการเปลี่ยนทาสก์ที่จะทำงาน จะทำการเก็บค่าคอนเท็กซ์ของทาสก์ (task context) หรือซีพียูรีจิสเตอร์ (CPU registers) เอาไว้ในพื้นที่สำหรับเก็บซึ่งเป็นสแตกของทาสก์นั้น เมื่อได้ทำการเก็บค่าเสร็จแล้วทาสก์ตัวใหม่จะทำการดึงค่าคอนเท็กซ์ (context) จากพื้นที่สำหรับเก็บค่า เพื่อนำกลับมาการทำงานต่อ ซึ่งการทำแบบนี้เรียกว่าคอนเท็กซ์สวิตชิ่ง (context switch) หรือทาสก์สวิตชิ่ง (task switch) โดยคอนเท็กซ์สวิตชิ่งจะมีการเพิ่มโอเวอร์เฮด (overhead) ให้กับแอพพลิเคชัน โดยซีพียูมีรีจิสเตอร์มากเท่าไรก็จะมีโอเวอร์เฮดมากขึ้น ส่วนเวลาในการทำคอนเท็กซ์สวิตชิ่งจะขึ้นกับจำนวนรีจิสเตอร์ที่ทำการเก็บและดึงกลับมาโดยซีพียู

2.3.4 เคอร์เนล

เคอร์เนล (kernel) เป็นส่วนหนึ่งของระบบมัลติทาสกิ้ง ซึ่งจะรับผิดชอบโดยทำหน้าที่ในการจัดการทาสก์ เช่น การจัดการเวลาของซีพียูและใช้ในการติดต่อระหว่างทาสก์ การให้บริการทั่วไปของเคอร์เนล คือ การทำคอนเท็กซ์สวิตชิ่ง ซึ่งการใช้งานเรียลไทม์เคอร์เนลทั่วไปจะใช้ในการจัดการทาสก์ โดยการออกแบบตัวระบบให้แอพพลิเคชั่นสามารถแบ่งออกได้หลาย ๆ ทาสก์ โดยเคอร์เนลจะมีการเพิ่มโอเวอร์เฮดเข้าไปในระบบเพราะต้องการใช้รอม (ROM) หรือพื้นที่โค้ด (code space) และแรม (RAM) ในส่วนของโครงสร้างข้อมูลของเคอร์เนลเพิ่ม ซึ่งต้องระวังเพราะธรรมดาแล้ว แต่ละทาสก์จะต้องการพื้นที่สแตกเป็นของตัวเอง ซึ่งมีแนวโน้มว่าจะกินพื้นที่ในแรมสูงพอสมควร

2.3.5 ตัวจัดลำดับงาน

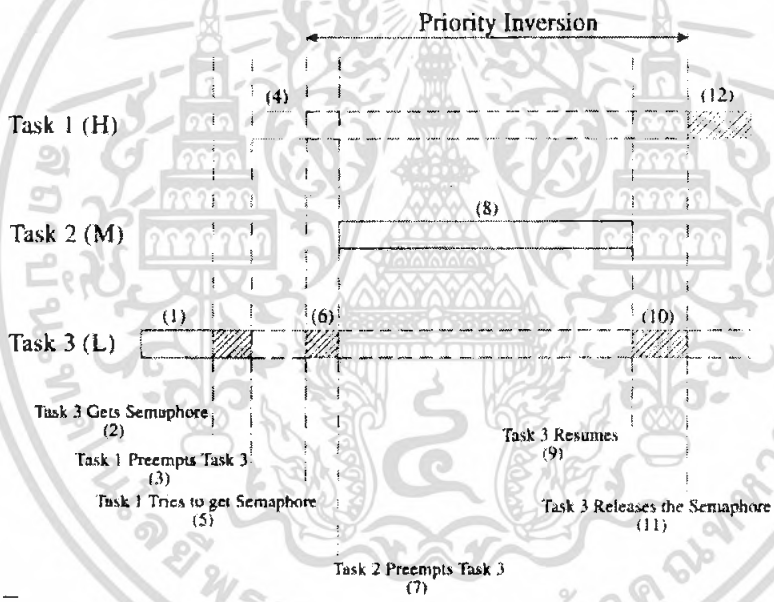
ตัวจัดลำดับงาน (scheduler) หรือเรียกว่าดิสแพทเชอร์ (dispatcher) เป็นส่วนหนึ่งในเคอร์เนลที่จะช่วยตัดสินว่าจะให้ทาสก์ตัวไหนทำงานต่อไป ซึ่งเรียลไทม์เคอร์เนลจะคิดจากค่าความสำคัญ(priority)เป็นหลัก โดยจะให้ทาสก์ที่มีความสำคัญสูงที่สุดและพร้อมที่จะทำงานได้ทำงานก่อน แต่อย่างไรก็ตามการตัดสินว่าจะให้ทาสก์ตัวไหนทำงาน จะขึ้นกับชนิดของเคอร์เนลด้วย โดยเคอร์เนลที่อาศัยค่าความสำคัญเป็นหลัก ประกอบไปด้วย 2 ชนิด นอนพรีเอมทีฟเคอร์เนล (non-preemptive kernel) และพรีเอมทีฟเคอร์เนล (preemtive kernel)

2.3.6 ความสำคัญของทาสก์

ความสำคัญ (priority) จะกำหนดให้ในแต่ละทาสก์ โดยทาสก์ที่มีความสำคัญจะถูกกำหนดให้มีความสำคัญสูงซึ่งแบ่งได้เป็นความสำคัญแบบสแตติก (static priority) และไดนามิก (dynamic priority) ซึ่งทาสก์จะเป็นแบบสแตติกเมื่อแต่ละทาสก์ไม่มีการเปลี่ยนความสำคัญระหว่างที่แอพพลิเคชั่นกำลังทำงาน โดยแต่ละทาสก์จะถูกกำหนดความสำคัญมาตั้งแต่แรกแล้ว ส่วนความสำคัญแบบไดนามิก ทาสก์สามารถเปลี่ยนความสำคัญได้ระหว่างที่แอพพลิเคชันกำลังทำงาน แต่อาจจะเกิดปัญหาคือไพรออริตี้อินเวอร์ชัน (priority inversion) ได้ ซึ่งในเรียลไทม์เคอร์เนลจะมีการป้องกันการเกิดไพรออริตี้อินเวอร์ชัน

2.3.7 พริออริตี้อินเวอร์ชัน

ไพรออริตี้อินเวอร์ชัน (priority inversion) เป็นปัญหาในเรียลไทม์ซิสเต็มและมักจะเกิดขึ้นกับเรียลไทม์เคอร์เนล



รูปที่ 2.6ปัญหาไพรออริตี้อินเวอร์ชัน (priority inversion problem)

กำหนดให้ทาสก์ 1 มีค่าความสำคัญสูงที่สุด ทาสก์ 2 รองลงมาและทาสก์ 3 มีความสำคัญต่ำที่สุด โดยทาสก์ 1 และทาสก์ 2 จะรอให้เกิดอีเวนต์ขึ้นก่อน

- ที่หมายเลข 1 ในขณะที่ทาสก์ 3 กำลังทำงานอยู่ ทาสก์ 3 ต้องการใช้งานทรัพยากรจึงเข้าใช้งานเซ - มาฟอร์ (semaphore) ที่หมายเลข 2

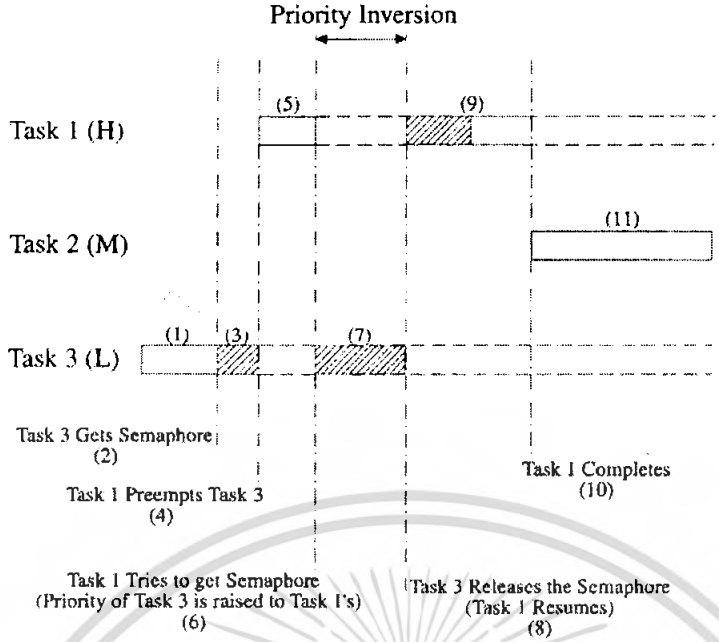
- ในขณะที่ทาสก์ 3 กำลังทำงานอยู่กับทรัพยากร(หมายเลข 4)นั้น ทาสก์ 1 ก็ได้เริ่มทำงานขึ้น(หมายเลข 2) ซึ่งทาสก์ 3 ก็จะหยุดทำงานชั่วคราว

- เมื่อทาสก์ 1 ทำงานไปขณะหนึ่ง ต้องการที่จะใช้งานทรัพยากร(หมายเลข 5) แต่ทาสก์ 3 กำลังทำงานอยู่ในทรัพยากร ดังนั้นทาสก์ 1 จึงพยายามเข้าใช้งานเซมาฟอร์ทำให้ทาสก์ 1 ถูกหยุดการทำงานไปชั่วขณะและทาสก์ 3 ก็จะกลับมาทำงานต่ออีกครั้ง(หมายเลข 6)

- เมื่อทาสก์ 3 ทำงานไปได้ซักพักจะมีอีเวนต์เกิดขึ้นมาและทาสก์ 2 เริ่มทำงาน (หมายเลข 7 และ 8) ทาสก์ 3 จึงหยุดทำงานเพื่อให้ทาสก์ 2 ทำงาน เมื่อทาสก์ 2 ทำงานเสร็จแล้ว ทาสก์ 3 จึงกลับมาทำงานในทรัพยากรต่อ(หมายเลข 9)

- เมื่อทาสก์ 3 ทำงานกับทรัพยากรเสร็จ (หมายเลข 10) จึงหยุดการทำงานกับเซมาฟอร์ (หมายเลข 11) ซึ่งในขณะนี้เคอร์เนลได้รู้ว่ามีทาสก์ที่มีความสำคัญสูงรอใช้งานเซมาฟอร์ดังนั้นจึงทำคอนเท็กซ์สวิตช์ให้ทาสก์ 1 กลับมาทำงาน ทาสก์ 1 จึงได้กลับมาทำงานและเข้าใช้งานทรัพยากร (หมายเลข 12) จนกระทั่งทาสก์ 1 ทำงานเสร็จ

จากเหตุการณ์ข้างต้นเป็นปัญหาไพรออริตี้อินเวอร์ชัน โดยการแก้ปัญหาไพรออริตี้อินเวอร์ชันที่เกิดขึ้นสามารถแก้ได้โดยการเปลี่ยนความสำคัญของทาสก์ 3 เมื่อได้เข้าไปใช้งานทรัพยากร และเปลี่ยนความสำคัญกลับเป็นค่าเดิมเมื่อทาสก์ทำงานเสร็จ ซึ่งมัลติทาสกิ้งเคอร์เนลสามารถที่จะเปลี่ยนความสำคัญได้เพื่อช่วยป้องกันการเกิดไพรออริตี้อินเวอร์ชัน แต่อย่างไรก็ตามการเปลี่ยนความสำคัญก็ต้องการใช้เวลาในการเปลี่ยน และถ้าทาสก์ 3 ทำงานเสร็จสิ้นก่อนที่จะเกิดทาสก์ 1 หรือทาสก์ 2 จะทำให้เกิดการเปลี่ยนเปลงค่าความสำคัญของทาสก์ที่สูญเปล่า โดยเคอร์เนลที่สามารถเปลี่ยนค่าความสำคัญได้อัตโนมัติ จะเรียกว่าไพรออริตี้อินเฮอริแตนส์ (priority inheritance)



รูปที่ 2.7เนลซึ่งรองรับไพรออริตี้อินเฮอริแตนส์ (kernel support priority inheritance)

จากภาพเคอร์เนลจะรองรับการทำไพรออริตี้อินเฮอริแตนส์

- เมื่อทาสก์ 3 ทำงานที่หมายเลข 1 และได้รับเซมาฟอร์เพื่อเข้าใช้งานทรัพยากร (หมายเลข 2) ทาสก์ 3 ก็จะเข้าไปใช้งานทรัพยากร (หมายเลข 3)

- ระหว่างนั้นทาสก์ 1 ก็ได้เริ่มทำงานขึ้น (หมายเลข 4และ5) ทาสก์ 1 พยายามที่จะใช้งานเซมาฟอร์ (หมายเลข 6) ซึ่ง เคอร์เนลจะเห็นว่าทาสก์ 3 กำลังใช้งานอยู่ แต่ว่าทาสก์ 3 มีความสำคัญที่ต่ำกว่าทาสก์ 1 ทำให้เคอร์เนลเปลี่ยนความสำคัญของทาสก์ 3 ให้สูงเท่ากับความสำคัญของทาสก์ 1 ดังนั้นเคอร์เนลจะให้ทาสก์ 3 ทำงานกับทรัพยากรต่อไปที่หมายเลข 7

- เมื่อทาสก์ 3 ทำงานกับทรัพยากรเสร็จแล้วก็จะเลิกใช้งานเซมาฟอร์ (หมายเลข 8) ตรงจุดนี้ เคอร์เนลจะทำการเปลี่ยนความสำคัญของทาสก์ 3 ให้กลับมาเป็นเหมือนเดิม และให้ทาสก์ 1 เข้าใช้งานเซมาฟอร์เพื่อทำงานต่อไป (หมายเลข 9)

- เมื่อทาสก์ 1 ทำงานเสร็จแล้ว (หมายเลข 10) ทาสก์ 2 ก็จะใช้งาน ซีพียูโดยที่ทาสก์ 2 พร้อมที่จะทำงานในช่วงระหว่างหมายเลข3ถึง10 แต่ไม่ได้ทำงาน ซึ่งตรงจุดนี้ยังเป็นไพรออริตี้อินเวอร์ชัน (ระหว่างทาสก์2กับทาสก์ 3)ที่หลีกเลี่ยงไม่ได้

การกำหนดความสำคัญให้แก่ทาสก์ก็เป็นสิ่งจำเป็น โดยที่เรียลไทม์ซิสเต็มจะต้องการทั้งซอฟต์เรียลไทม์และฮาร์ดเรียลไทม์ซิสเต็ม โดยที่ซอฟต์เรียลไทม์ซิสเต็มทาสก์จะทำงานให้เร็วที่สุดแต่อาจจะไม่เสร็จในเวลาที่กำหนด แต่ในฮาร์ดเรียลไทม์จะต้องเสร็จในเวลาที่กำหนด ด้วยเทคนิคที่เรียกว่า Rate Monotonic Scheduling หรือ RMS จะใช้ในการกำหนดความสำคัญของทาสก์โดยขึ้นกับความถี่ในการทำงาน โดยทาสก์ที่มักจะทำงานบ่อยจะกำหนดความสำคัญให้สูงสุด

2.4 ซิงโครไนเซชั่น(Synchronization)

2.4.1 ทรัพยากรและทรัพยากรที่ใช้ร่วมกัน(Resource and Share Resource)

ทรัพยากรหรือรีซอส (Resource) คือ สิ่งต่างๆที่ใช้งานโดยทาสก์ซึ่งทรัพยากรอาจจะเป็นพวกอุปกรณ์ I/O (I/O device) อย่างเช่น ปริ๊นเตอร์ , คีย์บอร์ด หรือพวกโครงสร้าง (structure) เป็นต้น โดยถ้าทรัพยากรสามารถใช้งานร่วมกันได้มากกว่า 1 ทาสก์จะเรียกว่า แชร์รีซอส (Share resource) ซึ่งแต่ละทาสก์จะต้องกำหนดการเข้าใช้งานไปยังแชร์รีซอสเพื่อป้องกันการที่ข้อมูลของแต่ละทาสก์ปนกันซึ่งจะทำให้ผลลัพธ์ผิดไปได้ ซึ่งเรียกว่า mutual exclusion

2.4.2 รีเอนทรานซี่(Reentrancy)

รีเอนทรานฟังก์ชั่น (reentrant function) คือ ฟังก์ชั่นที่สามารถใช้งานได้มากกว่า 1 ทาสก์โดยไม่ต้องกังวลว่าข้อมูลจะเกิดการเสียหาย โดยรีเอนทรานฟังก์ชั่นสามารถทำงานได้ทุกเวลาและย้อนกลับมาทีหลังได้โดยข้อมูลไม่สูญหาย รีเอนทรานฟังก์ชั่นจะใช้ทั้งตัวแปรโลคอล (local variable) อย่างเช่น ซีพียูรีจิสเตอร์ , ตัวแปรในสแตก หรืออาจจะใช้ในการป้องกันข้อมูลเมื่อมีการใช้งานตัวแปรโกลบอล (global variable)

```
void strcpy(char *dest, char *src)
{
  while (*dest++ = *src++) {
    ;
  }
  *dest = NUL;
}
```

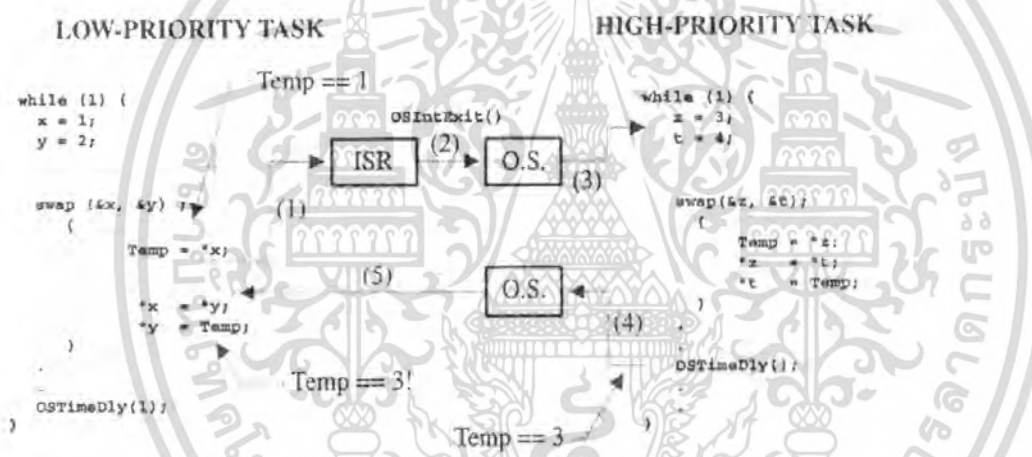
รูปที่ 2.8รีเอนทรานฟังก์ชั่น (reentrant function)

เนื่องจากการก็อปปี้ค่าอาร์กูเมนต์ไปยัง strcpy()จะใช้วิธีใส่ไว้ในทาสก์สแตกทำให้ strcpy() สามารถใช้งานได้หลาย ๆ ทาสก์โดยไม่ต้องกลัวว่าทาสก์จะเกิดการปนกันของพอยน์เตอร์ (pointer) ของทาสก์อื่น ๆ

```
int Temp;

void swap(int *x, int *y)
{
    Temp = *x;
    *x   = *y;
    *y   = Temp;
}
```

รูปที่ 2.9นอนรีเอนทรานฟังก์ชั่น (non-reentrant function)

ถ้าใช้งานพรีเอมทีฟเคอร์เนลและมีการเปิดใช้งานอินเตอร์รัพต์ เมื่อ swap() สามารถใช้งานได้ทุกทาสก์อาจจะเกิดเหตุการณ์ดังนี้



รูปที่ 2.10นอนรีเอนทรานฟังก์ชั่น (non reentrant function)

- ในขณะที่ทาสก์ที่ความสำคัญต่ำกว่าทำงานที่ swap() ได้มีการอินเตอร์รัพต์ที่หมายเลข 1 ISR จะทำงานที่ทาสก์ที่มีความสำคัญสูงกว่า

- เมื่อ ISR ทำงานเสร็จ (หมายเลข 2) เคอร์เนลจะทำการเปลี่ยนไปยังทาสก์ที่มีความสำคัญที่สูงกว่าทำงาน

- หลังจากทาสก์ทำงานเสร็จ ก็จะเรียกเคอร์เนลเพื่อทำการกลับไปทำทาสก์ที่มีความสำคัญต่ำกว่า (หมายเลขที่ 4) ซึ่งในขณะที่ทำงานอยู่ในทาสก์ที่ความสำคัญสูง จะมีการเปลี่ยนค่า Temp ซึ่งเป็นตัวแปรโก

ลบอล ทำให้เมื่อกลับมาทำงานในทาสก์ที่ความสำคัญต่ำ ค่า Temp จะมีการเปลี่ยนแปลง (หมายเลข 5) ซึ่งปัญหาที่เกิดขึ้นนี้สามารถแก้ไขได้โดยการ

- ประกาศ Temp ให้อยู่ภายใน swap()

- ปิดการใช้งาน อินเตอร์รัพต์ก่อนที่จะทำงาน และเปิดใช้งานหลังจากทำงานเสร็จ

- ใช้เซมาฟอร์ (semaphore)

2.4.3 มิวทอลเอ็กซ์คลูชั่น(Mutual Exclusion)

วิธีในการให้ทาสก์แต่ละตัวติดต่อกับทาสก์ตัวอื่น ๆ คือ ผ่านทางการใช้ โครงสร้างข้อมูล (data structure)อย่างเช่น ตัวแปรโกลบอล (glabal variable) , พอยท์เตอร์ (pointer) , บัฟเฟอร์(buffer) และอื่น อย่างไรก็ตามในการใช้ ข้อมูลร่วมกันเพื่อแลกเปลี่ยนข้อมูลจะต้องมั่นใจว่าแต่ละทาสก์ใช้งานข้อมูลแล้วไม่ทำให้ข้อมูลเสียหาย โดยวิธีการในการป้องกันในการใช้งานทรัพยากรร่วมกันจะมีดังนี้

2.4.3.1 การเปิดและปิดอินเตอร์รัพท์

วิธีที่ง่ายและรวดเร็วที่จะป้องกันการเข้าใช้งานทรัพยากรของทาสก์อื่นคือการดิสเอเบิล (disable) การใช้งาน อินเตอร์รัพต์หลังจากนั้นค่อยทำการ เอนาเบิล (enable) เมื่อทำงานนั้นเสร็จแล้ว แต่ในการใช้การดิสเอเบิลการใช้อินเตอร์รัพต์นั้นจะต้องระวังเพราะถ้ายกเลิกใช้งานนานเกินไป จะมีผลต่อระบบในการตอบสนองอินเตอร์รัพต์ ซึ่งก็คือ interrupt latency โดยจะต้องพิจารณาตรงจุดนี้ เมื่อต้องการจะเปลี่ยนหรือคัดลอกตัวแปร (variable) บางตัว แต่อย่างไรก็ตามวิธีนี้เป็นทางหนึ่งในการที่ทาสก์จะสามารถใช้งานตัวแปรหรือโครงสร้างข้อมูลร่วมกับ ISR ซึ่งกรณีนี้ จะต้องดิสเอเบิลการใช้งาน อินเตอร์รัพต์ให้มีช่วงเวลาสั้นที่สุด

```
Disable interrupts;
Access the resource (read/write from/to variables);
Reenable interrupts;
```

รูปที่ 2.11การดิสเอเบิลและเอนาเบิลอินเตอร์รัพต์ (disable and enable interrupt)

ถ้าใช้งานเคอร์เนลจะสามารถดิสเอเบิลการใช้งานอินเตอร์รัพต์ได้นานเท่ากับที่ไม่ทำให้เกิดปัญหากับ interrupt latency ซึ่งจำเป็นจะต้องรู้ว่าควรดิสเอเบิลการใช้งานอินเตอร์รัพต์ได้นานเท่าไร

2.4.3.2 เทสท์แอนด์เซ็ต(Test-and-Set)

ถ้าไม่ได้ใช้งานเคอร์เนลก็ควรจะใช้การกำหนดค่าตัวแปรโกลบอล ไว้เพื่อใช้ตรวจสอบให้รู้ว่าทรัพยากรมีการใช้งานอยู่หรือไม่ เช่น ถ้ามีการใช้งานทรัพยากรในขณะนั้น ให้กำหนดค่าของตัวแปรโกลบอลเป็น 1 และถ้าไม่มีการใช้งาน ให้กำหนดค่าเป็น 0 ซึ่งวิธีการนี้เรียกว่า Test-And-Set หรือ TAS

```
Disable interrupts;
if ('Access Variable' is 0) {
    Set variable to 1;
    Reenable interrupts;
    Access the resource;
    Disable interrupts;
    Set the 'Access Variable' back to 0;
    Reenable interrupts;
} else {
    Reenable interrupts;
    /* You don't have access to the resource, try back later; */
}
```

รูปที่ 2.12 test and set

2.4.3.3 ปิดและเปิดตัวจัดลำดับงาน

ถ้าทาสก์ไม่ได้ใช้งานตัวแปรหรือโครงสร้างข้อมูลร่วมกับ ISR จะสามารถดิสเอเบิลและเอนาเบิลการจัดลำดับได้ ทำให้ทาสก์ 2 ทาสก์หรือมากกว่านี้สามารถใช้ข้อมูลร่วมกันได้โดยไม่ต้องแย่งกัน ซึ่งในขณะที่การจัดลำดับถูกยกเลิก และอินเตอร์รัพต์เปิดใช้งาน เมื่อมีอินเตอร์รัพต์เกิดขึ้นมา ISR จะทำงานโดยทันที หลังจาก ISR ทำงานเสร็จเคอร์เนลจะกลับไปยัง อินเตอร์รัพต์ทาสก์ (interrupt task) หรือทาสก์ที่กำหนดโดย ISR ซึ่งเตรียมพร้อมจะทำงาน และการจัดลำดับจะกลับมาใช้งานอีกครั้งเพื่อทำให้ทาสก์ได้ทำงาน ซึ่งการทำอย่างนี้ใช้งานได้ดี แต่จะเป็นการทำลายจุดประสงค์ของการใช้งานเคอร์เนล

```
void Function (void)
{
    OSSchedLock();
    .
    .    /* You can access shared data in here (interrupts are recognized) */
    .
    OSSchedUnlock();
}
```

รูปที่ 2.13การดิสเอเบิลและเอนาเบิลการจัดลำดับ (disable and enable scheduler)

2.4.3.4 เซมาฟอร์(Semaphore)

เซมาฟอร์ (Semaphore) ถูกคิดขึ้นมาโดย Edgser Dijkstra ซึ่งจะใช้ในมัลติทาสกิ้งเคอร์เนลโดยจะใช้เพื่อ

- ควบคุมการเข้าใช้งานทรัพยากรที่มีการใช้งานร่วมกัน (mutual exclusion)

- ส่งสัญญาณเมื่อมีการเกิดอีเวนต์ขึ้น

- อนุญาติให้ทาสก์ 2 ตัวสามารถซิงโครไนซ์ (synchronize) การทำงานได้

เซมาฟอร์เป็นเสมือนคีย์ (key) ที่จะให้การทำงานสามารถทำต่อไปได้ ถ้าเซมาฟอร์ถูกใช้งานอยู่ ทาสก์ที่ร้องขอการใช้งานจะถูกหยุดการทำงานชั่วคราวและรอไปจนกว่าเซมาฟอร์จะถูกเลิกใช้งานโดยทาสก์ที่ใช้งานอยู่ ซึ่งเซมาฟอร์แบ่งออกเป็น 2 ประเภทคือ

- ไบนารี่เซมาฟอร์ (binary semaphore) ซึ่งค่าจะมีได้แค่ 0 กับ 1 และ

- เคาท์ติงเซมาฟอร์ (counting semaphore) ซึ่งค่าที่เป็นไปได้คือ 0 กับ 255,65535 หรือ 4292967295 ซึ่งขึ้นกับว่ากระบวนการเซมาฟอร์ที่ใช้งาน ใช้ 8 , 16 หรือ 32 บิต ตามลำดับตามขนาดของเคอร์เนล โดยระหว่างที่ทาสก์กำลังรอการใช้งาน เซมาฟอร์นั้นเคอร์เนลจะเก็บการทำงานของทาสก์ที่รอเอาไว้ด้วย

โดยทั่วไปแล้วจะมีการทำงานอยู่ 3 อย่างที่ดำเนินการบนเซมาฟอร์คือ INITIALIZE หรือเรียกกว่า CREATE , WAIT หรือ PEND และ SIGNAL หรือ POST ค่าเริ่มต้นของเซมาฟอร์จะถูกกำหนดเมื่อเซมาฟอร์เริ่มทำงาน และงานใน Waiting List จะว่าง

เมื่องานต้องการใช้งานเซมาฟอร์จะเข้าสู่ WAIT ก่อน ถ้าเซมาฟอร์ว่างค่าของเซมาฟอร์จะมีค่ามากกว่า 0 และเมื่อ เซมาฟอร์ถูกเข้าใช้งานโดยทาสก์ ค่าของเซมาฟอร์จะถูกลดลง ถ้าค่าของเซมาฟอร์เป็น 0 ทาสก์ที่ต้องการเข้ามาใช้งานจะเข้าไปอยู่ใน Waiting List และเคอร์เนลสามารถกำหนดไทม์เอาท์ (timeout) ได้ ถ้าเซมาฟอร์ยังไม่ว่างในเวลาที่กำหนดไว้โดยทาสก์ที่ร้องขอไป ทาสก์นั้น ๆ จะเตรียมพร้อมทำงานและโค้ดแสดงความผิดพลาด (error code) ซึ่งแสดงให้เห็นว่าเกิดไทม์เอาท์จะถูกส่งกลับไปยังผู้เรียกใช้งาน

ทาสก์จะเลิกการใช้งานเซมาฟอร์โดยการทำงาน SIGNAL ถ้าไม่มีทาสก์รอใช้งาน ค่าของเซมาฟอร์จะถูกเพิ่มค่าขึ้น แต่ถ้ามีทาสก์รอการใช้งานเซมาฟอร์ โดยที่ได้มีทาสก์ตัวหนึ่งพร้อมที่จะทำงานแล้วแต่ค่า

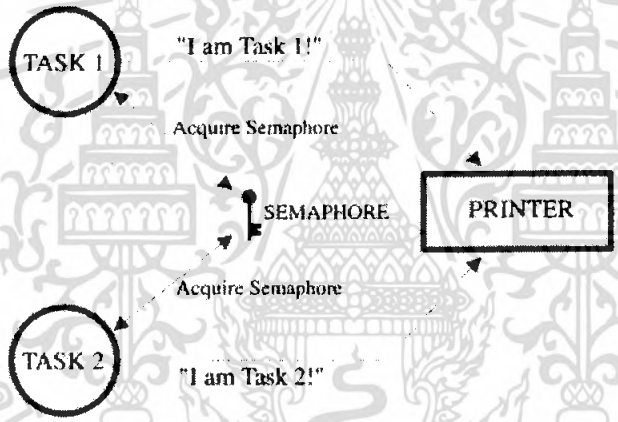
ของเซมาฟอร์ยังไม่ได้เพิ่มค่า คีย์ (key) จะถูกกำหนดให้ทาสก์ที่รอการใช้งาน ซึ่งขึ้นอยู่กับเคอร์เนลว่าจะให้ทาสก์ตัวไหนได้รับเซมาฟอร์โดยขึ้นกับ

- ทาสก์ที่มีความสำคัญสูงสุด ที่รอการใช้งาน เซมาฟอร์และ

- ทาสก์ที่มาร้องขอเซมาฟอร์ลำดับแรก ๆ หรือ First In First Out (FIFO)

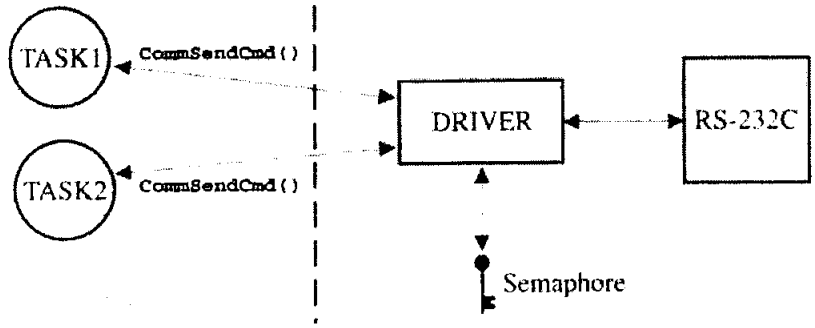
โดยเคอร์เนลบางตัวสามารถกำหนดได้เมื่อเซมาฟอร์เริ่มต้นทำงาน

เซมาฟอร์มีประโยชน์เมื่อทาสก์ใช้งาน อุปกรณ์ I/O ร่วมกัน อย่างเช่น ถ้าไม่มีการใช้งานเซมาฟอร์ กรณีที่มีทาสก์ 2 ตัวเข้าใช้งานปริ๊นเตอร์พร้อมกัน ทำให้เมื่อปริ๊นออกมาแล้ว ข้อความที่ได้ออกมาอาจจะปนกันก็ได้ ซึ่งในกรณีนี้ เมื่อมีการใช้งานเซมาฟอร์ โดยกำหนดค่าเริ่มต้นเป็น 1 (ไบนารีเซมาฟอร์) กฎในการเข้าใช้ปริ๊นเตอร์ คือ ต้องได้รับเซมาฟอร์ของทรัพยากรก่อนที่จะเข้าใช้งาน



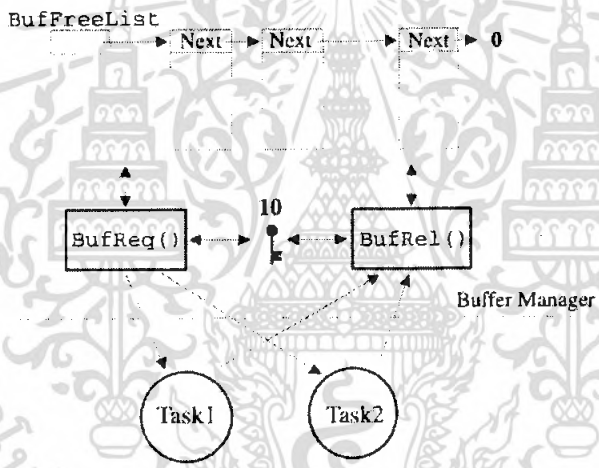
รูปที่ 2.14การเข้าใช้ทรัพยากรโดยการใช้งานเซมาฟอร์

จากภาพจะแสดงเซมาฟอร์ในรูปของกุญแจ ซึ่งจะใช้ในการเข้าใช้งานทรัพยากรทำให้รู้ลำดับการเข้าใช้งานทรัพยากร โดยดูจาก เซมาฟอร์ซึ่งจะดีกว่าถ้าเซมาฟอร์ได้ถูกเอนแคปซูลเลต (encapsulate) และทำให้แต่ละทาสก์ไม่รู้ว่าได้รับเซมาฟอร์เมื่อเข้าใช้งานทรัพยากร ตัวอย่างเช่น อุปกรณ์หรือดีไวซ์ (device) เมื่อทาสก์ต้องการส่งคำสั่งไปยังอุปกรณ์ ก็จะทำการเรียกฟังก์ชั่น โดยฟังก์ชั่นจะเรียกใช้งานเซมาฟอร์ต่ออีกที



รูปที่ 2.15การซ่อนเซมาฟอร์จากทาสก์

เริ่มต้นทาสก์ 1 ติดต่อเข้ามาโดยจะเรียกผ่านฟังก์ชั่นเพื่อจะได้เซมาฟอร์มา จากนั้นอุปกรณ์จะทำงานตามการทำงานของทาสก์ ถ้ามีทาสก์อื่นได้ส่งคำสั่งเพื่อจะใช้งาน แต่พบว่าพอร์ต (port) กำลังถูกใช้งานอยู่ ทาสก์ที่เรียกมาทีหลังจะถูกหยุดการทำงานชั่วคราวจนกว่าเซมาฟอร์จะว่าง



รูปที่ 2.16เคาท์ติง เซมาฟอร์ (counting semaphore)

เคาท์ติงเซมาฟอร์ (counting semaphore) จะใช้เมื่อทรัพยากรสามารถใช้งานได้มากกว่า 1 ทาสก์ขึ้นไปในเวลาเดียวกัน อย่างเช่นการจัดการของบัฟเฟอร์พูล (buffer pool) สมมุติให้เริ่มต้นบัฟเฟอร์พูลมีจำนวน 10 บัฟเฟอร์โดยทาสก์สามารถใช้งานบัฟเฟอร์ได้โดยการเรียกใช้ BufReq() และถ้าบัฟเฟอร์ไม่ได้ใช้งานต่อก็จะคืนด้วยการใช้ BufRel()

```
BUF *BufReq(void)
{
  BUF *ptr;

  Acquire a semaphore;
  Disable interrupts;
  ptr         = BufFreeList;
  BufFreeList = ptr->BufNext;
  Enable interrupts;
  return (ptr);
}
```

```
void BufRel(BUF *ptr)
{
   Disable interrupts;
   ptr->BufNext = BufFreeList;
   BufFreeList  = ptr;
   Enable interrupts;
   Release semaphore;
}
```

รูปที่ 2.17การจัดการบัฟเฟอร์โดยการใช้งานเซมาฟอร์

ในการใช้งานบัฟเฟอร์ (ในที่นี้มี 10 บัฟเฟอร์) จะสามารถรองรับการร้องขอได้ 10 อย่างเพราะว่าเสมือนกับมี 10 คีย์ เมื่อเซมาฟอร์ถูกใช้งานทั้งหมด (บัฟเฟอร์เต็ม) และมีทาสก์ต้องการใช้งาน ทาสก์ตัวนั้นจะถูกหยุดการทำงานชั่วคราวจนกว่าเซมาฟอร์จะว่าง เมื่อทาสก์ทำงานเสร็จสิ้น จะทำการคืนบัฟเฟอร์โดยการเรียก BufRel() ซึ่งบัฟเฟอร์จะทำการแทรกเข้าไปในลิงค์ลิส (link list) ของบัฟเฟอร์ก่อนจะทำการคืนเซมาฟอร์

โดยทั่วไปแล้วมักจะใช้งานเซมาฟอร์ แต่ในงานบางอย่าง เช่น การเข้าใช้งานตัวแปรที่ใช้งานร่วมกัน (share variable)ทั่ว ๆ ไปมักจะใช้การการดิสเอเบิลและเอนาเบิลอินเตอร์รัพต์เพราะการใช้เซมาฟอร์จะเสียเวลากว่า

2.4.4 เดตล็อก(Deadlock)

เดตล็อก (Deadlock) หรือ deadly embrace คือกรณีที่ทาสก์ 2 ตัวต่างรอการใช้งานทรัพยากรที่ใช้งานโดยอีกตัว เช่น ทาสก์ T1 ใช้งานทรัพยากร R1 และทาสก์ T2 ใช้งานทาสก์ T2 ถ้า T1 ต้องการใช้งาน R2 และ T2 ต้องการใช้ R1 จะเกิดเดตล็อกขึ้น ซึ่งการแก้ปัญหาทำได้โดย

- ก่อนทำงานต้องได้รับทรัพยากรก่อน

- ให้การใช้งานทรัพยากรตามลำดับ

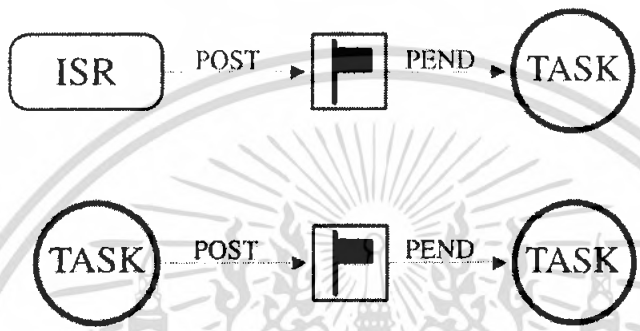
- ในการเลิกใช้งานทรัพยากรต้องเลิกใช้งานแบบย้อนลำดับ

เคอร์เนลทั่วไปจะอนุญาติให้กำหนดไทม์เอาท์ (timeout) ได้เมื่อใช้งานเซมาฟอร์ซึ่งจะช่วยในการป้องกันการเกิด เดตล็อก ถ้าทาสก์ที่ต้องการใช้งานเซมาฟอร์ไม่ได้รับในเวลาที่ต้องการ ทาสก์นั้นจะกลับไป

ทำงานต่อ และโค้ดแสดงความผิดพลาด (error code) จะถูกส่งไปเพื่อไปแจ้งว่าเกิดไทม์เอาท์ขึ้น ซึ่งโค้ดแสดงความผิดพลาดตรงนี้จะช่วยป้องกันที่การที่ทาสก์คิดว่าได้รับทรัพยากรนั้น โดยเดตล็อกมักจะเกิดกับระบบมัลติทาสกิ้งที่มีขนาดใหญ่ ไม่ค่อยจะปรากฏในระบบสมองกลฝังตัว (embedded system)

2.4.5 ซิงโครไนเซชั่น(Synchronization)

ทาสก์สามารถซิงโครไนซ์กับ ISR หรือกับทาสก์ที่ไม่มีการเปลี่ยนแปลงข้อมูลได้ โดยการใช้งานเซมาฟอร์



รูปที่ 2.18การซิงโครไนซ์ของทาสก์และ ISR

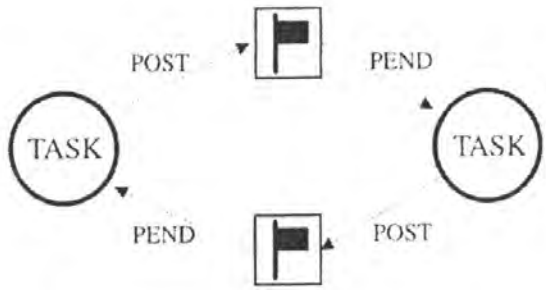
ในกรณีนี้เซมาฟอร์จะถูกแสดงในรูปธง (flag) เพื่อแสดงว่ามันได้รับสัญญาณจากอีเวนท์ โดยเมื่อใช้กระบวนการซิงโครไนซ์เซชัน (synchronization) เซมาฟอร์จะถูกกำหนดค่าเริ่มต้นให้เป็น 0 ซึ่งการใช้เซมาฟอร์แบบนี้จะเรียกว่า unilateral rendezvous โดยทาสก์จะกำหนดค่าเริ่มต้นให้กับการทำงานของ I/O และรอเข้าใช้งานเซมาฟอร์ เมื่อการทำงานของ I/O เสร็จสิ้น ISR หรือทาสก์อื่นจะส่งสัญญาณไปยังเซมาฟอร์และทาสก์ที่รอใช้งานเซมาฟอร์จะได้เข้ามาใช้งาน

ถ้าเคอร์เนลรองรับการทำงานของเคาท์ติงเซมาฟอร์ เซมาฟอร์จะทำการเก็บเหตุการณ์ต่างๆที่ยังไม่ได้จัดการเอาไว้ ซึ่งถ้ามีมากกว่า 1 ทาสก์ที่รอการทำงาน เคอร์เนลจะต้องจัดการให้ดังนี้

- ทาสก์ที่มีความสำคัญสูงสุดที่รอ ได้ทำงาน และ

- ทาสก์ที่มารอก่อนได้ทำงาน

ซึ่งก็ขึ้นอยู่กับแอพพลิเคชั่นซึ่งเป็นไปได้ว่าจะมีทาสก์หรือ ISR มากกว่า 1 เหตุการณ์ที่เกิดขึ้นส่งสัญญาณไป ทาสก์ 2 ตัวสามารถซิงโครไนซ์งานของมันได้โดยการใช้ 2 เซมาฟอร์ซึ่งเป็นดังนี้



รูปที่ 2.19ทาสก์ซิงโครไนซ์กิจกรรมร่วมกัน

โดยเรียกว่า bilateral rendevous ซึ่งคล้ายกับ unilateral rendezvous เว้นแต่ว่าทั้ง 2 ทาสก์จะต้องซิงโครไนซ์กับอีกทาสก์หนึ่งก่อนจะทำงานต่อไป

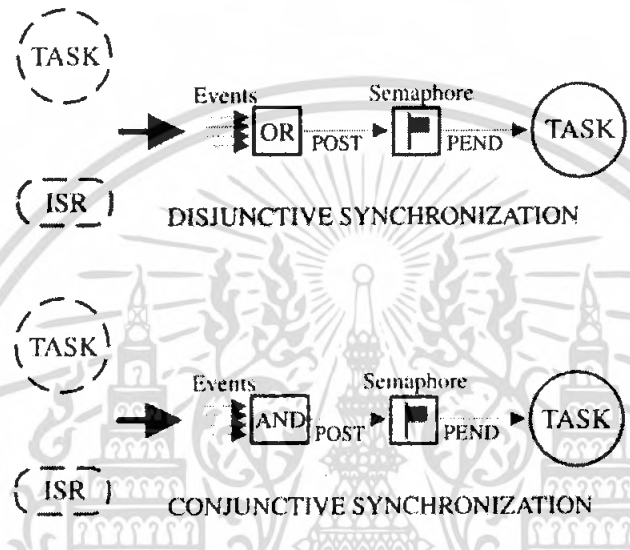
```
Task1()
{
    for (;;) {
        Perform operation;
        Signal task #2;                    (1)
        Wait for signal from task #2;      (2)
        Continue operation;
    }
}

Task2()
{
    for (;;) {
        Perform operation;
        Signal task #1;                    (3)
        Wait for signal from task #1;      (4)
        Continue operation;
    }
}
```

รูปที่ 2.20 code bilateral rendezvous

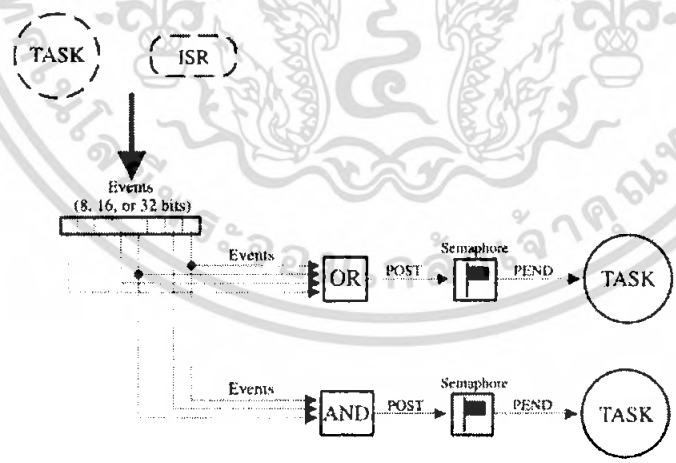
มีทาสก์ 2 ตัวทำงาน เมื่อทาสก์ตัวแรกทำงานไปถึงจุด ๆ หนึ่งก็จะส่งสัญญาณไปหาเซมาฟอร์เพื่อให้ทาสก์ 2 ได้ใช้งาน และจะหยุดรอให้ทาสก์ 2 ส่งสัญญาณกลับมา และในลักษณะเดียวกันทาสก์ 2 ทำงานไปถึงจุด ๆ หนึ่งก็จะส่งสัญญาณไปเพื่อให้ ทาสก์ 1 ได้ทำงานและรอสัญญาณตอบกลับมา ซึ่งในลักษณะนี้สแตกทั้งสองจะมีการซิงโครไนซ์กัน แต่ ISR ไม่สามารถทำงานในลักษณะ bilateral rendevous ได้ เพราะ ISR ไม่สามารถหยุดรอเซมาฟอร์

2.4.6 อีเวนท์แฟลก(Event Flags)

อีเวนท์เฟลก (Event Flag) จะใช้เมื่อทาสก์ต้องการซิงโครไนซ์กับอีเวนท์ที่เกิดขึ้นหลาย ๆ เหตุการณ์ โดยทาสก์สามารถ ซิงโครไนซ์เมื่อมีอีเวนท์ใดอีเวนท์หนึ่งจากทั้งหมดเกิดขึ้น ซึ่งจะเรียกว่า ดิสจังค์ทีฟ ซิงโครไนซ์เซชั่น (disjunctive synchronization) ซึ่งเป็นลอจิก OR และทาสก์สามารถซิงโครไนซ์เมื่อทุกเหตุการณ์ได้เกิดขึ้นซึ่งเรียกว่าคอนจังค์ทีฟ ซิงโครไนซ์เซชั่น (conjunctive synchronization) ซึ่งเป็นลอจิก AND ซึ่ง ดิสจังค์ทีฟและคอนจังค์ทีฟ ซิงโครไนซ์เซชั่น แสดงได้ดังนี้



รูปที่ 2.21ดิสจังค์ทีฟและคอนจังค์ทีฟ ซิงโครไนซ์เซชั่น



รูปที่ 2.22อีเวนท์เฟลก

2.4.7 การสื่อสารระหว่างทาสก์

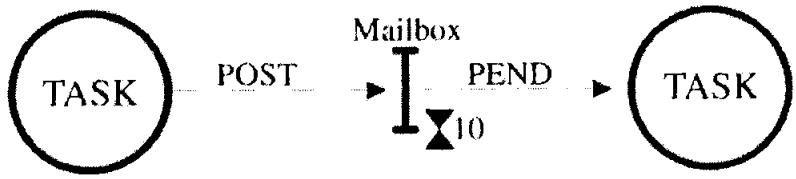
ในบางครั้งทาสก์หรือ ISR จำเป็นที่จะต้องสื่อสารข้อมูลข่าวสาร (informaiton) กับทาสก์อื่น ๆ ซึ่งการแลกเปลี่ยนข้อมูลข่าวสารเรียกว่าอินเตอร์ทาสก์คอมมูนิเคชัน (intertask communication) โดยข่าวสารสามารถสื่อสารระหว่างทาสก์ได้ 2 วิธีคือ ผ่านข้อมูลที่เป็นโกลบอล (global data) หรือโดยการส่งข้อความหรือแมสเสจ (message)

เมื่อใช้งานตัวแปรโกลบอลแต่ละทาสก์หรือ ISR จะต้องมั่นใจว่ามีผู้ใช้งานตัวแปรนั้นเพียงผู้เดียว โดยถ้าเป็น ISR วิธีที่จะทำให้มั่นใจคือใช้การดิสเอเบิลอินเตอร์รัพต์ (disable interrupt) ถ้าเป็นทาสก์ 2 ตัวใช้งานข้อมูลร่วมกัน สามารถใช้งานได้โดย ดิสเอเบิลและเอนาเบิลอินเตอร์รัพต์หรือใช้เซมาฟอร์ แต่ทาสก์สามารถสื่อสารกับ ISR ได้ทางเดียวคือใช้ตัวแปรโกลบอล ซึ่งทาสก์จะไม่รู้ว่าค่าของตัวแปรโกลบอลเปลี่ยนโดย ISR หรือไม่ เว้นแต่ว่า ISR จะส่งสัญญาณไปยังทาสก์ โดยใช้เซมาฟอร์หรือ เว้นแต่ว่าทาสก์จะคอยตรวจสอบค่าของตัวแปรเรื่อย ๆ ซึ่งวิธีแก้ไขในกรณีนี้สามารถใช้ แมสเสจเมลบ็อกซ์ (message mailbox) หรือแมสเสจคิว (message queue)ได้

2.4.7.1 แมสเสจ เมล์บอกส์(Message Mailbox)

แมสเสจ (message) จะสามารถส่งไปยังทาสก์โดยผ่านบริการของเคอร์เนลได้ โดยการแลกเปลี่ยนแมสเสจนี้จะเรียกว่า แมสเสจเมลบ็อกซ์ (Message Mailbox) ซึ่งมักจะเป็นตัวแปรใช้เก็บพอยท์เตอร์ (pointer-size variable) การใช้บริการผ่านเคอร์เนลนั้นทาสก์หรือ ISR สามารถสะสมแมสเสจ (พอยต์เตอร์) เข้าไปในเมลบ็อกซ์โดย 1 หรือหลาย ๆ ทาสก์สามารถรับแมสเสจได้ผ่านทางบริการของเคอร์เนล ซึ่งทั้งทาสก์ฝั่งส่งและทาสก์ฝั่งรับจะมีพอยท์เตอร์ชี้ไปที่เดียวกัน

Waiting List จะเกี่ยวข้องกับแต่ละเมลบ็อกซ์ ในกรณีที่มีทาสก์มากกว่า 1 ต้องการแมสเสจผ่านทางเมลบ็อกซ์ ทาสก์ที่ต้องการแมสเสจจากเมลบ็อกซ์ที่ว่างจะถูกหยุดการทำงานชั่วคราวและจะถูกใส่เข้าไปยัง Waiting List จนกว่าจะได้รับแมสเสจซึ่งทั่วไปแล้วเคอร์เนลจะมีไทม์เอาท์สำหรับทาสก์ที่รอแมสเสจ ถ้าทาสก์ไม่ได้รับแมสเสจก่อนเวลาไทม์เอาท์ที่กำหนด ทาสก์ตัวนั้นจะพร้อมที่จะทำงานและจะส่งโค้ดแสดงความผิดพลาด (error code) กลับไป เมื่อแมสเสจได้ถูกใส่เข้าไปในเมลบ็อกซ์แล้วทาสก์ที่จะได้รับแมสเสจคือ ทาสก์ที่มีค่าความสำคัญสูงที่สุด(คิดความสำคัญเป็นหลัก) หรือทาสก์ตัวแรกที่ร้องขอแมสเสจ (first-in-first-out)



รูปที่ 2.23แมสเสจเมลบ็อก

จากภาพแสดงการนำแมสเสจใส่เข้าไปในเมลบ็อกซ์ ซึ่งแสดงแทนเป็นไอ-บีม (I-beam) และไทม์เอาท์แสดงเป็น นาฬิกาทราย (hourglass) ตัวเลขเวลาที่อยู่ถัดไปจากนาฬิกาทราย คือจำนวนของคล๊อกทริก (clock tick) เคอร์เนลจะจัดการการให้บริการเมลบ็อกซ์ ดังนี้

- ค่าเริ่มต้นของเมลบ็อกซ์ซึ่งเริ่มต้นนั้นอาจจะมีแมสเสจหรือไม่ก็ได้

- ใส่แมสเสจเข้าไปใน เมลบ็อกซ์ (POST)

- รอแมสเสจที่จะใส่เข้ามาใน เมลบ็อกซ์ (PEND)

- รับแมสเสจจากเมลบ็อกซ์เมื่อมีแมสเสจ (ACCEPT) ถ้าเมลบ็อกซ์มีแมสเสจก็จะส่งจากเมลบ็อกซ์ไป และรีเทิร์นโค้ด (return code) จะใช้ในการแจ้งเตือนตัวเรียกใช้งานเกี่ยวกับผลลัพธ์ของการเรียกใช้งาน

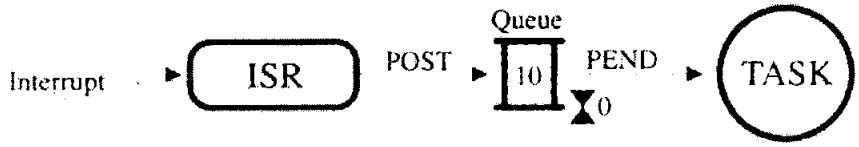
แมสเสจเมลบ็อกสามารถทำหน้าที่คล้ายไบนารีเซมาฟอร์ (binary semaphore) โดยมีแมสเสจในเมลบ็อกซ์จะแสดงว่าสามารถใช้งานทรัพยากรได้และ ถ้าเมลบ็อกซ์ว่าง แสดงว่าทรัพยากรกำลังถูกใช้งานโดยทาสก์ตัวอื่น

2.4.7.2 แมสเสจคิว(Message Queue)

แมสเสจคิว (message queue) จะใช้ในการส่งแมสเสจโดยสามารถส่งได้มากกว่า 1 แมสเสจไปยังทาสก์ซึ่งแมสเสจคิวมักจะเป็นอาเรย์ (array) ของเมล์บ็อกซ์ โดยทาสก์หรือ ISR สามารถใส่แมสเสจ (พอยท์เตอร์) เข้าไปในแมสเสจคิวโดยผ่านบริการของเคอร์เนลและเช่นเดียวกันทาสก์มากกว่า 1 สามารถรับแมสเสจผ่านบริการของเคอร์เนลได้ ซึ่งทั้งทาสก์ฝั่งส่งและฝั่งรับจะมีพอยท์เตอร์ชี้ไปที่เดียวกัน ซึ่งทั่วไปแล้วแมสเสจ แรกที่ใส่เข้าไปจะเป็นแมสเสจแรกที่ออกมา (FIFO)

การทำงานคล้ายกับเมลบ็อกซ์ซึ่ง Waiting List จะเกี่ยวข้องกับแต่ละแมสเสจคิวในกรณีที่มีทาสก์มากกว่า 1 ต้องการรับแมสเสจจากคิว ทาสก์ที่ต้องการแมสเสจจากคิวที่ว่างจะหยุดการทำงานชั่วคราวและใส่เข้าไปใน Waiting List จนกว่าจะได้รับแมสเสจซึ่งเคอร์เนลจะมีการกำหนดไทม์เอาท์สำหรับทาสก์ที่รอ

แมสเสจ ถ้าทาสก์ไม่ได้รับแมสเสจก่อนที่จะหมดเวลา ทาสก์นั้นจะพร้อมทำงานและจะส่งโค้ดแสดงความผิดพลาด (error code) กลับไป เมื่อแมสเสจใส่เข้าไปในคิว จะมีทั้งทาสก์ที่มีค่าความสำคัญสูงสุดหรือทาสก์ที่รอเป็นลำดับแรก มารอรับแมสเสจ



รูปที่ 2.24แมสเสจคิว

แสดง ISR ซึ่งใส่แมสเสจเข้าไปในคิว ซึ่งคิวจะแสดงโดยภาพไอ-บีม (I-beam) สองอัน เลข 10 แสดงจำนวน แมสเสจที่สามารถสะสมอยู่ในคิวได้ และ เลข 0 แสดงว่าทาสก์สามารถรอแมสเสจได้ตลอดจนกว่าแมสเสจจะมาถึง เคอร์เนลจะจัดการบริการแมสเสจคิวได้ดังนี้

- เริ่มต้นคิวจะถูกกำหนดให้ว่าง

- การใส่แมสเสจเข้าไปในคิว (POST)

- การรอแมสเสจที่ถูกใส่เข้าไปในคิว (PEND)

- รับแมสเสจถ้าอยู่ในคิว (ACCEPT) ถ้าคิวมีแมสเสจอยู่ จะถูกนำออกจากคิวและรีเทิร์นโค้ด (return code) จะใช้ในการแจ้งเตือนตัวเรียกใช้งาน เกี่ยวกับผลลัพธ์ของการเรียกใช้งาน

2.5 อินเตอร์รัพต์(Interrupt)

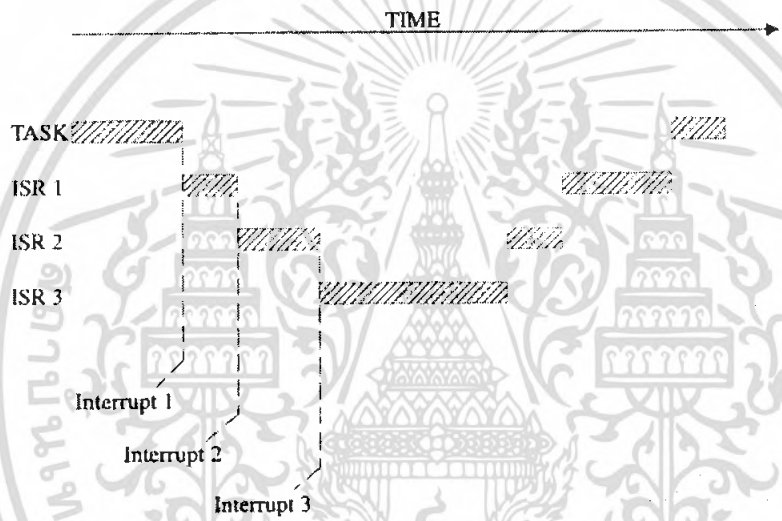
อินเตอร์รัพต์ (interrupt) เป็นกระบวนการทางฮาร์ดแวร์ (Hardware) ซึ่งใช้ในการแจ้งให้ซีพียูทราบว่าได้มีอซิงโครนัส อีเวนท์ (asynchronous event) เกิดขึ้น เมื่อแยกได้ว่าเกิดอะไรขึ้น ซีพียูจะทำการเก็บค่าคอนเท็กซ์ (context) บางส่วนหรือทั้งหมดเอาไว้และจะไปทำในกระบวนการย่อยที่เรียกว่า อินเตอร์รัพต์เซอวิสรูทีน (Intterrupt service routine) หรือ ISR โดยเมื่อทำ ISR เสร็จแล้วจะกลับไปยัง

- แบ็กกราวด์ (background) สำหรับ ระบบฟอร์กราวด์/แบ็กกราวด์ (foreground/background system)

- ทาสก์ที่เกิดอินเตอร์รัพต์สำหรับนอนพรีเอมทีฟเคอร์เนล (non-preemptive kernel)

- ทาสก์ที่มีความสำคัญสูงที่สุด ซึ่งพร้อมที่จะทำงานในพรีเอมทีฟเคอร์เนล (preemtive kernel)

อินเตอร์รัพต์จะช่วยให้ไมโครโปรเซสเซอร์จัดการกับเหตุการณ์ที่เกิดขึ้น โดยไมโครโปรเซสเซอร์สามารถจะไม่สน อินเตอร์รัพต์ที่เกิดขึ้นได้โดยการใช้คำสั่งดิสเอเบิลอินเตอร์รัพต์และเอนาเบิลอินเตอร์รัพต์ ในระบบเรียลไทม์ อินเตอร์รัพต์จะต้องดิสเอเบิลให้น้อยและระยะเวลาสั้นที่สุด ซึ่งการดิสเอเบิลอินเตอร์รัพต์จะมีผลต่อ interrupt latency และอาจจะทำให้อินเตอร์รัพต์ทำงานผิดพลาดได้ โปรเซสเซอร์สามารถจะทำอินเตอร์รัพต์แบบซ้อนกันได้ (interrupt nest) ซึ่งหมายความว่าขณะที่กำลังทำงานในอินเตอร์รัพต์นั้นโปรเซสเซอร์สามารถจะให้บริการกับอินเตอร์รัพต์ตัวอื่นที่สำคัญกว่าได้



รูปที่ 2.25อินเตอร์รัพต์แบบซ้อน

อินเตอร์รัพต์สามารถแบ่งออกได้เป็น 2 ชนิด คือ

- มาคส์เอเบิ้ลอินเตอร์รัพต์ (Maskable Interrupt) การอินเตอร์รัพท์ที่ซีพียูสามารถปฏิเสธได้ ซึ่งขึ้นกับว่าได้เปิดใช้งานอินเตอร์รัพต์หรือไม่

- นอนมาคส์เอเบิ้ลอินเตอร์รัพต์ (Non-Maskable Interrupt) การอินเตอร์รัพต์ที่ซีพียูไม่สามารถปฏิเสธ ซึ่งเมื่อมี อินเตอร์รัพต์ประเภทนี้เข้ามา ซีพียูจะต้องเข้ามาทำงานทันที

2.5.1 ความล่าช้าจากอินเตอร์รัพต์(Interrupt Latency)

สิ่งสำคัญที่สุดของเรียลไทม์เคอร์เนล คือ เวลาทั้งหมดที่อินเตอร์รัพต์ถูกดิสเอเบิลซึ่งเรียลไทม์ซิสเต็มจะดิสเอเบิลอินเตอร์รัพต์กับคริติคอลโค้ด (critical code) และจะเปิดใช้งานอีกครั้งเมื่อคริติคอลโค้ดได้ทำงานเสร็จแล้ว โดยยิ่งดิสเอเบิลอินเตอร์รัพต์นานเท่าไรก็จะยิ่งมี Interrupt latency มากยิ่งขึ้น

2.5.2 การตอบสนองต่ออินเตอร์รัพต์(Interrupt Response)

การตอบสนองของอินเตอร์รัพต์ (interrupt response) คือ ช่วงเวลาระหว่างที่รับอินเตอร์รัพต์เข้ามาและเริ่มทำงานในส่วนโค้ดของอินเตอร์รัพต์ ซึ่งซีพียูจะทำการเก็บค่าของคอนเท็กซ์ (context) ก่อนที่จะทำงานใน โค้ดของ ISR

- สำหรับระบบฟอร์กราวด์/แบ็กกราวด์ (foreground/background)และนอนพรีเอมทีฟเคอร์เนล (non-preemtive kernel) นั้นโค้ดใน ISR จะทำงานทันทีเมื่อเก็บค่าคอนเท็กซ์ของโปรเซสเซอร์เสร็จเรียบร้อย

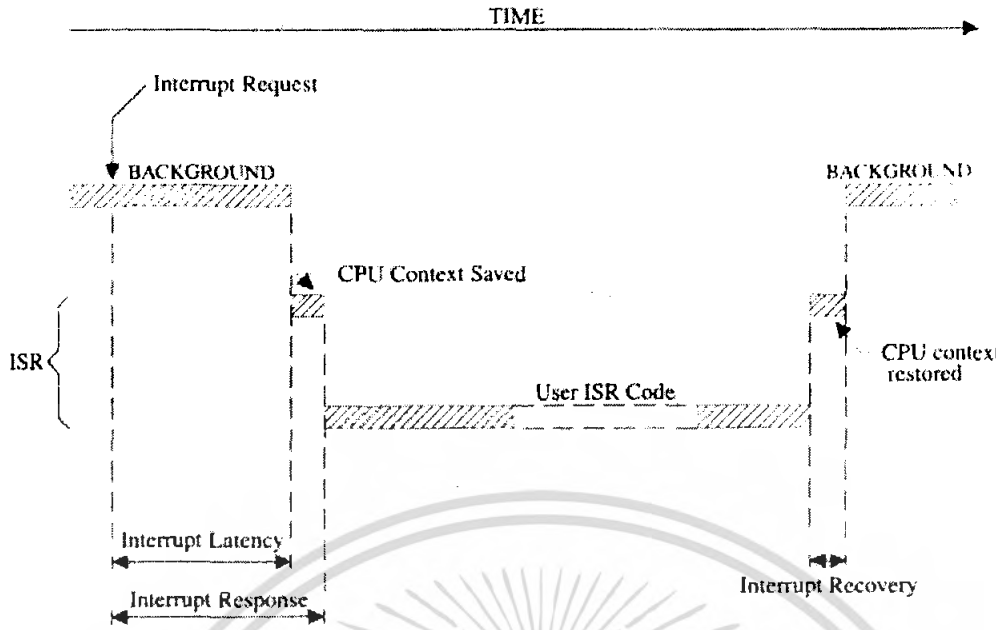
- สำหรับพรีเอ็มทีฟเคอร์เนล (preemptive kernel) จะมีการเรียกใช้งานฟังก์ชั่นพิเศษซึ่งจะต้องเรียกเพื่อแจ้งบอกให้เคอร์เนลรู้ว่า ISR กำลังทำงานอยู่และเคอร์เนลสามารถเก็บ การทำงานของอินเตอร์รัพต์แบบซ้อนกันได้ ซึ่งจะใช้เวลานานกว่าฟอร์กราวด์/แบ็กกราวด์และนอนพรีเอมทีฟเคอร์เนล เพราะมีขั้นตอนในการทำงานเยอะกว่า

2.5.3 การออกจากอินเตอร์รัพต์(Interrupt Recovery)

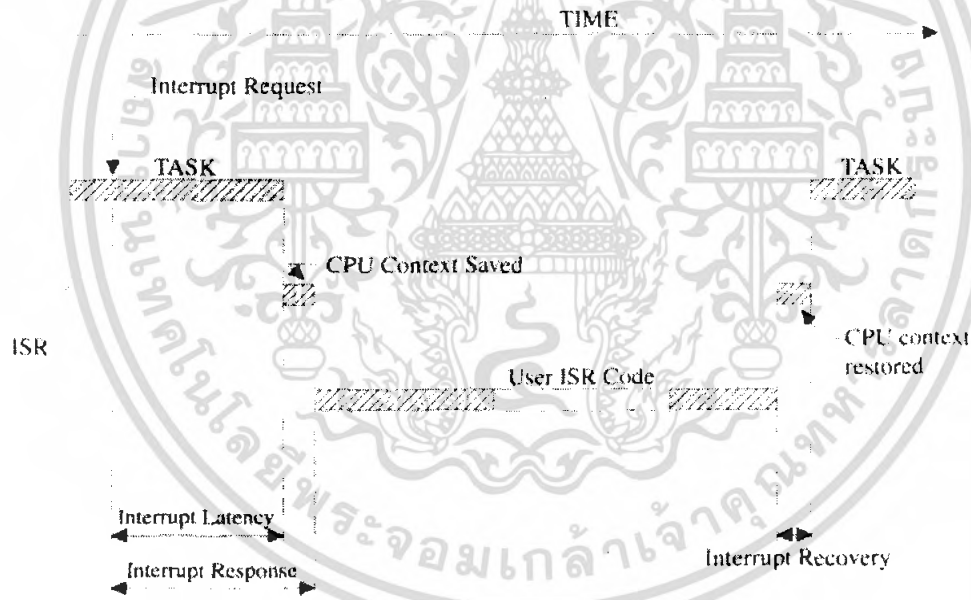
อินเตอร์รัพต์รีคัฟเวอร์รี่ (interrupt recovery) คือเวลาที่โปรเซสเซอร์ต้องการที่จะคืนค่าไปยังโค้ดส่วนที่ถูกอินเตอร์รัพต์

- ระบบฟอร์กราวด์/แบ็กกราวด์ (foreground/background system) และอินเตอร์รัพต์รีคัฟเวอร์ของนอนพรีเอ็มทีฟเคอร์เนล (non-preemptive kernel) จะทำงานคล้าย ๆ กัน คือ ดึงค่าของคอนเท็กซ์คืนให้ซีพียูและคืนค่ากลับไปยังทาสก์ที่ถูกอินเตอร์รัพต์

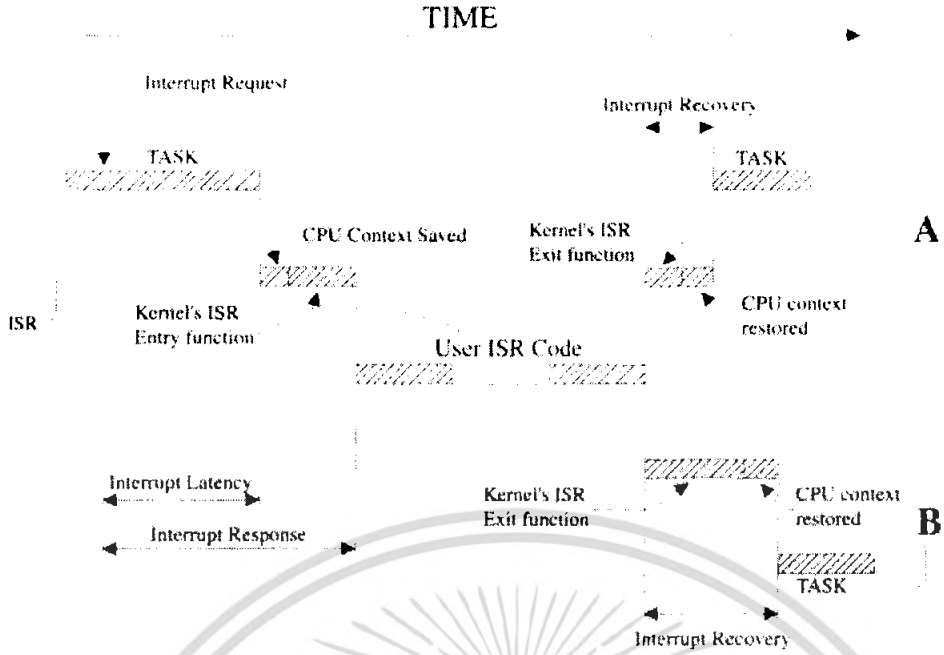
- สำหรับพรีเอมทีฟเคอร์เนล (preemptive kernel) นั้นอินเตอร์รัพต์รีคัพเวอร์จะซับซ้อนกว่าโดยฟังก์ชั่นที่จัดเตรียมโดยเคอร์เนลจะถูกเรียกเมื่อเสร็จสิ้น ISR ซึ่งจะช่วยเคอร์เนลในการเลือกกรณีเมื่อมีอินเตอร์รัพต์ซ้อนกัน (interrupt nest)โดยจะเลือกทาสก์ที่มีความสำคัญสูงสุดที่และพร้อมจะทำงานมาทำต่อจาก ISR



รูปที่ 2.26 Interrupt latency , response และ recovery (foreground/background)



รูปที่ 2.27 Interrupt latency , response และ recovery (non-preemptive kernel)



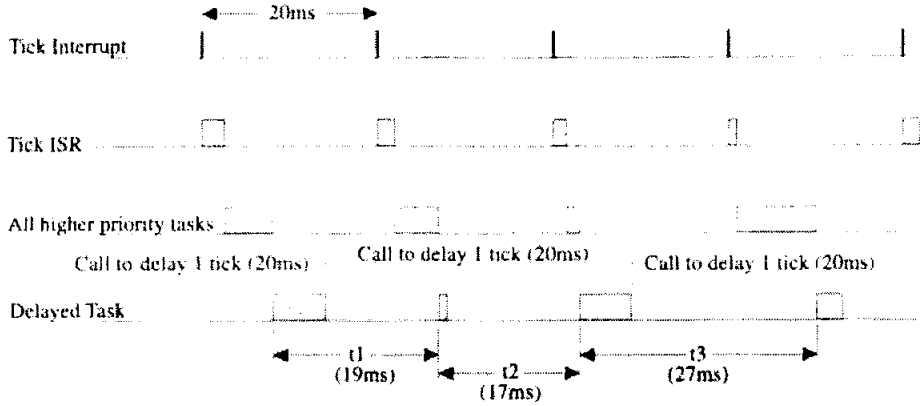
รูปที่ 2.28 Interrupt latency , response และ recovery (preemptive kernel)

2.5.4 คลอกทิก(Clock Tick)

clock tick คือ อินเตอร์รัพต์พิเศษซึ่งจะเกิดขึ้นเป็นรอบ ๆ ที่แน่นอน ซึ่งมักจะสร้างมาในช่วงระหว่าง 10 ถึง 200 ms ซึ่ง clock tick interrupt จะช่วยเคอร์เนลในการดีเลย์ทาสก์ (delay task) เพื่อหาจำนวน clock tick และช่วยจัดการ ไทม์เอาท์ เมื่อทาสก์รอให้อีเวนท์ให้เกิดขึ้น ถ้ามี tick rateที่สูงจะทำให้ระบบทำงานหนักขึ้นได้

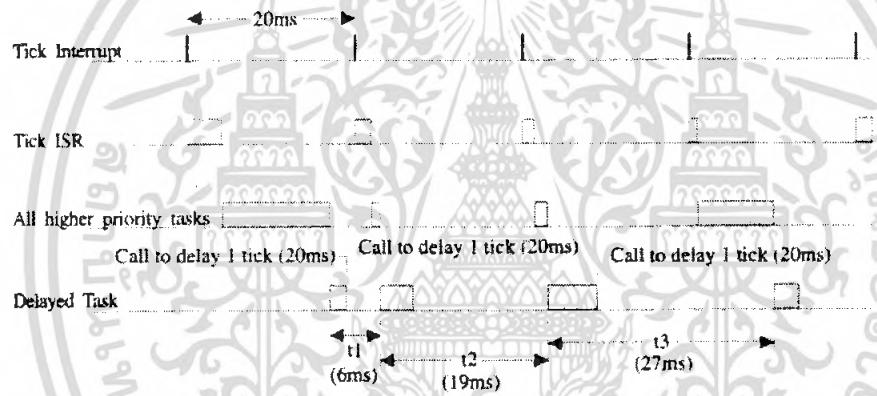
เคอร์เนลจะสามารถให้ทาสก์ดีเลย์ได้เพื่อหาให้จำนวนของ clock tick ที่แน่นอน ซึ่งเมื่อเกิดclock tick ครั้งหนึ่งก็จะมีการดีเลย์แต่ก็ไม่แน่เสมอไปว่าจะเกิดดีเลย์ทุกครั้งที่เกิด clock tick

กรณีที่ 1 จะแสดงให้เห็นว่าทาสก์ที่มีความสำคัญสูงและการทำงานของISR ก่อนหน้าทาสก์จะมีการดีเลย์ในแต่ละ1 ทิคก์ (tick) ซึ่งแต่ละทาสก์ต้องการจะดีเลย์ 20 ms แต่เพราะว่าค่าความสำคัญทำให้การทำงานจริงๆ จะไม่ตรง



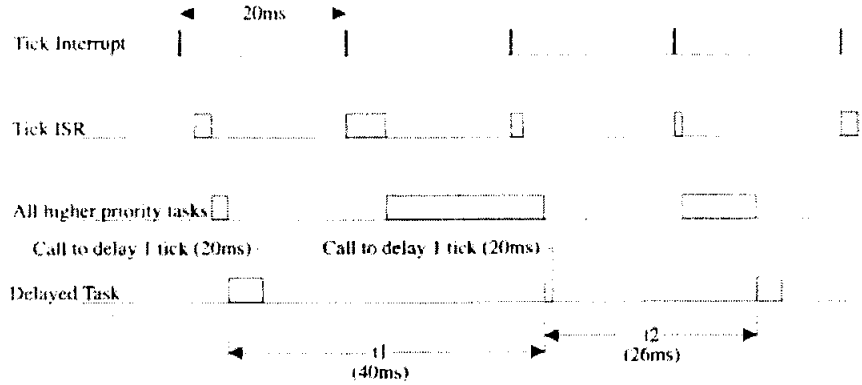
รูปที่ 2.29ดีเลย์ทาสก์ กรณีที่ 1

กรณีที่ 2 แสดงให้เห็นเมื่อเวลาในการทำงานของทาสก์ที่มีความสำคัญสูงและ ISR ใช้เวลาน้อยกว่า 1 ทิคก์ถ้าทาสก์มีดีเลย์ก่อนที่จะถึงคล็อกทิคก์ ทาสก์จะทำงานอีกครั้งในจะทันที



รูปที่ 2.30ดีเลย์ทาสก์ กรณีที่ 2

กรณีที่ 3 แสดงให้เห็นเมื่อเวลาในการทำงานทั้งหมดของทาสก์ที่มีความสำคัญสูงและ ISR มีระยะเวลามากกว่า 1 ทิคก์ทำให้ทาสก์พยายามดีเลย์ 1 ทิกซึ่งจริง ๆ แล้วทำงานไป 2 ทิก ซึ่งกรณีนี้อาจจะยอมรับในบางแอพพลิเคชั่น แต่โดยทั่วไปแล้วไม่ควร



รูปที่ 2.31ดีเลย์ทาสก์ กรณีที่ 3

กรณีต่าง ๆ จะปรากฏกับเรียลไทม์เคอร์เนลทั่วไป ซึ่งอาจจะทำให้ซีพียูทำงานหนักและเกิดข้อผิดพลาดได้ ซึ่งอาจจะแก้ปัญญหาได้โดยดังนี้

- เพิ่มคล็อกเรต (clock rate)ในไมโครคอนโทรลเลอร์ขึ้น

- เพิ่มระยะเวลาระหว่างทิคก์อินเตอร์รัพต์ (tick interrupt)

- ปรับปรุงค่าความสำคัญของทาสก์ (task priority)

- เขียนโค้ดในส่วนที่ต้องการความรวดเร็วด้วยการใช้ภาษาแอสแซมบลี (assembly)

- ใช้งานใช้งานไมโครโปรเซสเซอร์ที่มีความเร็ว

2.6 คริติคอลเซคชั่นของโค๊ด

คริติคอลเซคชั่น (critical section) ของโค้ดหรือเรียกว่าคริติคอลรีเจียน (critical region) คือ โค้ดซึ่งต้องการทำงานโดยไม่มีการขัดการทำงาน อย่างเช่นส่วนเริ่มการทำงานของโค้ดจะต้องไม่ถูกขัดจังหวะการทำงาน เพื่อป้องกันการเกิดการขัดจังหวะซึ่งส่วนใหญ่แล้วอินเตอร์รัพต์จะถูกดิสเอเบิลเอาไว้ ก่อนที่คริติคอลโค้ดจะทำงานและจะเอนาเบิลเมื่อส่วนคริติคอลโค้ดทำงานเสร็จแล้ว

2.7 ความต้องการหน่วยความจำ

ถ้าทำการใช้ระบบฟอร์กราวด์/แบ็กกราวด์ (foreground/background) จำนวนเมมโมรี่ (memory) ที่ต้องใช้งานขึ้นอยู่กับตัวโค้ดของแอพพลิเคชั่น ซึ่งมัลติทาสกิ้งเคอร์เนลจะแตกต่างออกไป โดยเคอร์เนลต้อง

การพื้นที่สำหรับโค้ดพิเศษหรือ ROMขนาดของ เคอร์เนลจะขึ้นอยู่กับหลายปัจจัย เช่น ฟีเจอร์(feature)ของเคอร์เนล การทำคอนเท็กซ์ สวิตชิ่ง (context switching) การจัดการเซมาฟอร์ ดีเลย์ไทม์เอาท์ต่างๆ และอื่น ๆ

เนื่องจากแต่ละทาสก์ทำงานอิสระต่อกัน ซึ่งแต่ละทาสก์จะมีพื้นที่สแตกเป็นของตัวเอง (RAM) โดยในการออกแบบนั้นจะต้องคิดถึงสแตกที่ต้องใช้ในแต่ละทาสก์ซึ่งไม่เพียงแต่ใช้สำหรับใช้งานทั่วไปในทาสก์เท่านั้น แต่ยังบ่งบอกถึงจำนวนการทำอินเตอร์รัพต์ซ้อนกัน (interrupt nest) ซึ่งก็ขึ้นอยู่กับโปรเซสเซอร์และเคอร์เนลที่ใช้

ถ้ามี RAM เพื่อใช้ในการทำงานมากพอ ก็จะต้องระมัดระวังในการใช้เพื่อลดความต้องการ RAM ในตัวแอพพลิเคชั่นทั้งหมด ซึ่งควรจะระมัดระวังในการใช้งานสำหรับ

- อาเรย์ (array) และโครงสร้าง(structure)ที่มีขนาดใหญ่
- การทำฟังก์ชั่นซ้อนกัน (function nesting)
- การทำอินเตอร์รัพต์ซ้อนกัน (interrupt nesting)
- การใช้ไลบรารีฟังก์ชั่น (library function) และฟังก์ชั่นซึ่งมีอาร์กูเมนต์ (argument) มาก

โดยสรุปมัลติทาสกิ้งซิสเต็มต้องการพื้นที่สำหรับโค้ด (ROM) และพื้นที่สำหรับข้อมูล (RAM) มากกว่าระบบฟอร์กราวด์/แบ็กกราวด์ (foreground/background) โดยขนาดของ ROM ขึ้นอยู่กับขนาดของเคอร์เนล และขนาดของ RAM ขึ้นอยู่กับจำนวนของทาสก์ในระบบ

2.8 ข้อดีและข้อเสียของเรียลไทม์เคอร์เนล

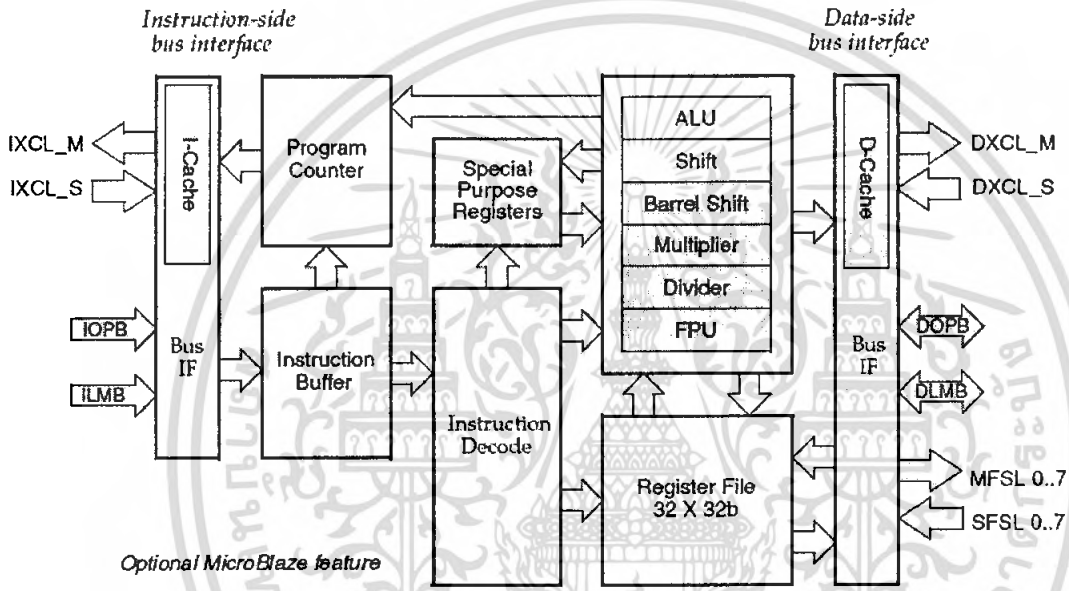
เรียลไทม์เคอร์เนล (real time kernel) หรือบางครั้งเรียกว่าเรียลไทม์ โอเปอร์เรติ้งซิสเต็ม (realtime operating system) หรือ RTOS จะใช้เพื่อให้ง่ายต่อการออกแบบและพัฒนากับเรียลไทม์แอพพลิเคชั่น ซึ่งฟังก์ชั่นต่าง ๆ สามารถเพิ่มขึ้นได้โดยไม่ต้องเปลี่ยนซอฟต์แวร์ตัวหลัก การใช้งาน RTOS จะออกแบบให้โค้ดของแอพพลิเคชั่นแยกย่อยออกเป็นทาสก์ และการทำงานจะสามารถตอบสนองได้ทันต่อเหตุการณ์และมีประสิทธิภาพ RTOS จะยังช่วยจัดการในเรื่องต่าง ๆ ไม่ว่าจะเป็นเซมาฟอร์ , เมลบ็อกซ์ , คิว , ไทม์ดีเลย์ เป็นต้น แต่ค่าใช้จ่ายในการพัฒนาให้รองรับกับงานหลาย ๆ ด้าน จะทำให้ต้นทุนในการพัฒนาและพื้นที่ของหน่วยความจำในระบบที่ต้องการอาจเพิ่มสูงได้

บทที่3

สถาปัตยกรรมของไมโครเบลส

3.1 ภาพรวม

ไมโครเบลสเป็นซอฟท์คอร์โพรเซสเซอร์แบบ RISC (reduced instruction set computer) ซึ่งถูกออกแบบมาให้ใช้บนเอฟพีจีเอของไซลิงค์ รูปต่อไปนี้คือผังการทำงานโดยรวมของไมโครเบลสคอร์



รูปที่ 3.1ผังการทำงานโดยรวมของ ไมโครเบลส คอร์

3.1.1 คุณสมบัติ

ไมโครเบลสเป็นซอฟคอร์โพรเซสเซอร์ที่อนุญาตให้เราปรับแต่งคุณสมบัติต่างๆให้เข้ากับงานของเราได้หลายที่ด้วยกัน

คุณสมบัติที่เปลี่ยนแปลงไม่ได้ได้แก่

- รีจิสเตอร์(register)ทั่วไปขนาด 32 บิตทั้งสิ้น 32 ตัว
- ชุดคำสั่งขนาด 32 บิตซึ่งประกอบไปด้วยโอเปอแรนด์(operand) 3 ตัวและวิธีการอ้างตำแหน่ง (addressing mode) 2 แบบ
- แอดเดรสบัส(address bus) ขนาด 32 บิต
- ไปป์ไลน์(pipeline)

ส่วนคุณสมบัติที่ปรับแต่งได้ของไมโครเบลสเป็นดังตารางนี้

Feature	MicroBlaze Versions				
	v2.10a	v3.00a	v4.00	v5.00	v6.00
Version Status	deprecated	deprecated	deprecated	deprecated	preferred
Processor pipeline depth	3	3	3	5	3/5
On-chip Peripheral Bus (OPB) data side interface	option	option	option	option	option
On-chip Peripheral Bus (OPB) instruction side interface	option	option	option	option	option
Local Memory Bus (LMB) data side interface	option	option	option	option	option
Local Memory Bus (LMB) instruction side interface	option	option	option	option	option
Hardware barrel shifter	option	option	option	option	option
Hardware divider	option	option	option	option	option
Hardware debug logic	option	option	option	option	option
Fast Simplex Link (FSL) interfaces	0-7	0-7	0-7	0-7	0-7
Machine status set and clear instructions	option	option	option	Yes	option
Instruction cache over IOPB interface	option	option	option	No	No
Data cache over IOPB interface	option	option	option	No	No
Instruction cache over CacheLink (IXCL) interface	-	option	option	option	option
Data cache over CacheLink (DXCL) interface	-	option	option	option	option
4 or 8-word cache line on XCL	-	4	4	option	option
Hardware exception support	-	option	option	option	option
Pattern compare instructions	-	-	option	Yes	option
Floating point unit (FPU)	-	-	option	option	option
Disable hardware multiplier[1]	-	-	option	option	option
Hardware debug readable ESR and EAR	-	-	Yes	Yes	Yes
Processor Version Register (PVR)	-	-	-	option	option
Area or speed optimized	-	-	-	-	option
Hardware multiplier 64-bit result	-	-	-	-	option
LUT cache memory	-	-	-	-	option

1. Used in Virtex™-II and subsequent families, for saving MUL18 and DSP48 primitives.

ตาราง 3.1คุณสมบัติที่ปรับแต่งได้ของไมโครเบลส เรียงตามรุ่นของไมโครเบลส

3.2 ชนิดและการจัดเรียงตัวของข้อมูล

ไมโครเบลส ใช้การจัดเรียงตัวของข้อมูลกลับบิตแบบบิ๊กเอนเดียน(Big-Endian bit-reversed) ในการแสดงข้อมูล ซึ่งชนิดของข้อมูลที่รองรับ โดยฮาร์ดแวร์ของ ไมโครเบลส นั้นได้แก่ เวิร์ด(word) , ฮาล์ฟเวิร์ด

(half word) และ ไบต์(byte) ซึ่งการจัดเรียงตัวของบิตและไบต์ของแต่ละชนิดของข้อมูลนั้นเป็นตามตาราง ต่อไปนี้

Byte address	n	n+1	n+2	n+3
Byte label	0	1	2	3
Byte significance	MSByte			LSByte
Bit label	0			31
Bit significance	MSBit			LSBit

ตาราง 3.2ข้อมูลชนิดเวิร์ด

Byte address	n	n+1
Byte label	0	1
Byte significance	MSByte	LSByte
Bit label	0	15
Bit significance	MSBit	LSBit

ตาราง 3.3ข้อมูลชนิดฮาล์ฟเวิร์ด

Byte address	n	
Bit label	0	7
Bit significance	MSBit	LSBit

ตาราง 3.4ข้อมูลชนิดไบต์

3.3 ชุดคำสั่ง

ชุดคำสั่งทั้งหมดของ ไมโครเบลส นั้นมีขนาด 32 บิต และแบ่งออกเป็น ชนิด A และ ชนิด B โดย ชนิด A นั้นมีรีจิสเตอร์ที่เป็นซอสโอเปอแรนด์(source operand)ทั้งสิ้น 2 ตัวและรีจิสเตอร์ที่เป็นเดสติเนชั่น

โอเปอแรนด์(destination operand)หนึ่งตัว ส่วน ชนิด B นั้นประกอบไปด้วยซอสรีจิสเตอร์ 1 ตัวเดสติเนชั่นรีจิสเตอร์ 1 ตัว และอิมมิเดียดโอเปอแรนด์(immediate operand) อีก 1 ตัว

3.4 รีจิสเตอร์

ไมโครเบลส นั้นประกอบไปด้วย รีจิสเตอร์แบบทั่วไปที่มีขนาด 32 บิตทั้งสิ้น 32 ตัวและ แบบมีหน้าที่พิเศษ ที่มีขนาด 32 บิต สูงสุดถึง 18 ตัวซึ่งจะมีกี่ตัวนั้นขึ้นอยู่กับการตั้งค่าของ ไมโครเบลส

3.5 สถาปัตยกรรมของไปป์ไลน์

การทำงานของ ไมโครเบลส นั้นได้ถูกทำไปป์ไลน์ไว้แล้ว โดยในเกือบทุกๆ คำสั่ง แต่ละ สเตจ (stage) นั้นจะใช้ 1 คลอกไซเคิล(clock cycle) ในการทำงาน ดังนั้นจำนวนคลอกไซเคิลที่ต้องใช้ในการทำงานคำสั่งให้เสร็จสิ้นนั้นจะเท่ากับจำนวนสเตจของไปป์ไลน์และในทุกๆคลอกไซเคิลนั้นจะมี 1 คำสั่งที่ทำงานเสร็จสิ้น แต่มีบางคำสั่งที่ต้องใช้มากกว่า 1 คลอกไซเคิลในเอกซิคิวสเตจซึ่งก็ยังสามารถทำงานได้โดยการสตอล(stall)ไปป์ไลน์

3.5.1 ไปป์ไลน์แบบ 3 สเตจ

หากมีการเปิดตัวเลือกแอเรียออพติไมเซชั่น (area optimization) ในขั้นตอนการสังเคราะห์ตัว ไมโครเบลส แล้วไปป์ไลน์จะถูกแบ่งออกเป็น 3 สเตจเพื่อลดขนาดของฮาร์ดแวร์ซึ่งได้แก่ เฟตช์(Fetch), ดีโค๊ด (Decode) และ เอกซิคิว(Execute)

	cycle 1	cycle 2	cycle 3	cycle4	cycle5	cycle6	cycle7
instruction 1	Fetch	Decode	Execute				
instruction 2		Fetch	Decode	Execute	Execute	Execute	
instruction 3			Fetch	Decode	Stall	Stall	Execute

รูปที่ 3.2ไปป์ไลน์แบบ 3 สเตจ

3.5.2 ไปป์ไลน์แบบ 5 สเตจ

หากไม่มีการเปิดตัวเลือกแอเรียออพติไมเซชั่นในขั้นตอนการสังเคราะห์ตัว ไมโครเบลส แล้วไปป์ไลน์จะถูกแบ่งออกเป็น 5 สเตจเพื่อเพิ่มประสิทธิภาพในการทำงาน ซึ่งได้แก่ Fetch(IF), Decode(OF), Execute(EX), Access Memory(MEM) และ Writeback(WB)

	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 7	cycle 8	cycle 9
instruction 1	IF	OF	EX	MEM	WB				
instruction 2		IF	OF	EX	MEM	MEM	MEM	WB	
instruction 3			IF	OF	EX	Stall	Stall	MEM	WB

รูปที่ 3.3ไปป์ไลน์แบบ 5 สเตจ

3.5.3 แบรนช์(Branches)

โดยปกติแล้ว คำสั่งที่อยู่ในเฟตช์และดีโค้ดสเตจนั้นจะถูกล้างทิ้งทันทีที่การเอกซีคิวนั้นพบกับคำสั่งประเภทแบรนช์จากนั้นไปป์ไลน์จะถูกโหลดด้วยคำสั่งใหม่ ทำให้การแบรนช์ใน ไมโครเบลส นั้นใช้ทั้งสิ้น 3 คลอคไซเคิลก่อนที่จะทำการทำงานต่อไปได้ ซึ่ง 2 ใน 3 คลอคไซเคิลที่เสียไปนี้ถูกใช้เพื่อเติมค่ากลับลงมาในไปป์ไลน์ดังนั้น เพื่อลดความล่าช้าที่เกิดขึ้นนี้ ไมโครเบลส จึงถูกออกแบบให้รองรับการแบรนช์แบบมีดีเลย์สลอต

3.5.3.1 ดีเลย์สลอต(Delay Slots)

ขณะที่ทำการแบรนช์โดยมีดีเลย์สลอตนั้น มีเพียงแค่เฟตช์สเตจในไปป์ไลน์เท่านั้นที่จะถูกล้างค่าทิ้ง ส่วนชุดคำสั่งที่อยู่ในดีโคดสเตจ(หรือเรียกว่าเป็นดีเลย์สลอต) นั้นจะยอมให้ถูกทำงานต่อไป การใช้เทคนิคนี้ช่วยลดความสูญเสียเวลาจากการแบรนช์จาก 2 คลอกไซเคิลเหลือเพียงคลอกไซเคิลเดียวเท่านั้น

3.6 รีเซ็ต(Reset), อินเตอร์รัพต์(Interrupts), เอ็กเซพชั่น(Exceptions) และเบรก(Break)

ไมโครเบลส นั้นรองรับสัญญาณ รีเซ็ต, อินเตอร์รัพต์, เอ็กเซพชั่นจากผู้ใช้, เบรกและ เอ็กเซพชั่นจากฮาร์ดแวร์โดยมีการเรียงลำดับความสำคัญจากมากไปหาน้อยดังนี้

- รีเซ็ต
- เอ็กเซพชั่นจากฮาร์ดแวร์
- นอนมาส์กเอเบิลเบรก(Non-maskable Break)
- เบรก
- อินเตอร์รัพต์
- เอกเซพชั่นจากผู้ใช้

ดารางต่อไปนี้เป็นตำแหน่งของเวกเตอร์แอดเดรส(Vector Address) และ รีจิสเตอร์ไฟล์รีเทอร์นแอดเดรส (Register File Return Address) สำหรับแต่ละเหตุการณ์

Event	Vector Address	Register File Return Address
Reset	0x00000000 - 0x00000004	-
User Vector (Exception)	0x00000008 - 0x0000000C	Rx
Interrupt	0x00000010 - 0x00000014	R14
Break: Non-maskable hardware	0x00000018 - 0x0000001C	R16
Break: Hardware		
Break: Software		
Hardware Exception	0x00000020 - 0x00000024	R17 or BTR
Reserved by Xilinx for future use	0x00000028 - 0x0000004F	-

ตาราง 3.5เวกเตอร์ และ ตำแหน่งของรีเทอร์นแอดเดรสไฟล์

3.6.1 รีเซต

เมื่อ ไมโครเบลส ได้รับสัญญาณ Reset หรือ Debug_Rst แล้ว จะทำการล้างไปป์ไลน์และเริ่มต้นการเฟตช์ชุดคำสั่งจากรีเซตเวกเตอร์(ที่ตำแหน่ง 0x0) ซึ่งสัญญาณรีเซตจากภายนอกทั้งสองแหล่งนี้ทำงานที่สถานะสูงและจะต้องมีความยาวไม่น้อยกว่า 16 ไซเคิล

3.6.1.1 คำสั่ง ซูโดโค๊ด(Psuedocode) ที่อธิบายกระบวนการรีเซต

```
PC   <-   0x00000000

MSR  <-   C_RESET_MSR

EAR  <-   0

ESR  <-   0

FSR  <-   0
```

3.6.2 อินเตอร์รัพต์

ไมโครเบลส รองรับการเกิดสัญญาณ อินเตอร์รัพต์ จากภายนอก (ที่ต่อเข้ากับช่องรับสัญญาณอินเตอร์รัพต์) โดยตัวโพรเซสเซอร์จะตอบสนองต่ออินเตอร์รัพต์ก็ต่อเมื่อบิต Interrupt Enable(IE) ใน Machine Status Register(MSR) นั้นถูกตั้งค่าเป็น 1 ไว้เท่านั้น เมื่อเกิดการอินเตอร์รัพต์คำสั่งที่อยู่ในเอกซิคิวชั่นสเตจจะยังคงถูกทำงานต่อไปจนเสร็จสิ้น ในขณะที่คำสั่งที่อยู่ในดีโค๊ดสเตจนั้นจะถูกเปลี่ยนเป็นคำสั่ง

แบรนช์ไปยังอินเตอร์รัพต์เวกเตอร์(ที่ตำแหน่ง 0x10) ส่วนอินเตอร์รัพท์รีเทอร์นแอดเดรส(interrupt return address)นั้นจะถูกโหลดจาก PC ไปไว้ยัง R14 โดยอัตโนมัติ และโพรเซสเซอร์ยังทำการเคลียค่าในบิต IE ใน MSR ทิ้งโดยอัตโนมัติ เพื่อปิดไม่ให้มีการเกิดอินเตอร์รัพต์ขึ้นอีก และบิต IE นั้นจะถูกเซ็ตกลับดังเดิมทันทีที่มีการเรียกใช้คำสั่ง RTID

อินเตอร์รัพต์นั้นจะไม่มีการตอบสนองโดยโพรเซสเซอร์ถ้าหากบิต break in progress(BIP) หรือ exception in progress(EIP) ใน MSR นั้นถูกตั้งเป็น 1

3.6.2.1 ความล่าช้า

เวลาที่ถูกใช้โดย ไมโครเบลส เพื่อที่จะเข้าสู่ฟังก์ชันการตอบสนองอินเตอร์รัพต์(ISR) โดยนับจากที่สัญญาณอินเตอร์รัพต์เกิดขึ้นนั้นขึ้นอยู่กับการตั้งค่าของตัวโพรเซสเซอร์และความล่าช้าจากตัวเมมโมรี่คอนโทรลเลอร์ในการเก็บอินเตอร์รัพต์เวกเตอร์โดยถ้าหาก ไมโครเบลส ถูกตั้งค่าไว้ให้มีตัวหารซึ่งเป็นฮาร์ดแวร์ความล่าช้าสูงที่สุดจะเกิดขึ้นหากสัญญาณอินเตอร์รัพต์เกิดในขณะที่กำลังทำงานคำสั่งหารอยู่

3.6.2.2 คำสั่งซูโดโค้ดที่อธิบายกระบวนการอินเตอร์รัพต์

```
r14     <-    PC

PC      <-    0x00000010

MSR[IE]       <-    0
```

บทที่4

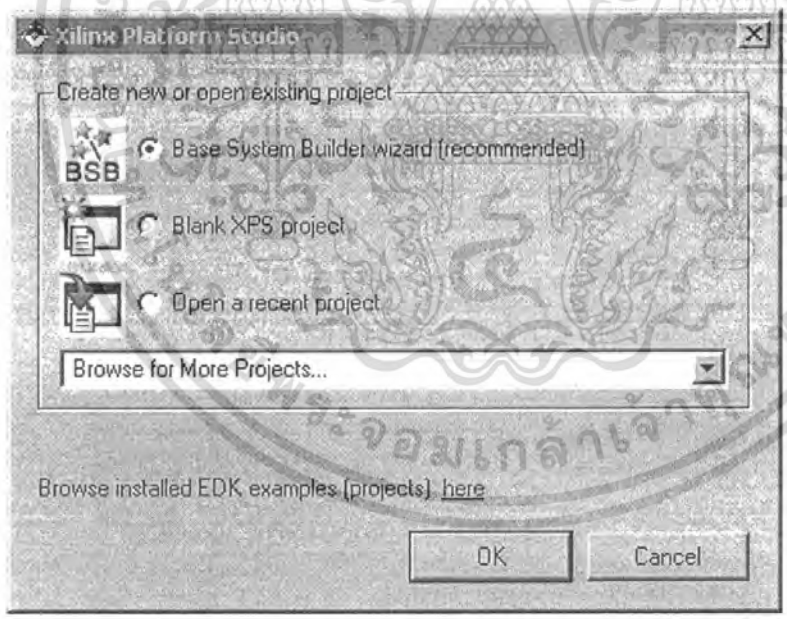
การสร้างระบบไมโครเบลสอย่างง่าย

4.1 ภาพรวม

เนื้อหาในบทนี้จะเป็นเอกสารประกอบการทดลองการสร้างระบบไมโครเบลสอย่างง่าย โดยภายในระบบจะประกอบไปด้วย ตัวคอร์ของไมโครเบลส และ ยูอาร์ท(UART) โมดูล โดยการทำงานของระบบคือ เมื่อระบบถูกทำการรีเซตจะพิมพ์ตัวหนังสือว่า "Hello MicroBlaze!" ออกมาทางยูอาร์ทโมดูล

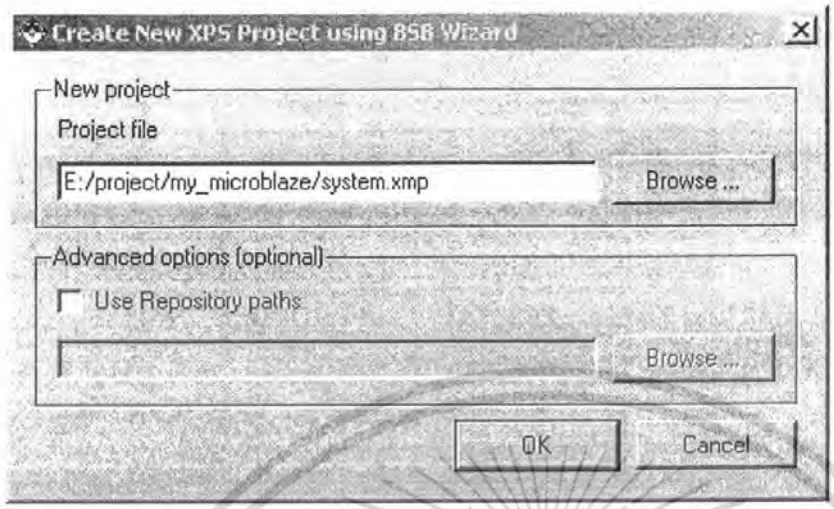
4.2 การสร้างระบบไมโครเบลสด้วย เบสซิสเต็มบิลเดอร์วิซาร์ด (Base System Builder Wizard)

1. หลังจากเปิดโปรแกรมไซลิ้งแพลทฟอร์มสตูดิโอ(Xilinx Platform Studio) ขึ้นมาทำการเลือกไปที่เบสซิสเต็มบิลเดอร์วิซาร์ด



รูปที่ 4.1การเลือกเบสซิสเมบิลเดอร์วิซาร์ด

2. ทำการเลือกตำแหน่งสำหรับเซฟโปรเจคที่ต้องการ



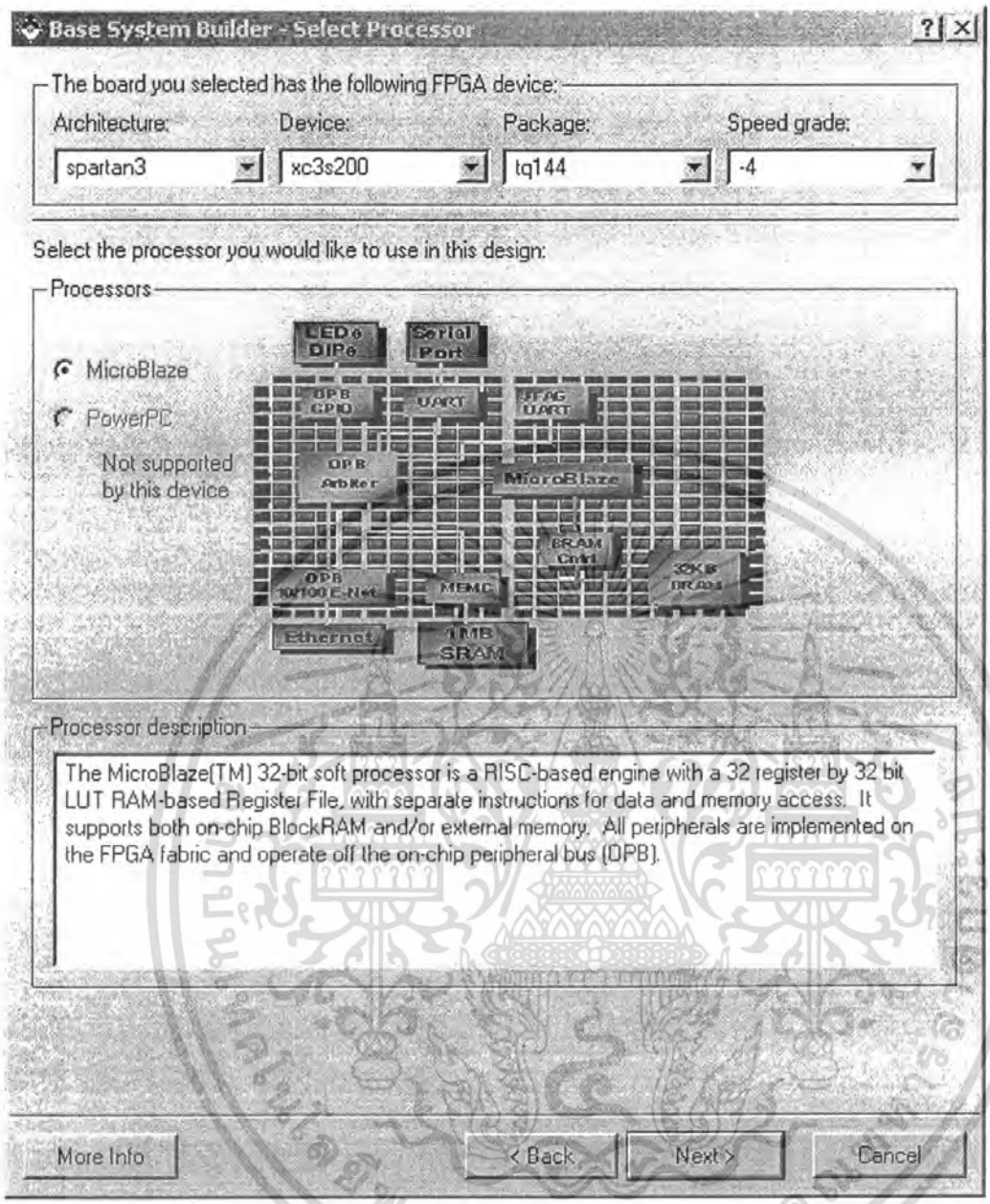
รูปที่ 4.2การเลือกตำแหน่งในการเซฟโปรเจค

3. เลือกที่ "I would like to create a new design" กด Next

4. เลือกที่ "I would like to create a system for a custom board" เนื่องจากเราไม่ได้ใช้บอร์ดของไซลิ้ง และไม่มีบอร์ดเดฟินิชันไฟล์ (Board definition file) มาจากผู้ผลิตบอร์ด

5. เลือกชนิดของเอฟพีจีเอให้ตรงกับบอร์ดที่เราใช้ซึ่งในการทดลองนี้เราใช้บอร์ดที่ใช้ชิพสปาร์ทาน3 เบอร์ เอ็กซ์ซี3เอส200(xc3s200) ของบริษัทเอเพ็กอินสตรูเมนท์(Apex Instrument)

6. เลือกชนิดของโพรเซสเซอร์ให้เป็นไมโครเบลส เนื่องจากในชิพบางเบอร์นั้นจะมีพาวเวอร์พีซี ซึ่งเป็นฮาร์ดโพรเซสเซอร์คอร์อยู่บนชิพ หลังจากตั้งค่าต่างๆ เสร็จแล้วจะได้ดังรูป



รูปที่ 4.3การตั้งค่าเกี่ยวกับเอฟพีจีเอ และโพรเซสเซอร์ที่จะใช้

7. กำหนดค่าเกี่ยวกับตัวไมโครเบลสที่จะสร้างดังนี้

   a. "Refernce clock frequency" คือความถี่ของออสซิเลเตอร์บนบอร์ด ในที่นี้คือ 25 เมกะเฮิร์ท

   b. "Processor-Bus clock frequency" คือความถี่ที่เราจะใช้ในตัวโพรเซสเซอร์ โดยสามารถตั้งให้สูงกว่าออสซิลเลเตอร์ได้โดยระบบจะทำการใช้ดีซีเอ็มโมดูลในการเพิ่มความถี่ให้

   c. "Reset polarity" คือลักษณะของสัญญาณรีเซ็ต ซึ่งปุ่มกดบนบอร์ดของเรานั้นเมื่อกดจะได้สัญญาณ '1' ดังนั้นเราจึงตั้งเป็น "Active HIGH"

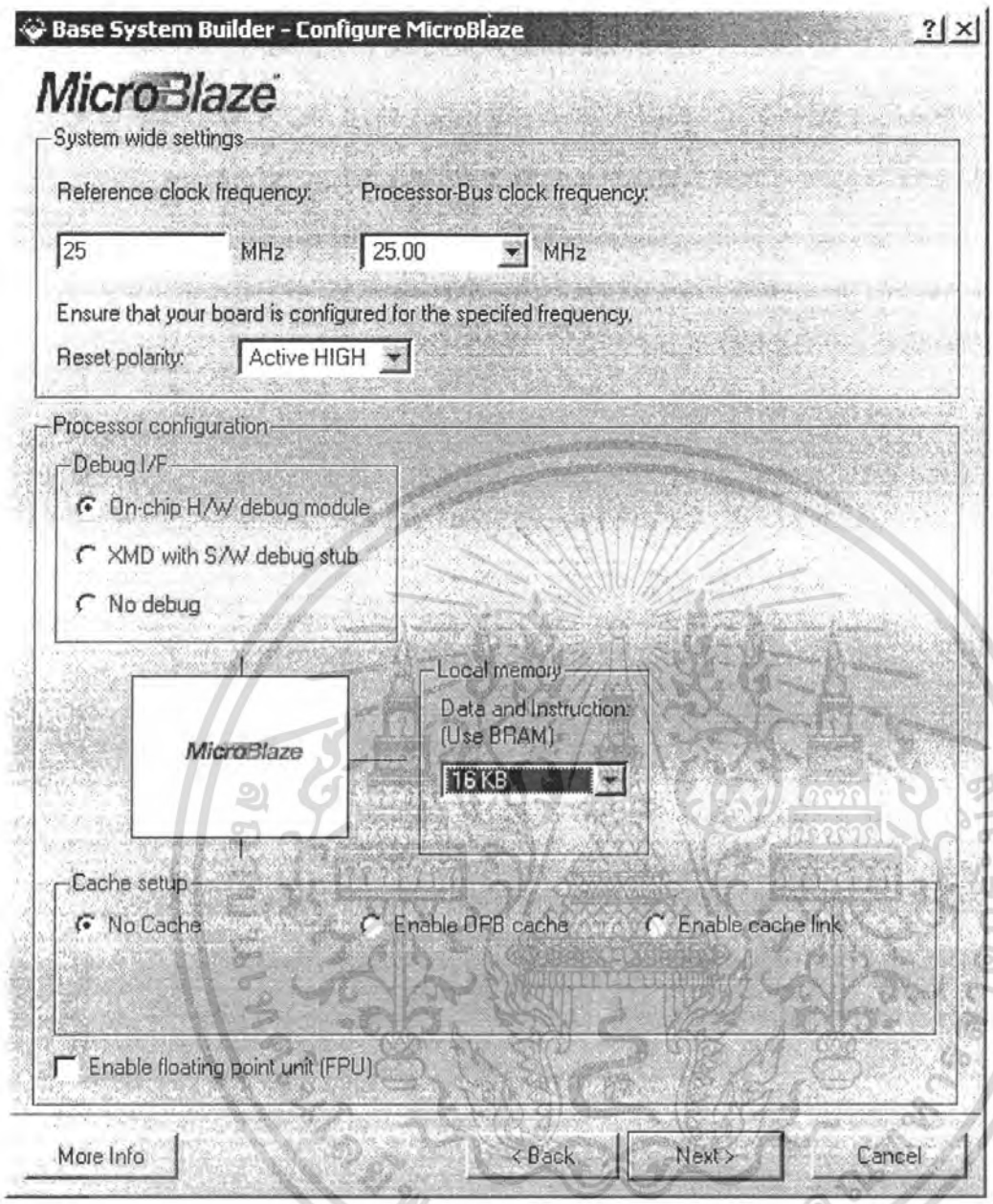
d. “Debug I/F” เป็นการเลือกอินเตอร์เฟสในการดีบักของระบบ

e. “Local memory” เป็นการเลือกขนาดของบลอคแรมที่จะใช้สำหรับเก็บดาต้า และ อินสตรัคชัน ของระบบ

f. “cache setup” เป็นการเลือกว่าจะให้ทำแคชขึ้นมาในระบบหรือไม่

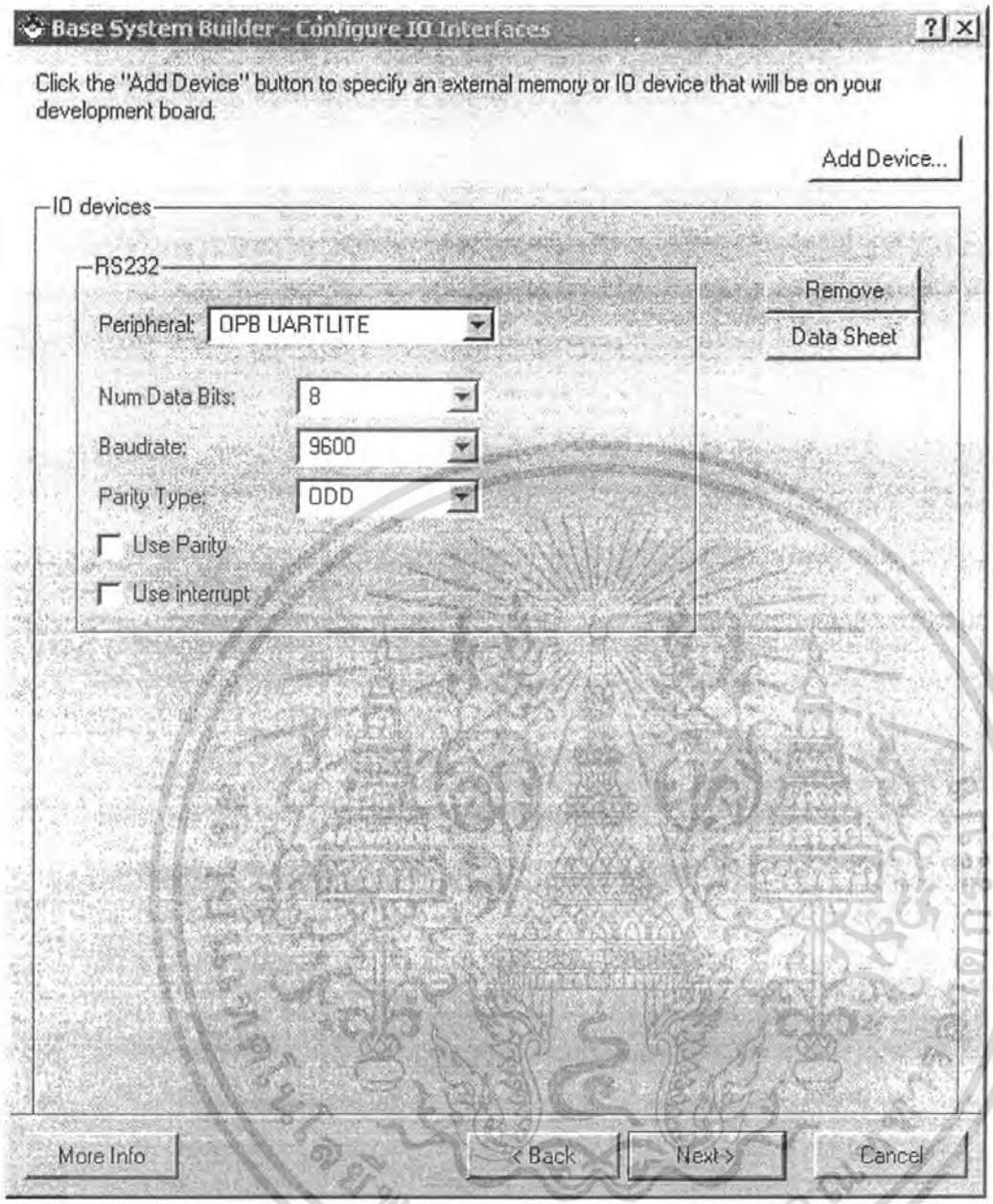
g. “Enable floating point unit” เป็นการเลือกว่าจะมีการใช้งานหน่วยประมวลผลเลขทศนิยมหรือไม่

หลังจากตั้งค่าต่างๆเสร็จเรียบร้อยจะมีหน้าตาดังนี้



รูปที่ 4.4การตั้งค่าต่างๆของตัวไมโครเบลส

8. ทำการเพิ่มไอโอต่างๆ ของระบบโดยการกดที่ปุ่ม “Add Device” โดยในที่นี้เราจะทำการเพิ่มยูอาร์ทโมดูลเข้ามาในระบบเพียงอย่างเดียว ซึ่งเราสามารถเพิ่มโมดูลอื่นๆอีกในภายหลังได้ หลังจากที่เพิ่มยูอาร์ทโมดูลแล้วทำการตั้งค่าต่างๆดังรูป



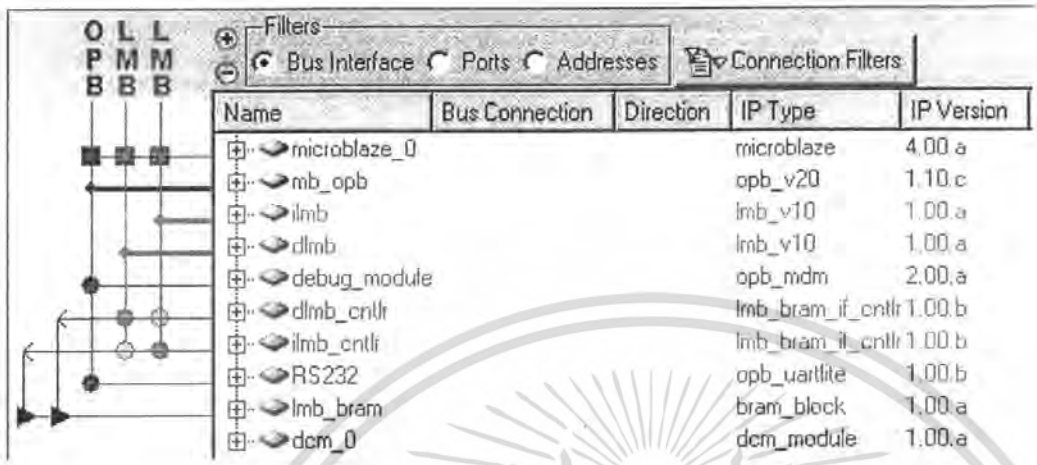
รูปที่ 4.5การเพิ่มไอโอของระบบ

9. สำหรับในหน้าต่อไปจะเป็นการเพิ่มเพอรีเฟอรัลของระบบ ซึ่งในที่นี้เราจะยังไม่เพิ่มอะไร ให้ทำการกด "Next" ข้ามไปก่อน

10. สำหรับหน้า "Software Setup" จะเป็นการตั้งค่าเกี่ยวกับซอฟแวร์ของระบบ โดยเราจะทำการแมพ STDIN และ STDOUT ไปยัง RS232 โมดูลทั้งคู่

11. หลังจากทำการตั้งค่าระบบทั้งหมดเสร็จสิ้นแล้วทำการกดปุ่ม "Generate" แล้วโปรแกรมจะทำการสร้างระบบออกมาให้ตามที่ตั้งค่าไว้

4.3 ภาพของระบบที่ถูกสร้างขึ้นมา



รูปที่ 4.6ภาพรวมของระบบที่ถูกสร้างขึ้น

หลังจากที่เราทำการตั้งค่าต่างๆในเบสซิสเต็มบิลเดอร์วิซาร์ดเรียบร้อยแล้ว โปรแกรมจะทำการสร้างระบบให้เราดังภาพ โดยจะมีตัวซีพียูคือ "microblaze_0" เชื่อมต่อเป็นมาสเตอร์บนบัสทั้งสามบัสได้แก่ "mb_opb" ซึ่งเป็น โอพีบี หรือออนชิพเพอริเฟอรัลบัส (On chip Peripheral Bus) "ilmb" และ "dlmb" ซึ่งเป็น แอลเอ็มบี หรือ โลคอลเมมโมรี่บัส (Local Memory Bus) สำหรับอินสตรัคชันและดาต้าตามลำดับ โดยจุดสี่เหลี่ยมสีเขียวและน้ำเงินแสดงถึงความเป็นมาสเตอร์บนบัสโอพีบีและแอลเอ็มบีตามลำดับ

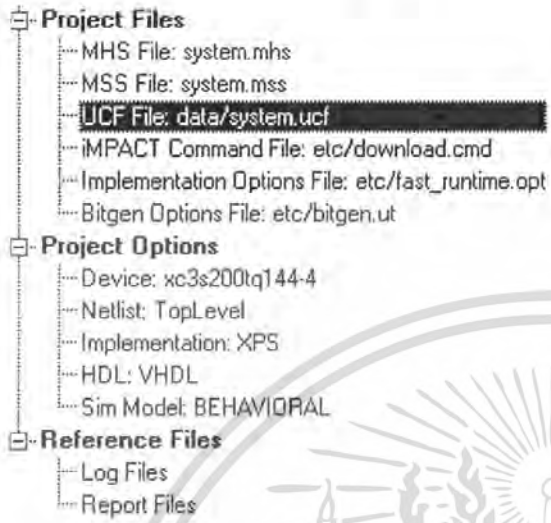
มีอุปกรณ์สองตัวที่เป็นสลาฟบนโอพีบีได้แก่ "debug_module" ซึ่งเป็นฮาร์ดแวร์ที่ใช้ในการดีบักโปรแกรมของระบบและ "RS232" หรือยูอาร์ทโมดูลซึ่งใช้ในการรับส่งข้อมูลผ่านทางโปรโตคอลยูอาร์ท โดยจุดกลมสีเขียวแสดงถึงความเป็นสลาฟบนบัสโอพีบี

อุปกรณ์อีกสองตัวได้แก่ "dlmb_cntlr" และ "ilmb_cntlr" เป็นสลาฟคอนโทรลเลอร์บนบัสแอลเอ็มบี ซึ่งมีหน้าที่ในการติดต่อกับบลอคแรมในตัวเอฟพีจีเอ โดยจุดกลมสีน้ำเงินแสดงถึงความเป็นสลาฟบนแอลเอ็มบี และเส้นสีม่วงแสดงถึงการเชื่อมต่อกับบลอคแรม

สำหรับอุปกรณ์ที่ชื่อเป็นสีดำคืออุปกรณ์ที่ไม่ได้เชื่อมต่ออยู่บนบัสใดเลยได้แก่ "dcm_0" ซึ่งเป็นวงจรเฟสลอคลูปที่ใช้ในการเพิ่มและลดความถี่จากแหล่งกำเนิดความถี่ของระบบหรือออสซิลเลเตอร์ของบอร์ดนั่นเอง

4.4 การนำระบบไมโครเบลสลงไปทำงานบนบอร์ดเอฟพีจีเอ

1. ทำการเปิดยูซีเอฟ(ucf) ไฟล์ของระบบ "system.ucf" ขึ้นมาแก้ไข



รูปที่ 4.7การเปิดยูซีเอฟไฟล์

2. ทำการแก้ไขยูซีเอฟไฟล์โดยทำการเอาคอมเมนท์ตรงหน้าเนท(net)เหล่านี้ออก "sys_clk_pin", "sys_rst_pin", "fpga_0_RS232_RX_pin" และ "fpga_0_RS232_TX_pin" แล้วทำการใส่ตำแหน่งของขาเหล่านั้นตรง "LOC=" โดยตำแหน่งของขาสามารถเปิดดูได้จากคู่มือการใช้บอร์ด หลังจากที่ทำการแก้ไขเสร็จแล้วไฟล์จะมีหน้าตาดังนี้

```
1  ##############################################################################
2  ## This system.ucf file is generated by Base System Builder based on the
3  ## settings in the selected Xilinx Board Definition file. Please add other
4  ## user constraints to this file based on customer design specifications.
5  ##############################################################################
6
7  Net sys_clk_pin LOC=P127;
8  Net sys_rst_pin LOC=P11;
9  ## System level constraints
10 Net sys_clk_pin TNM_NET = sys_clk_pin;
11 TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 40000 ps;
12 Net sys_rst_pin TIG;
13
14 ## IO Devices constraints
15
16 #### Module RS232 constraints
17
18 # Net fpga_0_RS232_req_to_send_pin LOC=;
19 Net fpga_0_RS232_RX_pin LOC=P129;
20 Net fpga_0_RS232_TX_pin LOC=P128;
21
22
```

รูปที่ 4.8ไฟล์ยูซีเอฟที่ถูกแก้ไขแล้ว

3. เลือกเมนูไปที่ "Hardware > Generate Bitstream" แล้วรอดูที่คอนโซลด้านล่าง ซึ่งขั้นตอนนี้ค่อนข้างใช้เวลานานพอสมควร ขึ้นอยู่กับข้อบังคับของระบบที่เราตั้งไว้ อย่างเช่นถ้าหากตั้งให้คอร์ของไมโครเบลสทำงานที่ความถี่ 100 เมกะเฮิร์ทซ์ก็อาจจะใช้เวลาถึง 6 ชั่วโมงได้ หากตั้งค่าตามตัวอย่างนี้ทุกประการจะใช้เวลาประมาณ 5 นาที (ทำงานโดยเพนเทียมโฟร์ 3.0 กิกะเฮิร์ทซ์ ไฮเปอร์เทรดดิ้ง) โดยเมื่อทำงานเสร็จสิ้นที่คอนโซลจะมีหน้าตาดังนี้

```
Saving bit stream in "system.bit".
Bitstream generation is complete.

Done!
```

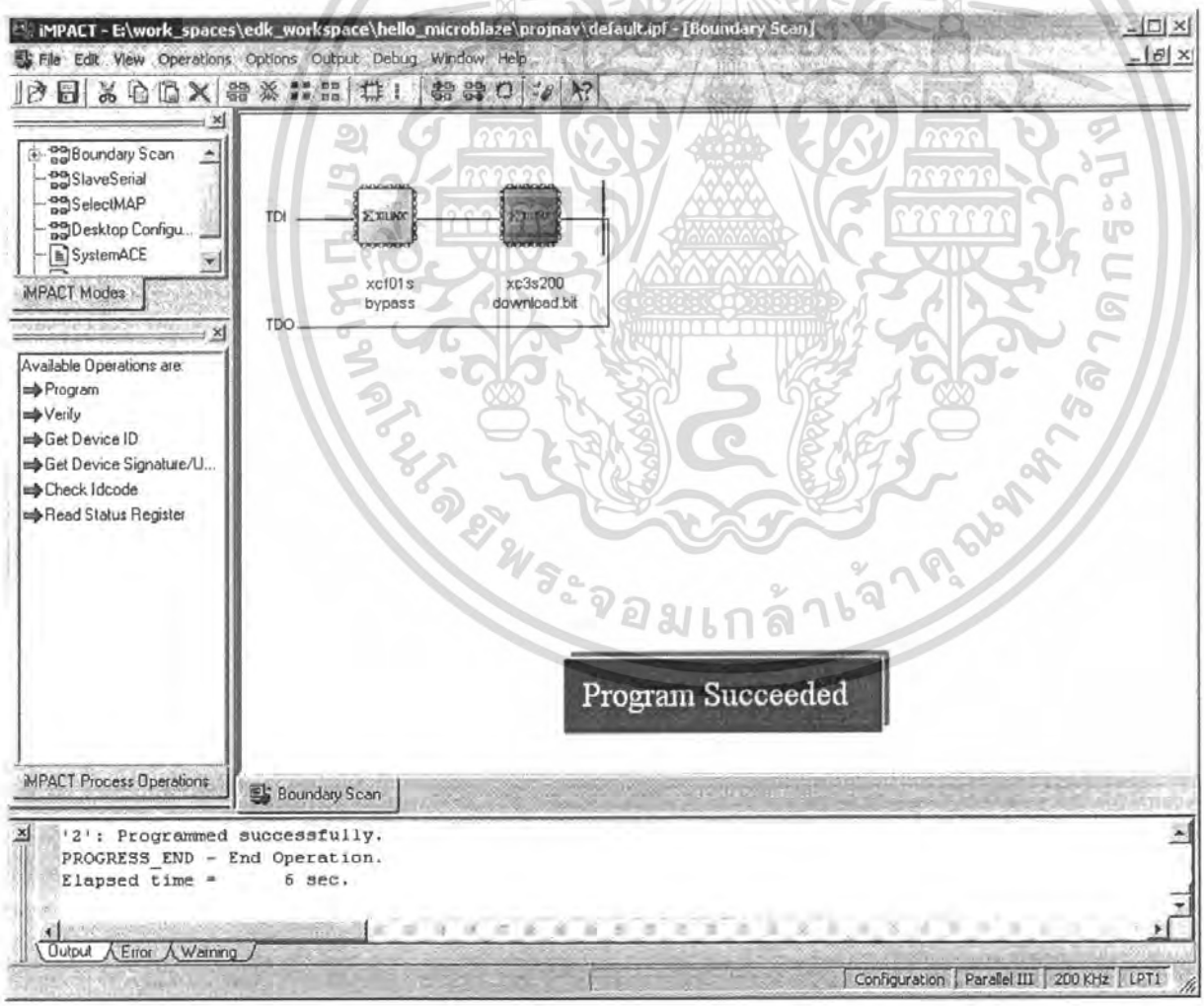
รูปที่ 4.9คอนโซลหลังจากทำการสร้างบิตสตรีมแล้ว

4. ไปที่แถบ "Applications" แล้วทำการคลิกขวาที่ "microblaze_0_bootloop" แล้วเลือก "Mark to Initialize BRAMs" และหากมีตัวอื่นที่ถูกเลือก "Mark to Initialize BRAMs" อยู่ให้ทำการเลือกออกให้หมด โดยโปรแกรมบูทลูปนี้เป็นโปรแกรมที่จะทำการบรานซ์อยู่กับที่เพื่อไม่ให้ตัวไมโครเบลสไปทำชุดคำสั่งที่ไม่พึงประสงค์เนื่องจากยังไม่มีโปรแกรมถูกโหลดลงไป

รูปที่ 4.10การเลือกให้บูทลูปไปอยู่บนบลอคแรม

5. ไปที่เมนู "Device Configuration > Update Bitstream" แล้วโปรแกรมจะทำการรวมชุดคำสั่งเริ่มต้นที่จะใส่ไปในบลอคแรม (ในที่นี้คือ "microblaze_0_bootloop") ไปไว้ในบิตสตรีมที่จะนำไปลงเอฟพีจีเอ "system.bit" ได้เป็นไฟล์ใหม่ชื่อ "download.bit"

6. เชื่อมต่อสายเจแทค(JTAG) กับบอร์ดเอฟพีจีเอจากนั้นทำการเปิดโปรแกรมอิมแพคแล้วทำการโหลดบิตสตรีมลงไปยังบอร์ดเอฟพีจีเอ



รูปที่ 4.11ดาวโหลดบิตสตรีมด้วยโปรแกรมอิมแพค

4.5 การพัฒนาซอฟแวร์บนระบบไมโครเบลสที่สร้างขึ้น

สำหรับการเขียนซอฟแวร์บนไมโครเบลสเราจะใช้โปรแกรมอีกตัวหนึ่งในชุดอีดีเค(EDK) ของไซลิ้ง ที่ชื่อว่าไซลิ้งแพลทฟอร์มสตูดิโอเอสดีเค (Xilinx Platform Studio SDK) ซึ่งเป็นโปรแกรมที่ดัดแปลงมาจากโปรแกรมอีคลิปส์(Eclipse) ของไอบีเอ็ม

1. ทำการสร้างไฟล์ไดรเวอร์ของอุปกรณ์ต่างๆในระบบโดยไปยังเมนู "Software > Generate Libraries and BSPs" ในโปรแกรมไซลิ้งแพลทฟอร์มสตูดิโอ โดยซอสไฟล์ของไดรเวอร์ต่างๆจะอยู่ในโฟลเดอร์ "microblaze_0\libsrc" ในโฟลเดอร์โปรเจค และเฮดเดอร์ไฟล์ที่จำเป็นต้องใช้ในการเขียนโปรแกรมประยุกต์จะอยู่ในโฟล์เดอร์ "microblaze_0\include"
2. เปิดโปรแกรม ไซลิ้งแพลทฟอร์มสตูดิโอเอสดีเคขึ้นมาแล้วทำการสร้างโปรเจคใหม่โดยไปที่ "File > New > Project" แล้วเลือกที่ "Managed make C project"
3. กด "Browse" แล้วเลือกไปยังไฟล์ "system.xmp" ของโปรเจคที่เราสร้างไว้

รูปที่ 4.12การตั้งค่าในเอ็กซ์พีเอสเอสดีเค

4. ตั้งชื่อโปรเจคว่า "hello" จากนั้นกด "Finish" โปรแกรมจะทำการสร้างโปรเจคขึ้นมาให้เราพร้อมไฟล์เริ่มต้นที่มีฟังก์ชัน "main" เปล่าๆ
5. ทำการอินคลูดไฟล์ "stdio.h" เข้ามายังโปรเจคและสั่งให้โปรเจคปริ้น "Hello Microblaze!" ออกมาทาง STDOUT ซึ่งถูกแมพไปยังยูอาร์ทโมดูลแล้ว

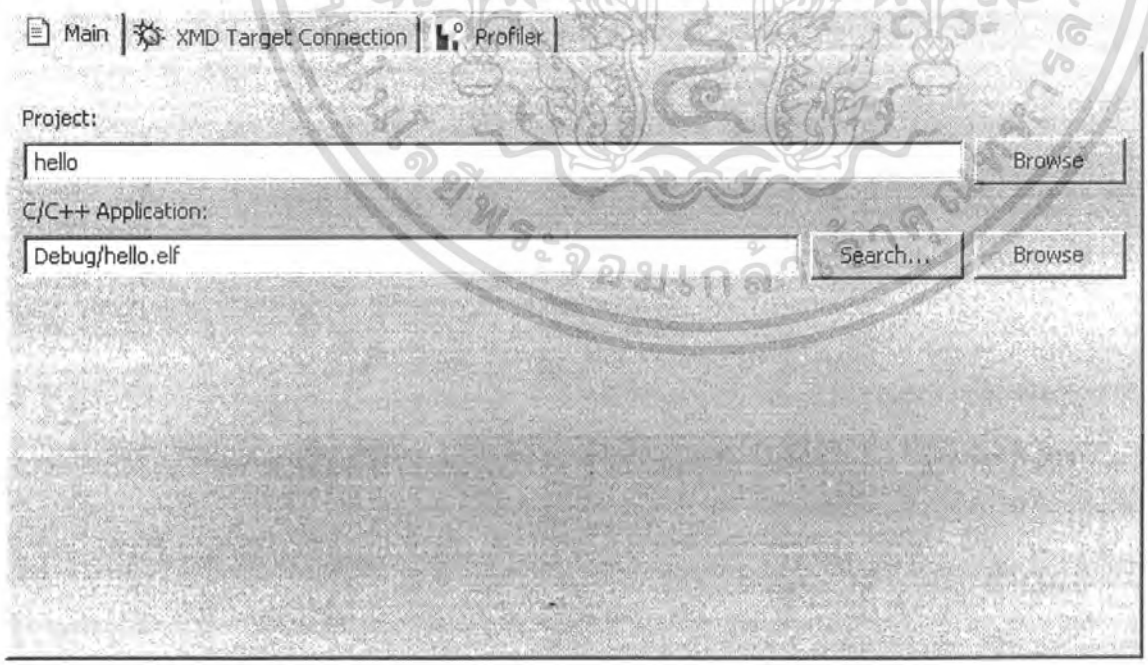
```
// Welcome to Xilinx Platform Studio SDK !
//
// This is an automatically created source file to help you get s
// To add more files, navigate to File -> New -> File
// You may delete this file if you want to use only other files f
//
#include <stdio.h>
int main()
{
    xil_printf("Hello Microblaze!\r\n");
    return 0;
}
```

รูปที่ 4.13โปรแกรม "Hello Microblaze!"

6. ทำการเซฟ โปรแกรมจะถูกคอมไพล์และลิงค์เองโดยอัตโนมัติ

7. เชื่อมต่อพอร์ตอาร์เอส232 ของบอร์ดเข้ากับพีซี จากนั้นเปิดโปรแกรมไฮเปอร์เทอร์มินอลหรือโปรแกรมอื่นที่เทียบเท่าขึ้นมาแล้วทำการตั้งค่าบอดเรทให้ตรงกับที่ตั้งไว้ในยูอาร์ทโมดูล

8. ในเอ็กซ์พีเอสเอสดีเค กดที่เมนู "Run > Run" แล้วในแถบ "Main" ตรง "Project" กด "Browse" แล้วเลือกไปยังโปรเจคของเราและตรง "C/C++ Application" กด "Search" แล้วเลือกไปยังไฟล์ .elf ในโปรเจคของเรา กด "Apply" และ "Close"



รูปที่ 4.14การตั้งค่าการรัน

9. กดปุ่ม "Run" สีเขียวตรงทูลบาร์ด้านบน โปรแกรมเอ็กซ์พีเอสเอสดีเคจะทำการส่งโปรแกรมของเรา ลงไปทำงานบนบอร์ดผ่านทางดีบักโมดูล

รูปที่ 4.15ปุ่มรัน

10. ดูผลลัพท์ในโปรแกรมเทอร์มินัลที่เปิดทิ้งไว้ ไมโครเบลสจะส่งข้อความ "Hello Microblaze!" ออกมาทางพอร์ทอาร์เอส232ดังรูป

รูปที่ 4.16ผลการทำงานของโปรแกรม

บทที่5

การใช้งานจีพีไอโอบนระบบไมโครเบลส

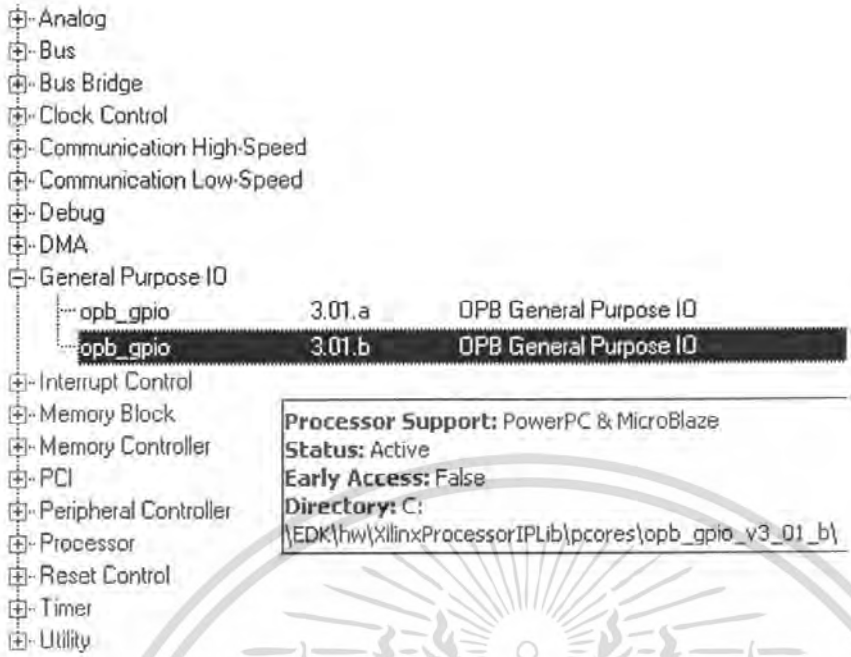
5.1 ภาพรวม

เนื้อหาในบทนี้จะเป็นเอกสารประกอบการทดลองการใช้งานจีพีไอโอ (GPIO (General Purpose Input Output)) ของระบบไมโครเบลสโดยจะทำการส่งเอาท์พุทออกไปยังหลอดแอลอีดี และทำการอ่านอินพุทจากสวิตช์

5.2 ทำการเพิ่มไอพีคอร์ลงในระบบไมโครเบลส

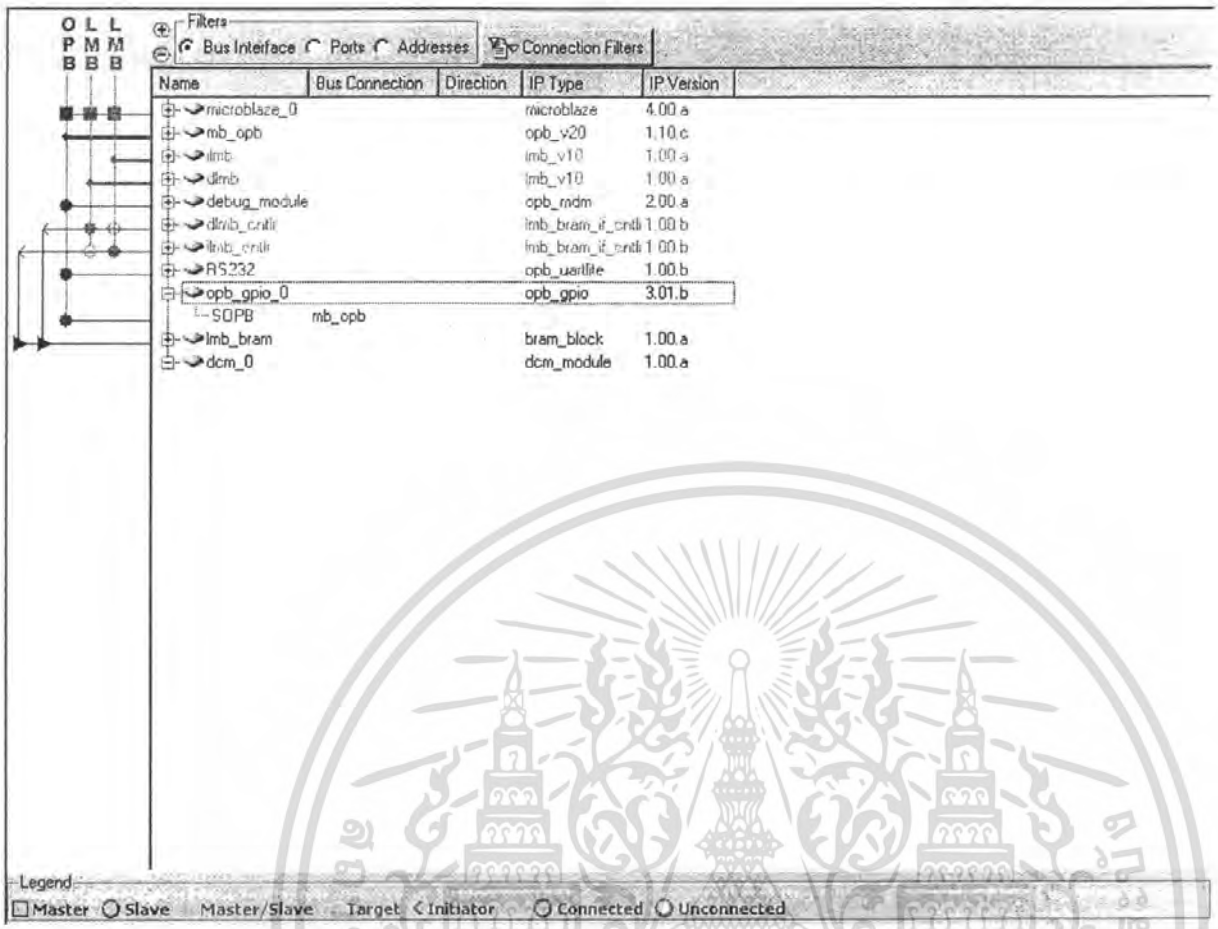
สำหรับในการทดลองนี้เราจะใช้ระบบไมโครเบลสเดิมที่สร้างไว้จากการทดลองที่แล้วมาทำการเพิ่มไอพีคอร์(IP Core (Intellectual Properties Core)) ที่เป็นจีพีไอโอเข้าไปโดยมีขั้นตอนการทำดังนี้

1. เปิดโปรแกรมไซลิ้งแพลทฟอร์มสตูดิโอขึ้นมาแล้วทำการเปิดโปรเจคที่เคยทำไว้จากการทดลองที่แล้วขึ้นมา
2. ไปที่แถบ “IP Catalog” แล้วเลือกไปที่ “General Purpose IO” แล้วทำการดับเบิ้ลคลิกที่ “opb_gpio” เพื่อทำการเพิ่มจีพีไอโอ ไอพีคอร์ ลงในระบบ



รูปที่ 5.1การเพิ่มจีพีไอโอ

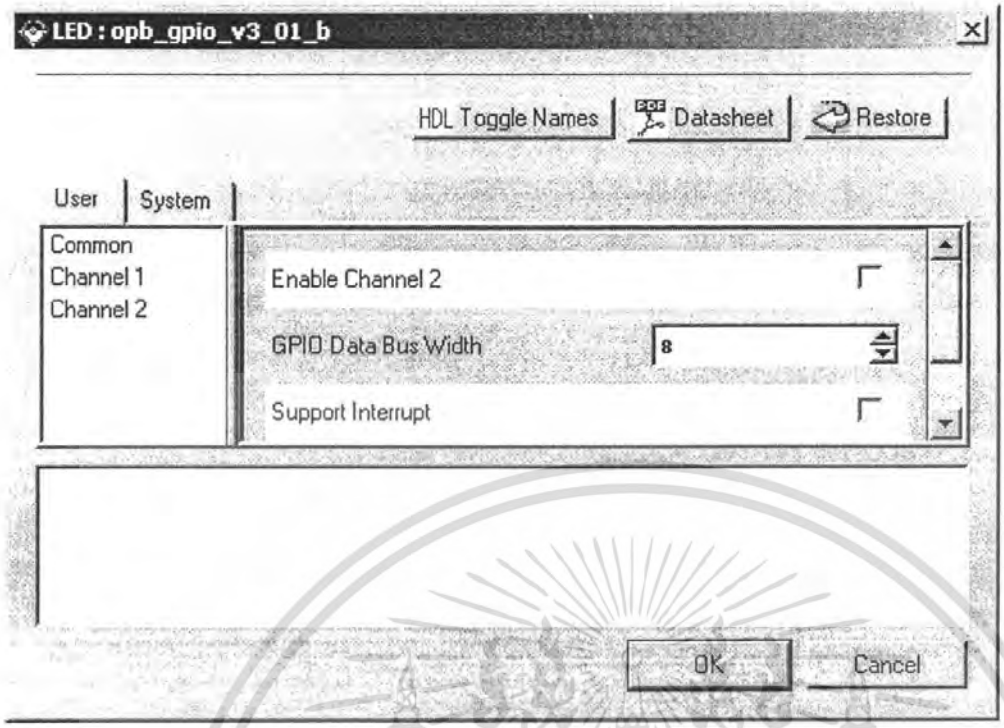
3. ทำการคลิกที่จุดสีเขียวเพื่อเชื่อมต่อ “opb_gpio_0” เข้ากับโอพีบี



รูปที่ 5.2การเชื่อมต่อจีพีไอโอเข้ากับโอพีบี

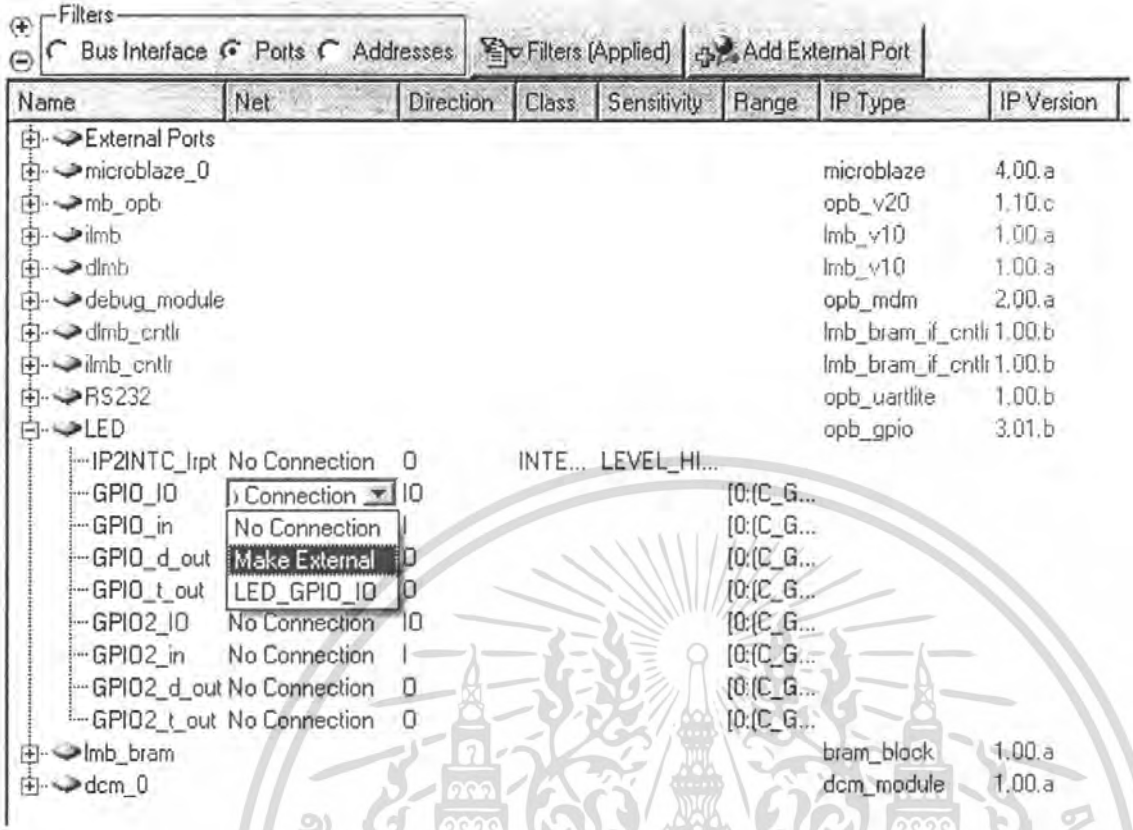
4. ทำการแก้ไขชื่อ “opb_gpio_0” เป็น “LED”

5. ดับเบิ้ลคลิกที่ “LED” แล้วทำการตั้งค่า “GPIO Data Bus Width” ให้เท่ากับจำนวนแอลอีดีบนบอร์ด ซึ่งในที่นี้เป็นแปดดวง



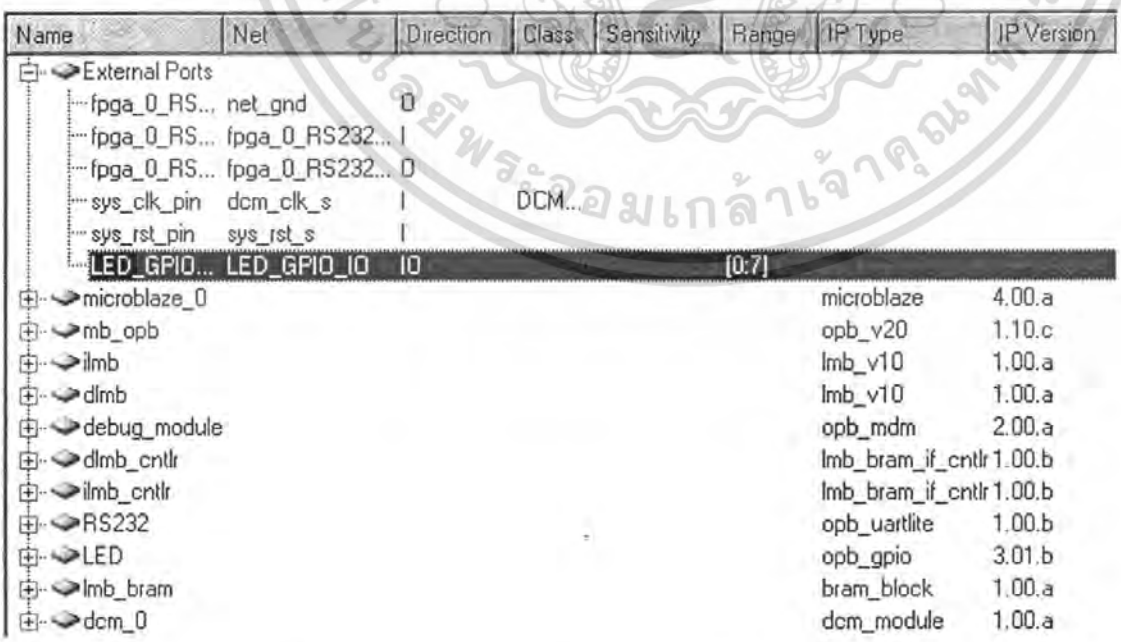
รูปที่ 5.3การแก้ความกว้างบัสของจีพีไอโอ

6. กดตรง "Filters" ไปที่ "Ports" แล้วดูที่ "LED" ตรงแถบ "Net" ที่ "GPIO_IO" เปลี่ยนจาก "No Connection" เป็น "Make External" เพื่อให้ขาของจีพีไอโอเชื่อมต่อออกไปนอกตัวชิพดังภาพ



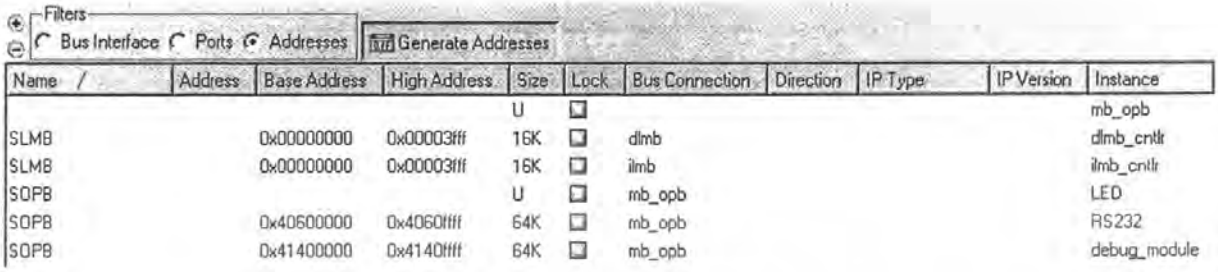
รูปที่ 5.4การทำขาพินให้เชื่อมต่อกับภายนอก

7. ไปดูที่ "External Ports" จะพบว่ามี "LED_GPIO_IO_pin" ขึ้นมาซึ่งเชื่อมต่ออยู่กับเน็ทที่ชื่อว่า "LED_GPIO_IO"



รูปที่ 5.5พอร์ทของระบบหลังจากทำการเพิ่มขาภายนอก

8. เปลี่ยน "Filters" ไปที่ "Addresses" จะพบว่าตรง "LED" จะยังไม่มีแอดเดรสถูกแมพอยู่ให้กดปุ่ม "Generate Addresses" เพื่อทำการสร้างแอดเดรสแมพใหม่



Name /	Address	Base Address	High Address	Size	Lock	Bus Connection	Direction	IP Type	IP Version	Instance
				U						mb_opb
SLMB		0x00000000	0x00003fff	16K		dlmb				dlmb_cntlr
SLMB		0x00000000	0x00003fff	16K		ilmb				ilmb_cntlr
SOPB				U		mb_opb				LED
SOPB		0x40600000	0x4060ffff	64K		mb_opb				RS232
SOPB		0x41400000	0x4140ffff	64K		mb_opb				debug_module

รูปที่ 5.6การทำแอดเดรสแมพ

9. ทำซ้ำขั้นตอนเดิมทั้งหมดในการเพิ่ม "opb_gpio" เข้าสู่ระบบอีกครั้งโดยครั้งนี้ให้ตั้งชื่อเป็น "switch" สร้างพอร์ตภายนอก และทำแอดเดรสแมพให้เรียบร้อย

รูปที่ 5.7ภาพรวมของระบบหลังจากเพิ่มสวิตช์แล้ว

10. ทำการแก้ไขยูซีเอฟไฟล์เพื่อแมพขาของสวิตช์และแอลอีดีบนบอร์ดจริงโดยใช้ชื่อเน็ทตามชื่อพอร์ตภายนอก (LED_GPIO_IO_pin และ switch_GPIO_IO_pin) โดยใส่ <0> ถึง <7> ต่อท้ายไว้เพื่อเป็นการบอกว่าเป็นขาที่เท่าไร ทำการแมพขาทั้งหมดไปยังบอร์ด

```
4   ## user constraints to this file based on customer desi
5   ##############################################################
6
7   Net sys_clk_pin LOC=P127;
8   Net sys_rst_pin LOC=P11;
9   ## System level constraints
10  Net sys_clk_pin TNM_NET = sys_clk_pin;
11  TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 40000 ps;
12  Net sys_rst_pin TIG;
13
14  ## IO Devices constraints
15
16  #### Module RS232 constraints
17
18  # Net fpga_0_RS232_req_to_send_pin LOC=;
19  Net fpga_0_RS232_RX_pin LOC=P129;
20  Net fpga_0_RS232_TX_pin LOC=P128;
21
22  Net LED_GPIO_IO_pin<0> LOC=P35;
23  Net LED_GPIO_IO_pin<1> LOC=P33;
24  Net LED_GPIO_IO_pin<2> LOC=P32;
25  Net LED_GPIO_IO_pin<3> LOC=P31;
26  Net LED_GPIO_IO_pin<4> LOC=P30;
27  Net LED_GPIO_IO_pin<5> LOC=P28;
28  Net LED_GPIO_IO_pin<6> LOC=P27;
29  Net LED_GPIO_IO_pin<7> LOC=P26;
30
31  Net switch_GPIO_IO_pin<0> LOC=P25;
32  Net switch_GPIO_IO_pin<1> LOC=P24;
33  Net switch_GPIO_IO_pin<2> LOC=P23;
34  Net switch_GPIO_IO_pin<3> LOC=P21;
35  Net switch_GPIO_IO_pin<4> LOC=P20;
36  Net switch_GPIO_IO_pin<5> LOC=P18;
37  Net switch_GPIO_IO_pin<6> LOC=P17;
38  Net switch_GPIO_IO_pin<7> LOC=P15;
39
```

รูปที่ 5.8การแมพขาบนยูซีเอฟไฟล์

11. ทำการสร้างและอัพเดทบิตสตรีม แล้วดาวโหลดบิตสตรีมลงบอร์ดตามขั้นตอนที่เคยทำในบทที่แล้ว จากนั้นทำการสร้างไลบรารี่ไฟล์ให้เรียบร้อยและเปิดตัวเอ็กซ์พีเอสเอสดีเคขึ้นมา

5.3 การเขียนโปรแกรมเพื่อควบคุมจีพีไอโอ

1. สร้างโปรเจคใหม่ในเอ็กซ์พีเอสเอสดีเคตามขั้นตอนเดิมในบทที่แล้ว หรือจะใช้โปรเจคเดิมก็ได้ ทำการอินคลูดไฟล์เพิ่มอีกสองไฟล์ดังนี้

   a. "xparameters.h" เป็นไฟล์ที่ทำการประกาศแอดเดรสแมพของแต่ละอุปกรณ์ในระบบไว้

b. "xgpio_l.h" เป็นไฟล์ที่ประกาศโปรโตไทป์สำคัญๆ ของฟังก์ชันที่เกี่ยวกับการใช้งานจีพีไอโอเอาไว้

2. สำหรับการอ่านค่าจากจีพีไอโอนั้น ก่อนอื่นเราต้องใช้ฟังก์ชัน

XGpio_mSetDataDirection(BaseAddress, Channel, DirectionMask)

ในการเขียนค่าลงไปยังรีจิสเตอร์ทิศทางของจีพีไอโอก่อน โดยพารามิเตอร์ของฟังก์ชันนี้มีดังนี้

a. BaseAddress เป็นค่าเบสแอดเดรสของโมดูลจีพีไอโอที่ต้องการใช้ สามารถดูได้จากไฟล์ "xparameters.h"

b. Channel คือแชนเนลของจีพีไอโอที่ต้องการใช้งานซึ่งจากระบบที่เราสร้างแต่ละจีพีไอโอจะมีเพียงแชนเนลเดียวคือแชนเนล1

c. DirectionMask เป็นบิตแมสค์ที่บอกว่าขาใดของจีพีไอโอจะเป็นอินพุตและขาใดจะเป็นเอาท์พุต โดยถ้าตั้งเป็น 0 จะเป็นเอาท์พุตและ 1 จะเป็นอินพุต

XGpio_mGetDataReg(BaseAddress, Channel)

เป็นฟังก์ชันที่ใช้ในการอ่านค่าขึ้นมาจากดาต้ารีจิสเตอร์ของจีพีไอโอ โดยพารามิเตอร์ของฟังก์ชันนี้นั้นมีความหมายเหมือนกับของฟังก์ชันที่ผ่านมา และจะรีเทอร์นค่าที่อยู่ภายในดาต้ารีจิสเตอร์ออกมา

3. ทำการเขียนโปรแกรมให้อ่านค่าจากสวิตช์แล้วทำการพิมพ์ออกทางยูอาร์ทโมดูลดังต่อไปนี้

```
// Welcome to Xilinx Platform Studio SDK !
//
// This is an automatically created source file to help you get s
// To add more files, navigate to File -> New -> File
// You may delete this file if you want to use only other files f
//
#include <stdio.h>
#include "xparameters.h"
#include "xgpio_l.h"

int main()
{
    unsigned char sw;
    XGpio_mSetDataDirection(XPAR_SWITCH_BASEADDR, 1, 0xFF);
    sw = XGpio_mGetDataReg(XPAR_SWITCH_BASEADDR, 1);
    xil_printf("0x%X\r\n",sw);
    return 0;
}
```

รูปที่ 5.9โปรแกรมอ่านค่าจากสวิตช์

4. ทดลองรันโปรแกรม จากนั้นลองโยกสวิตช์เปลี่ยนแล้วกดปุ่มรีเซตบนบอร์ดดูจะได้ผลลัพธ์ที่เทอร์มินอลดังนี้

รูปที่ 5.10ค่าที่พิมพ์ออกมาทางเทอร์มินอลหลังจากเปลี่ยนสวิตช์

5. สำหรับการเขียนค่าออกไปแสดงผลทางจีพีไอโอนั้นเราจำต้องใช้ฟังก์ชันเหล่านี้

   XGpio_mSetDataDirection(BaseAddress, Channel, DirectionMask)

   ใช้ในการเขียนค่าลงในรีจิสเตอร์ทิศทางของจีพีไอโอดังที่ได้กล่าวไปแล้ว

   XGpio_mSetDataReg(BaseAddress, Channel, Data)

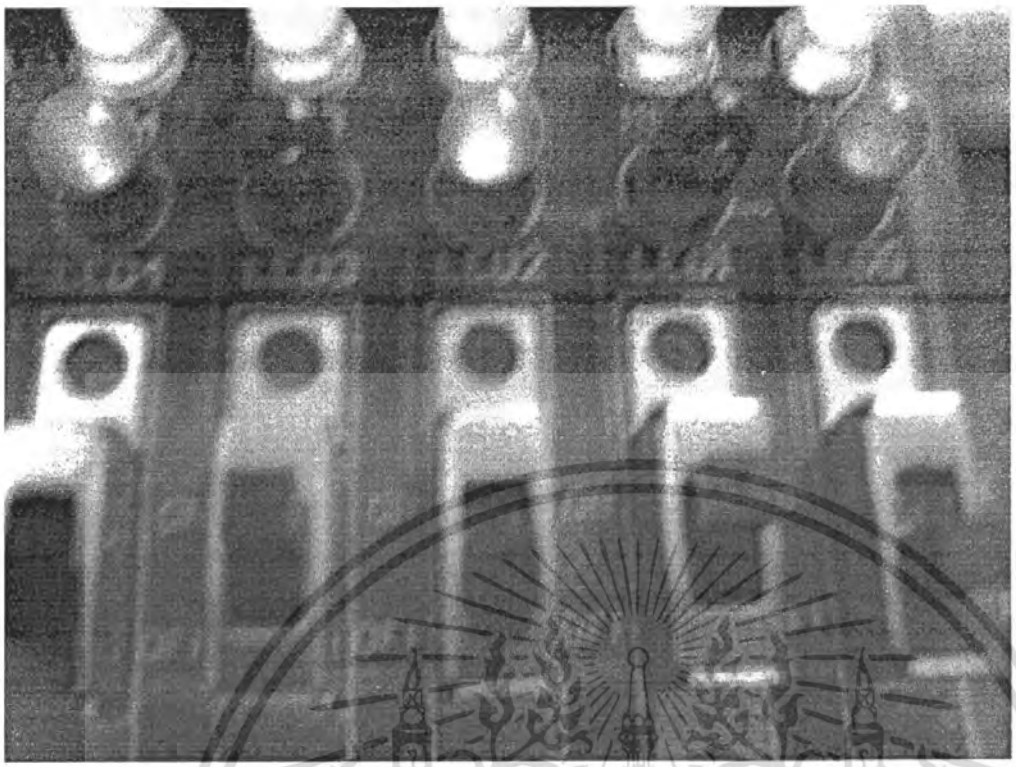
   ใช้ในการเขียนค่าลงในดาต้ารีจิสเตอร์ของจีพีไอโอ โดยพารามิเตอร์ "Data" คือข้อมูลที่เราต้องการจะเขียนออกไปยังจีพีไอโอ

6. ทำการเขียนโปรแกรมให้อ่านค่าจากสวิตช์แล้วนำมาแสดงผลที่แอลอีดีดังต่อไปนี้

```
// Welcome to Xilinx Platform Studio SDK !
//
// This is an automatically created source file to help you get s
// To add more files, navigate to File -> New -> File
// You may delete this file if you want to use only other files f
//
#include <stdio.h>
#include "xparameters.h"
#include "xgpio_l.h"

int main()
{
    unsigned char sw;
    XGpio_mSetDataDirection(XPAR_SWITCH_BASEADDR, 1, 0xFF);
    XGpio_mSetDataDirection(XPAR_LED_BASEADDR, 1, 0x00);
    while(1)
    {
        sw = XGpio_mGetDataReg(XPAR_SWITCH_BASEADDR, 1);
        XGpio_mSetDataReg(XPAR_LED_BASEADDR, 1,sw);
    }
    return 0;
}
```

รูปที่ 5.11โปรแกรมในการอ่านค่าสวิตช์และเขียนออกแอลอีดี

7. ทดลองนำโปรแกรมไปทำงานดูจะพบว่าแอลอีดีจะติดและดับตามสวิตช์ที่เปลี่ยนไปดังรูป



รูปที่ 5.12การทำงานของสวิตช์และแอดอีดี

บทที่6

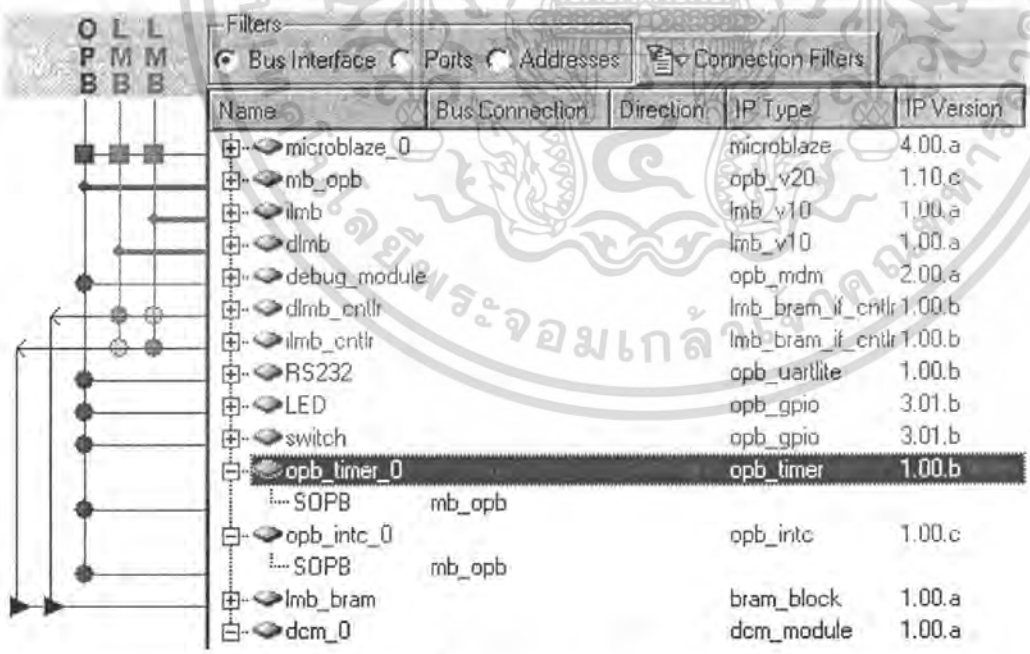
การใช้งานอินเทอรัพท์และไทม์เมอร์ในระบบไมโครเบลส

6.1 ภาพรวม

เนื้อหาในบทนี้จะเป็นเอกสารประกอบการทดลองการใช้งานไทม์เมอร์และอินเทอร์รัพท์คอนโทรลเลอร์ในระบบไมโครเบลส โดยจะทำการเพิ่มอินเทอร์รัพท์คอนโทรเลอร์และไทม์เมอร์ลงในระบบที่ได้สร้างไว้ในบทที่แล้ว

6.2 การเพิ่มไทม์เมอร์และอินเทอร์รัพท์คอนโทรลเลอร์ลงในระบบ

1. ทำการเปิดโปรแกรมเอ็กซ์พีเอสขึ้นมาแล้วทำการเปิดโปรเจคที่ได้ทำไว้ในบทที่แล้วขึ้นมา
2. ในแถบ "IP Catalog" ทำการเพิ่ม "Timer > opb_timer" และ "Interrupt Control > opb_intc" ลงในระบบ
3. คลิกที่วงกลมสีเขียวเพื่อเชื่อมต่อไอพีคอร์ทั้งสองเข้ากับโอพีบี

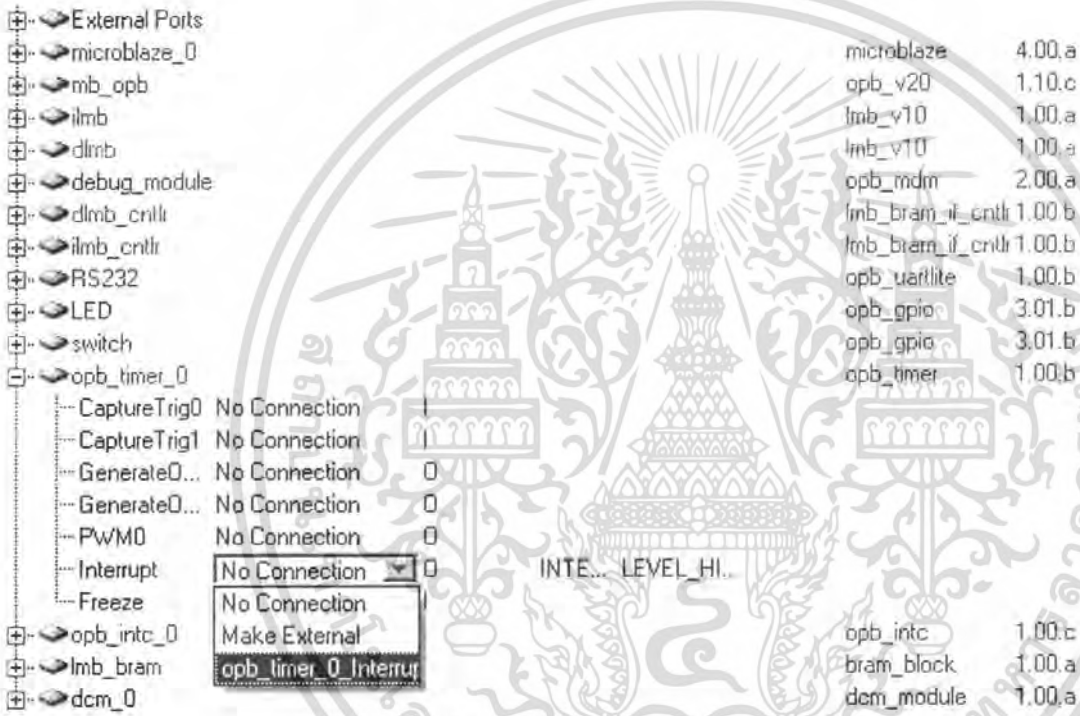


รูปที่ 6.1เพิ่มไทม์เมอร์และอินเทอร์รัพท์คอนโทรลเข้าสู่ระบบ

4. ในไทม์เมอร์หนึ่งชุดนั้นจะมีไทม์เมอร์ทั้งสิ้นสองตัว แต่ในระบบนี้เราจะใช้เพียงตัวเดียว ดังนั้นทำการดับเบิ้ลคลิกที่ "opb_timer_0" และเลือก "Only One Timer is present"

5. ดับเบิ้ลคลิกที่ "opb_intc_0" แล้วตั้งค่า "Number of Interrupt Inputs" ให้เป็น 1 เนื่องจากเราจะตอบสนองสัญญาณอินเทอร์รัพท์จากไทม์เมอร์เพียงแค่โมดูลเดียว

6. เปลี่ยน "Filters" ไปที่ "Ports" แล้วดูที่ "opb_timer_0" ตรง "Interrupt" ตั้งเน็ทไปที่ "opb_timer_0_Interrupt"



รูปที่ 6.2การตั้งค่าเน็ทของไทม์เมอร์

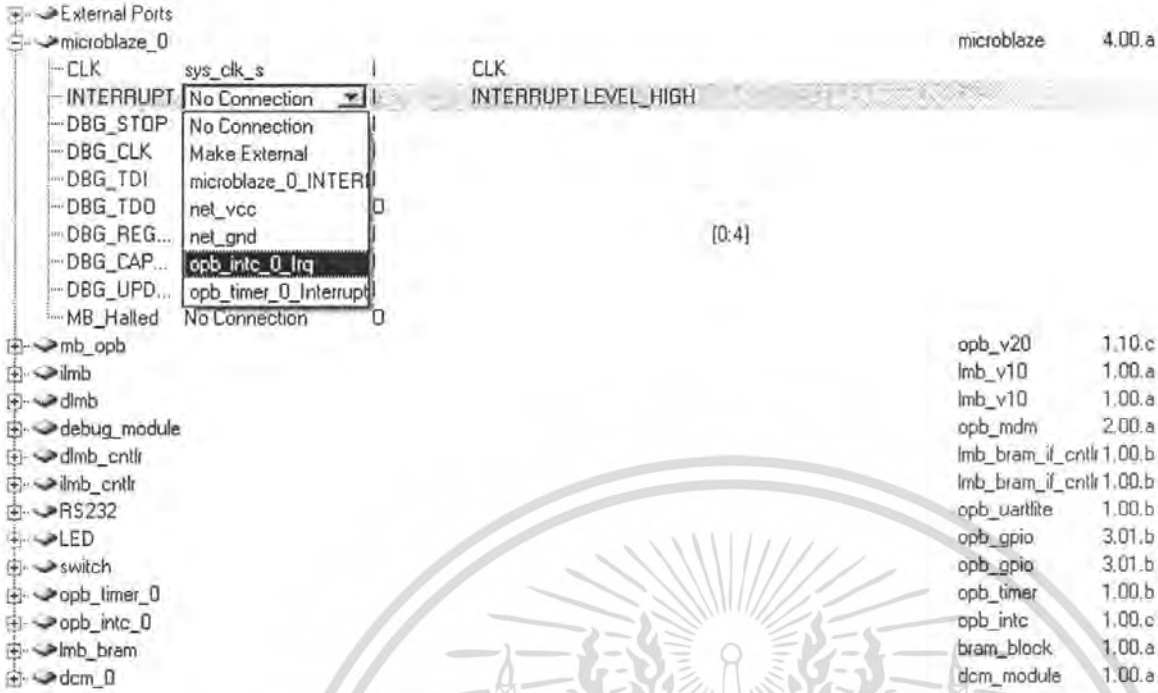
7. ดูที่ opb_intc_0 ทำการแมพ Intr ไปยังเน็ท "opb_timer_0_Interrupt" เพื่อให้สัญญาณอินเทอร์รัพท์จากไทม์เมอร์วิ่งเข้าสู่อินเทอร์รัพท์คอนโทรลเลอร์

รูปที่ 6.3การตั้งค่าเน็ท Intr ของอินเทอร์รัพท์คอนโทรล

8. ทำการแมพ "Irq" ของ "opb_intc_0" ไปยังเน็ทที่ชื่อ "opb_intc_0_Irq"

รูปที่ 6.4การตั้งค่าเน็ท Irq ของอินเทอร์รัพท์คอนโทรล

9. คูที่ "microblaze_0" ทำการแมพสัญญาณ INTERRUPT ไปที่เน็ท "opb_intc_0_Irq" เพื่อให้สัญญาณอินเทอร์รัพท์ที่ถูกสร้างจากอินเทอร์รัพท์คอนโทรลเลอร์วิ่งเข้าสู่ตัวไมโครเบลส



รูปที่ 6.5การตั้งค่าเน็ท INTERRUPT ของตัวไมโครเบลส

10. เปลี่ยน "Filters" ไปที่ "Addresses" แล้วทำการเจ็นเนอเรทแอดเดรสแมพใหม่

11. เนื่องจากการทดลองนี้ไม่มีการเปลี่ยนแปลงขาภายนอกจึงไม่ต้องแก้ไขยูซีเอฟไฟล์ใหม่ ทำการสร้างและอัพเดทบิตสตรีมแล้วดาวโหลดลงบอร์ดได้เลย

12. ทำการสร้างไลบรารี่ไฟล์ต่างๆให้พร้อมแล้วเปิดเอ็กซ์พีเอสเอสดีเคขึ้นมา

6.3 การเขียนโปรแกรมเพื่อใช้งานไทม์เมอร์ และอินเทอร์รัพท์คอนโทรลเลอร์

6.3.1 การเขียนโปรแกรมเพื่อใช้งานไทม์เมอร์

1. สำหรับการเขียนโปรแกรมกับไทม์เมอร์ เราต้องทำการอินคลูดไฟล์ต่อไปนี้เข้ามาในโปรแกรม

   a. "xparameters.h" เป็นไฟล์ที่ทำการประกาศแอดเดรสแมพของแต่ละอุปกรณ์ในระบบไว้

   b. "xtmrctr_l.h" เป็นไฟล์ที่ทำการประกาศโปรโตไทป์ของฟังก์ชันที่เกี่ยวกับไทม์เมอร์

2. ในการเขียนโปรแกรมเพื่อควบคุมไทม์เมอร์นั้นมีฟังก์ชันสำคัญๆ ดังนี้

   void XTmrCtr_mSetLoadReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber,Xuint32

   RegisterValue)

ใช้ในการเขียนค่าลงในรีจิสเตอร์เริ่มต้นของตัวไทม์เมอร์

void XTmrCtr_mSetControlStatusReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber, Xuint32 RegisterValue)

ใช้ในการเขียนค่าลงในคอนโทรลรีจิสเตอร์ของไทม์เมอร์

Xuint32 XTmrCtr_mGetTimerCounterReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)

ใช้ในการอ่านค่าจากรีจิสเตอร์เคาท์เตอร์ในตัวไทม์เมอร์

3. ทำการเขียนโปรแกรมให้ไทม์เมอร์ทำการนับลงและพิมพ์ค่าในรีจิสเตอร์ออกมาทางยูอาร์ทโมดูล ดังนี้

```
// Welcome to Xilinx Platform Studio SDK !
//
// This is an automatically created source file to help you get started.
// To add more files, navigate to File -> New -> File
// You may delete this file if you want to use only other files for your project.
//
#include <stdio.h>
#include "xparameters.h"
#include "xgpio_l.h"
#include "xtmrctr_l.h"

int main()
{
    unsigned long tmr_count;
    /* Set the number of cycles the timer counts*/
    XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_0_BASEADDR, 0, 0xFFFFFFFF);
    /* Reset the timers, and clear interrupts */
    XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_0_BASEADDR,
    0,XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );
    /* Start the timers */
    XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_0_BASEADDR,
    0,XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_AUTO_RELOAD_MASK |
    XTC_CSR_DOWN_COUNT_MASK);
    while(1)
    {
        tmr_count = XTmrCtr_mGetTimerCounterReg(XPAR_OPB_TIMER_0_BASEADDR, 0);
        xil_printf("0x%X\r\n",tmr_count);
    }
    return 0;
}
```

รูปที่ 6.6โปรแกรมไทม์เมอร์นับลง

4. ทดลองนำโปรแกรมไปทำงานดูจะได้ผลลัพธ์ดังต่อไปนี้ซึ่งแสดงว่าไทม์เมอร์กำลังทำการนับลงอยู่

รูปที่ 6.7ผลลัพธ์ของโปรแกรมไทม์เมอร์นับลง

6.3.2 การเขียนโปรแกรมเพื่อใช้งานอินเทอร์รัพท์

1. สำหรับการเขียนโปรแกรมกับอินเทอร์รัพท์ เราต้องทำการอินคลูดไฟล์ต่อไปนี้เข้ามาในโปรแกรม

   a. "xparameters.h" เป็นไฟล์ที่ทำการประกาศแอดเดรสแมพของแต่ละอุปกรณ์ในระบบไว้

   b. "xintc_l.h" เป็นไฟล์ที่ทำการประกาศโปรโตไทป์ของฟังก์ชันที่เกี่ยวกับอินเทอร์รัพท์คอนโทรลเลอร์

2. ฟังก์ชันที่สำคัญๆในการใช้งานอินเทอร์รัพท์มีดังนี้

   microblaze_enable_interrupts()

   ใช้ในการเปิดการทำงานอินเทอร์รัพท์ของไมโครเบลส

   void XIntc_RegisterHandler(Xuint32 BaseAddress, int InterruptId, XInterruptHandler Handler,

   void *CallBackRef)

   ใช้ในการรีจิสเตอร์ฟังก์ชันตอบสนองการทำงานของอินเทอร์รัพท์

   void XIntc_mMasterEnable(Xuint32 BaseAddress)

   เป็นฟังก์ชันที่ใช้ในการเปิดการทำงานของอินเทอร์รัพท์คอนโทรลเลอร์

void XIntc_mEnableIntr(Xuint32 BaseAddress, Xuint32 EnableMask)

ใช้ในการเปิดอินเทอร์รัพท์รีเควสสำหรับสัญญาณจากแต่ละแหล่ง

3. สร้างฟังก์ชันตอบสนองการทำงานของอินเทอร์รัพท์ดังนี้

```
void timer_int_handler(void * baseaddr_p)
{
    unsigned int csr;
    csr = XTmrCtr_mGetControlStatusReg(XPAR_OPB_TIMER_0_BASEADDR, 0);
    xil_printf("Interrupted!\r\n");
    XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_0_BASEADDR, 0, csr);
}
```

รูปที่ 6.8ฟังก์ชันตอบสนองการทำงานของอินเทอร์รัพท์

4. เขียนโปรแกรมเพื่อรีจิสเตอร์ฟังก์ชันตอบสนองการทำงานของอินเทอร์รัพท์และสั่งให้ไทม์เมอร์ทำงานดังนี้

```
int main()
{
    unsigned long tmr_count;
    /* Enable MicroBlaze interrupts */
    microblaze_enable_interrupts();
    /* Register the timer interrupt handler in the vector table */
    XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR,
    XPAR_OPB_INTC_0_OPB_TIMER_0_INTERRUPT_INTR,
    (XInterruptHandler)timer_int_handler,(void *)XPAR_OPB_TIMER_0_BASEADDR);
    /* Start the interrupt controller */
    XIntc_mMasterEnable(XPAR_OPB_INTC_0_BASEADDR);
    /* Enable timer interrupt requests in the interrupt controller */
    XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR, XPAR_OPB_TIMER_0_INTERRUPT_MASK)
    /* Set the number of cycles the timer counts*/
    XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_0_BASEADDR, 0, 0xFFFFFF);
    /* Reset the timers, and clear interrupts */
    XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_0_BASEADDR,
    0,XTC_CSR_INT_OCCURED_MASK | XTC_CSR_LOAD_MASK );
    /* Start the timers */
    XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_0_BASEADDR,
    0,XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_AUTO_RELOAD_MASK |
    XTC_CSR_ENABLE_INT_MASK | XTC_CSR_DOWN_COUNT_MASK);
    while(1)
    {
        tmr_count = XTmrCtr_mGetTimerCounterReg(XPAR_OPB_TIMER_0_BASEADDR, 0);
        xil_printf("0x%X\r\n",tmr_count);
    }
    return 0;
}
```

รูปที่ 6.9โปรแกรมรีจิสเตอร์ฟังก์ชันตอบสนองอินเทอร์รัพท์

5. ทดลองนำโปรแกรมไปทำงานดูจะได้ผลลัพธ์ดังต่อไปนี้ พบว่าทุกครั้งที่ไทม์เมอร์นับลงจนถึง 0 จะเกิดการอินเทอร์รัพท์ขึ้น

รูปที่ 6.10 ผลลัพธ์ของโปรแกรมตอบสนองอินเทอร์รัพท์

บทที่7
การสร้างไอพีคอร์เพื่อใช้กับระบบไมโครเบลส

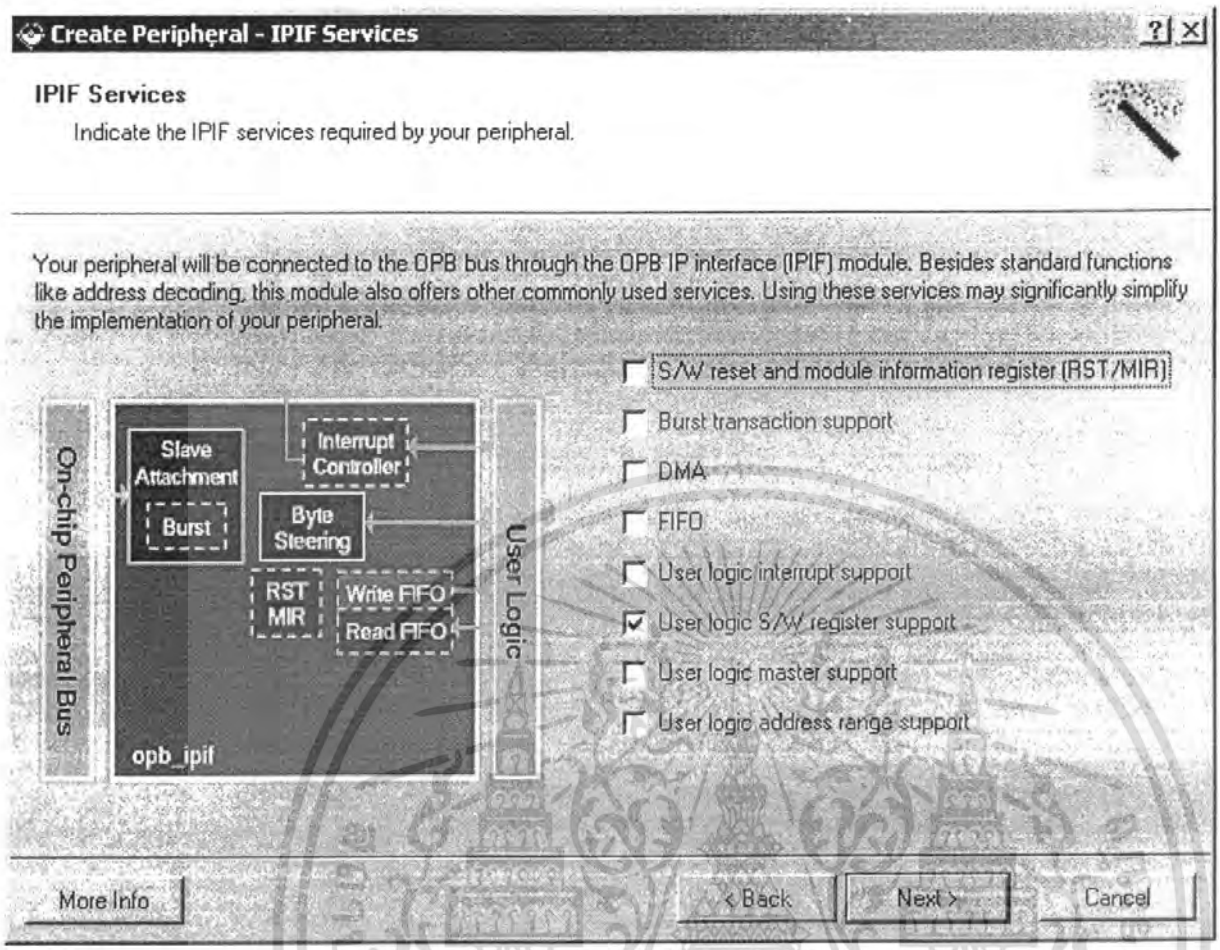
7.1 ภาพรวม

เนื้อหาในบทนี้จะเป็นเอกสารประกอบการทดลองสร้างไอพีคอร์ของตนเอง เพื่อนำไปเชื่อมต่อกับโอพีบีของไมโครเบลส โดยไอพีคอร์ที่จะสร้างเป็นตัวอย่างในบทนี้คือเซเว่นเซกเมนท์คอนโทรลเลอร์ โดยผู้ใช้งานเพียงแค่เขียนตัวเลขที่เป็นบีซีดีลงไปยังรีจิสเตอร์ของโมดูลนี้โมดูลจะจัดการนำตัวเลขนั้นออกไปแสดงผลยังเซเว่นเซกเมนท์และทำการสแกนให้โดยอัตโนมัติ

7.2 โอพีบีไอพีอินเตอร์เฟส

การเชื่อมต่อไอพีคอร์ของผู้ใช้งานเข้าสู่ระบบไมโครเบลสนั้นเราจะใช้คอร์ที่ชื่อว่าโอพีบีไอพีไอเอฟเป็นตัวช่วยในการเชื่อมต่อ ซึ่งโอพีบีไอพีไอเอฟนี้เราสามารถสร้างได้โดยวิซาร์ดในโปรแกรมเอ็กซ์พีเอสดังนี้

1. เปิดโปรแกรมเอ็กซ์พีเอสและเปิดโปรเจคเก่าขึ้นมา
2. ไปที่เมนู “Hardware > Create or Import Peripheral”
3. โปรแกรมจะแสดงหน้าวิซาร์ดขึ้นมา กด “Next”
4. เลือก “Create templates for a new peripheral”
5. เลือกที่เซฟไฟล์สำหรับเพอรีเฟอรอลของเรา
6. ทำการตั้งชื่อเพอรีเฟอรอล ซึ่งในที่นี้ขอตั้งชื่อว่า “seven_segment”
7. เลือกว่าจะให้เพอริเฟอรอลของเราเชื่อมต่อกับบัสใดในระบบ ในที่นี้เลือกเป็นโอพีบี
8. ทำการเลือกคุณสมบัติที่จะให้ไอพีไอเอฟที่กำลังจะถูกสร้างขึ้นมารองรับ ในที่นี้เลือกให้มีซอฟแวร์รีจิสเตอร์ในฝั่งยูเซอร์ลอจิคดังรูป



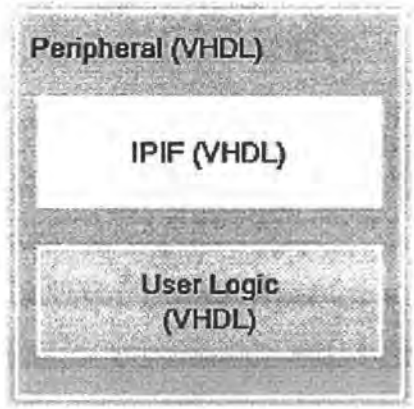
รูปที่ 7.1การตั้งค่าไอพีไอเอฟ

9. เลือกจำนวนและความกว้างของรีจิสเตอร์ภายในส่วนยูเซอร์ลอจิค ในที่นี้ตั้งไว้ที่ 1 ตัวขนาด 32 บิต

10. ทำการเลือกสัญญาณไอพีอินเตอร์คอนเนคที่จะทำการส่งต่อไปยังส่วนยูเซอร์ลอจิค ซึ่งในหน้านี้เราไม่จำเป็นต้องทำการตั้งค่าเพิ่มเติม สามารถใช้ค่าเริ่มต้น ทำการกด "Next" ข้ามไปได้เลย

11. ในหน้าต่อไปจะเป็นเกี่ยวกับซิมูเลชันซัพพอร์ท กด "Next" ข้ามไปได้เลย

12. กด "Next" ไปจนสุดแล้วกด "Finish" โปรแกรมจะทำการสร้างโค๊ดภาษาวีเอชดีแอลสำหรับเชื่อมต่อคอร์ของเรากับโอพีบีขึ้นมาให้ในลักษณะตามรูปต่อไปนี้



รูปที่ 7.2การเชื่อมต่อไอพีที่สร้างขึ้นกับโอพีบี

13. สิ่งที่เราจำเป็นต้องทำต่อไปคือเขียนโค๊ดภาษาวีเอชดีแอลในส่วนยูเซอร์ลอจิกเพื่อให้คอร์ที่เราสร้างขึ้นมานั้นทำงานตามที่เราต้องการ

7.3 การสร้างยูเซอร์ไอพีคอร์

1. หลังจากทำการสร้างไอพีไอเอฟทางวิซาร์ดแล้ว ไปดูที่โฟล์เดอร์ "pcores" ในโปรเจคของเรา จะพบว่ามี "seven_segment_v1_00_a" อยู่

2. กดเข้าไปภายในโฟลเดอร์ จะมีโฟลเดอร์ที่ชื่อ "hdl\vhdl" อยู่ ด้านในจะมีไฟล์อยู่ทั้งสิ้นสองไฟล์ ได้แก่ "seven_segment.vhd" ซึ่งเป็นโค๊ดภาษาวีเอชดีแอลที่เป็นทอปเลเวลของคอร์ของเรา และไฟล์ "user_logic.vhd" ซึ่งเป็นโค๊ดภาษาวีเอชดีแอลในส่วนที่เราต้องทำการแก้ไข

3. เปิดไฟล์ "user_logic.vhd" ขึ้นมา ด้านในจะมีโค๊ดส่วนหนึ่งที่วิซาร์ดทำการสร้างให้เราเรียบร้อยแล้ว ทำการแก้ไขดังต่อไปนี้

4. ส่วน "entity" ในส่วน "port" ทำการเพิ่มขาสัญญาณต่อไปนี้ลงไป

```
 97:  port
 98:  (
 99:    -- ADD USER PORTS BELOW THIS LINE ------------------
100:    --USER ports added here
101:    SevenSeg                       : out   std_logic_vector (0 to 7);
102:    SevenSeg_Dec                   : out   std_logic_vector (0 to 3);
103:
104:    -- ADD USER PORTS ABOVE THIS LINE ------------------
105:
```

รูปที่ 7.3การเพิ่มพอร์ทในยูเซอร์ลอจิก

5. ในส่วน "architecture" ทำการเพิ่ม "signal" ต่อไปนี้ลงไป

```
126:
127: architecture IMP of user_logic is
128:
129:   --USER signal declarations added here, as needed for user logic
130:   signal rCntDly                    : std_logic_vector(31 downto 0);
131:   signal r7Seg                      : std_logic_vector(0 to 7);
132:   signal rDec7Seg                   : std_logic_vector(0 to 3);
133:
```

รูปที่ 7.4 "signal" ในส่วน "architecture"

6. เพิ่มโค๊ตการทำงานต่อไปนี้ลงไปด้านใต้ “begin” ในส่วนของ “architechture”

```
146:   --USER logic implementation added here
147:   SevenSeg <=       r7Seg;
148:   SevenSeg_Dec <=   rDec7Seg;
149:
150:   u_rCntDly : Process (Bus2IP_Reset, Bus2IP_Clk) Is
151:   Begin
152:     if ( Bus2IP_Reset ='1' ) then
153:         rCntDly <= (others => '0');
154:
155:     elsif ( rising_edge(Bus2IP_Clk) ) then
156:         if ( rCntDly = x"0002625A" )then
157:             rCntDly <= (others => '0');
158:         else
159:             rCntDly <= rCntDly + x"00000001";
160:         end if;
161:    end if;
162:   End Process u_rCntDly;
163:
164:   u_r7Seg : Process (Bus2IP_Reset, Bus2IP_Clk) Is
165:   Begin
166:     if ( Bus2IP_Reset='1' ) then
167:         r7Seg <= (others=>'0');
168:
169:     elsif ( rising_edge(Bus2IP_Clk) ) then
170:         case rDec7Seg is
171:             when "1110" =>
172:                 r7Seg <= to_char(conv_integer(slv_reg0(28 to 31)));
173:             when "1101" =>
174:                 r7Seg <= to_char(conv_integer(slv_reg0(24 to 27)));
175:             when "1011" =>
176:                 r7Seg <= to_char(conv_integer(slv_reg0(20 to 23)));
177:             when "0111" =>
178:                 r7Seg <= to_char(conv_integer(slv_reg0(16 to 19)));
179:             when others =>
180:                 r7Seg <= (others=>'0');
181:         end case;
182:     end if;
183:     End Process u_r7Seg;
184:
185:     u_rDec7Seg : Process (Bus2IP_Reset, Bus2IP_Clk) Is
186:     Begin
187:         if ( Bus2IP_Reset='1' ) then
188:             rDec7Seg <= "1110";
189:
190:         elsif ( rising_edge(Bus2IP_Clk) ) then
191:             if ( rCntDly=x"2625A" ) then
192:                 rDec7Seg <= rDec7Seg(1 to 3) & rDec7Seg(0);
193:             else
194:                 rDec7Seg <= rDec7Seg;
195:             end if;
196:         end if;
197:     End Process u_rDec7Seg;
```

รูปที่ 7.5โค๊ดการทำงานของเซเว่นเซกเมนท์คอนโทรลเลอร์

7. ทำการเพิ่มส่วนของการแปลงฟอนท์ต่อไปนี้ลงในส่วนของ “architecture”

```
123: ------------------------------------------------------------------------------
124: -- Architecture section
125: ------------------------------------------------------------------------------
126:
127: architecture IMP of user_logic is
128:     ---------------------------------------
129:     -- convert a std_logic value to a character
130:     ---------------------------------------
131:     type stdlogic_to_digit_fonts_t is array(0 to 9) of std_logic_vector(0 to 7);
132:     constant to_char : stdlogic_to_digit_fonts_t := (
133:         0 => "11111100",
134:         1 => "01100000",
135:         2 => "11011010",
136:         3 => "11110010",
137:         4 => "01100110",
138:         5 => "10110110",
139:         6 => "10111110",
140:         7 => "11100000",
141:         8 => "11111110",
142:         9 => "11110110"
143:         );
144:
145:   --USER signal declarations added here, as needed for user logic
146:   signal rCntDly                 : std_logic_vector(31 downto 0);
147:   signal r7Seg                   : std_logic_vector(0 to 7);
148:   signal rDec7Seg                : std_logic_vector(0 to 3);
149:
```

รูปที่ 7.6การแปลงฟอนท์ของเซเว่นเซกเมนท์

8. ทำการเซฟและปิดไฟล์ จากนั้นทำการเปิดไฟล์ “seven_segment.vhd” ซึ่งเป็นทอปเลเวลขึ้นมาทำการแก้ไข

9. เพิ่มพอร์ทต่อไปนี้ในส่วนของ “entity”

```
116:   port
117:   (
118:     -- ADD USER PORTS BELOW THIS LINE ------------------
119:     --USER ports added here
120:     SevenSeg                       : out std_logic_vector (0 to 7);
121:     SevenSeg_Dec                   : out std_logic_vector (0 to 3);
122:
123:     -- ADD USER PORTS ABOVE THIS LINE ------------------
```

รูปที่ 7.7พอร์ทที่เพิ่มในส่วนทอปเลเวล

10. เพิ่มพอร์ทแมพต่อไปนี้ในส่วนการสร้างอินสแตนท์ของยูเซอร์ลอจิค

```
375:  ------------------------------------------
376:  -- instantiate the User Logic
377:  ------------------------------------------
378:  USER_LOGIC_I : entity seven_segment_v1_00_a.user_logic
379:    generic map
380:    (
381:      -- MAP USER GENERICS BELOW THIS LINE ---------------
382:      --USER generics mapped here
383:      -- MAP USER GENERICS ABOVE THIS LINE ---------------
384:
385:      C_DWIDTH                       => USER_DWIDTH,
386:      C_NUM_CE                       => USER_NUM_CE
387:    )
388:    port map
389:    (
390:      -- MAP USER PORTS BELOW THIS LINE ------------------
391:      --USER ports mapped here
392:      SevenSeg                       => SevenSeg,
393:      SevenSeg_Dec                   => SevenSeg_Dec,
394:      -- MAP USER PORTS ABOVE THIS LINE ------------------
395:
```

รูปที่ 7.8การทำพอร์ทแมพที่ทอปเลเวล

11. ทำการเซฟและปิดไฟล์

12. หากเรานำคอร์ที่เราสร้างเสร็จนี้มาใช้ในโปรแกรมเอ็กซ์พีเอสเลย เราจะพบว่าจะยังไม่มีพอร์ทที่เราเพิ่งจะสร้างเพิ่มที่ทอปเลเวล เนื่องจากเรายังไม่ได้ทำการแก้ไขเอ็มพีดีไฟล์ (Microprocessor Peripheral Description file)

13. ไปยังโฟลเดอร์ "pcores\seven_segment_v1_00_a\data" ภายในโฟลเดอร์ของโปรเจคของเรา แล้วทำการเปิดเอ็มพีดีไฟล์ขึ้นมา

14. ทำการเพิ่มพอร์ททั้งสองที่เราสร้างขึ้นมาลงในเอ็มพีดีไฟล์ดังนี้ จากนั้นทำการเซฟ

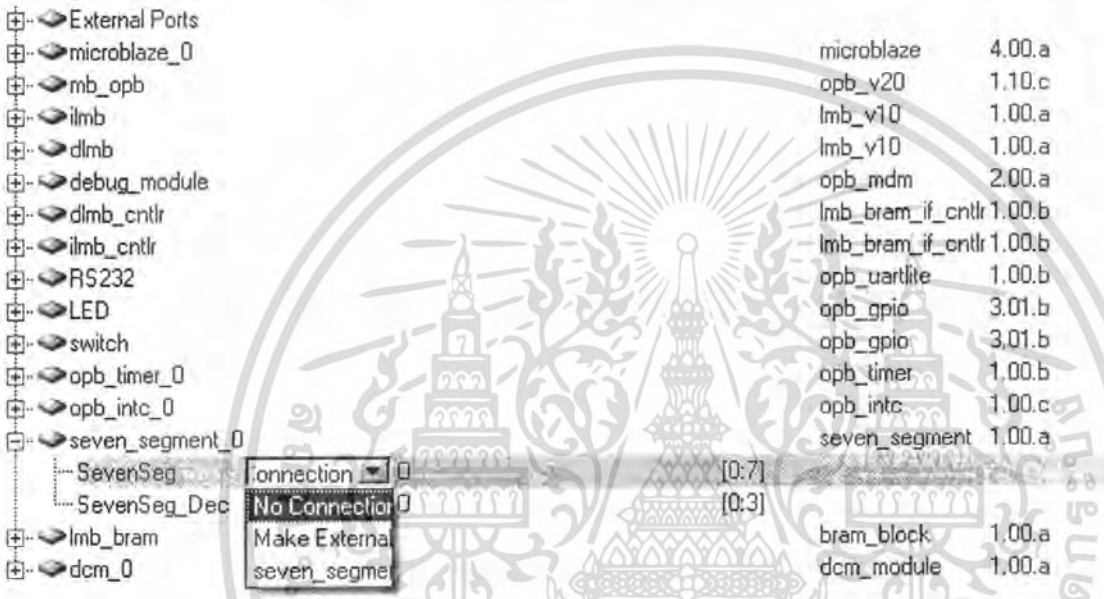
```
43
44  PORT SevenSeg = "", DIR = O , VEC = [0:7]
45  PORT SevenSeg_Dec = "", DIR = O , VEC = [0:3]
46
47  END
48
```

รูปที่ 7.9เพิ่มพอร์ทลงในเอ็มพีดีไฟล์

15. ดูที่โปรแกรมเอ็กซ์พีเอสตรงแถบ “IP Catalog” ที่ “Project Repository” จะมีคอร์ “seven_segment” อยู่ทำการเพิ่มคอร์ดังกล่าวเข้าสู่ระบบ และเชื่อมต่อกับโอพีบีให้เรียบร้อย

16. เปลี่ยน “Filters” ไปที่ “Ports” แล้วดูที่ “seven_segment” ถ้าหากยังไม่มีพอร์ทใดๆ ให้ทำการปิดและเปิดโปรแกรมเอ็กซ์พีเอสอีกครั้งหนึ่ง

17. ทำการแมพขาพอร์ททั้งคู่ออกมาเป็นขาภายนอกและแก้ไขยูซีเอฟไฟล์ให้เรียบร้อย



รูปที่ 7.10การแมพขาอุปกรณ์ออกมาเป็นขาภายนอก

18. ทำการแมพแอดเดรสใหม่ให้เรียบร้อยในหน้า “Filters” ที่เป็น “Addresses”

19. ทำการสร้างและอัพเดทบิตสตรีมแล้วทำการดาวโหลดลงบอร์ดให้เรียบร้อย

20. สั่งให้เอ็กซ์พีเอสทำการสร้างไฟล์ไลบรารี่ต่างๆ โดย “LibGen” ตามขั้นตอนที่เคยทำ จากนั้นเปิดโปรแกรมเอ็กซ์พีเอสเอสดีเคขึ้นมาเพื่อทำซอฟแวร์ต่อไป

7.4 การเขียนซอฟท์แวร์เพื่อสั่งงานไอพีคอร์ที่สร้างขึ้นเอง

1. หลังจากที่ทำการรัน “LibGen” จากโปรแกรมเอ็กซ์พีเอสแล้วจะมีไฟล์เฮดเดอร์ใหม่ที่ประกาศฟังก์ชันโปรโตไทป์สำหรับการควบคุมไอพีคอร์ของเราขึ้นมา ชื่อว่า “seven_segment.h” ทำการอินคลูดไฟล์ดังกล่าวเข้ามาให้โปรแกรมของเรา

2. เราจะใช้ฟังก์ชันต่อไปนี้ในการเขียนค่าลงในรีจิสเตอร์ของไอพีคอร์ของเรา

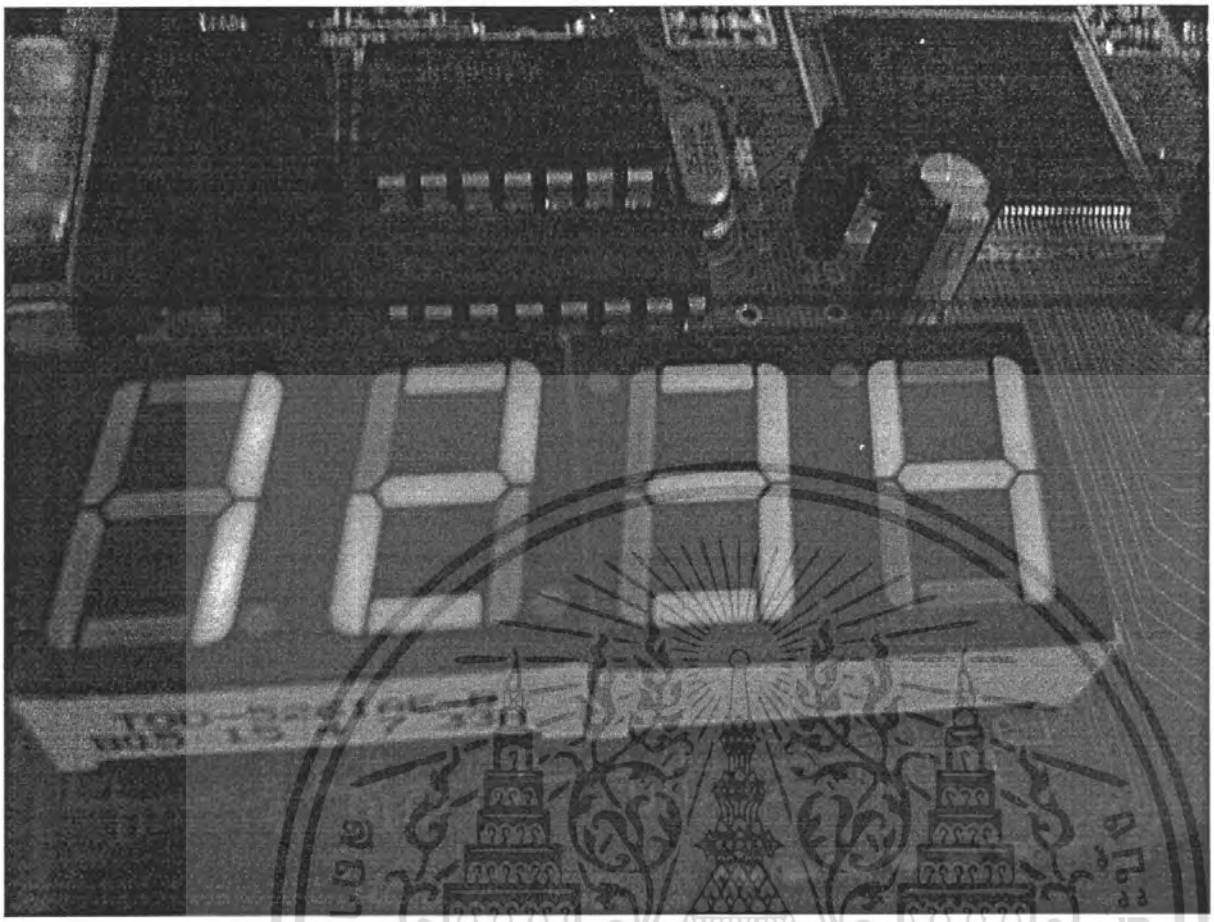
   SEVEN_SEGMENT_mWriteSlaveReg0(BaseAddress, Value)

3. ทดลองเขียนโปรแกรมให้ทำการเขียนตัวเลขบีซีดี "0x1234" ลงไปในรีจิสเตอร์ของเซเว่นเซกเมนท์โมดูลดังต่อไปนี้

```
// Welcome to Xilinx Platform Studio SDK !
//
// This is an automatically created source file to help you get started.
// To add more files, navigate to File -> New -> File
// You may delete this file if you want to use only other files for your project
//
#include <stdio.h>
#include "xparameters.h"
#include "seven_segment.h"

int main()
{
    SEVEN_SEGMENT_mWriteSlaveReg0(XPAR_SEVEN_SEGMENT_0_BASEADDR, 0x1234);
    return 0;
}
```

รูปที่ 7.11โปรแกรมสั่งงานเซเว่นเซกเมนท์โมดูล

4. ทดลองนำโปรแกรมไปทำงานดูจะพบว่ามีตัวเลข "1234" ติดที่เซเว่นเซกเมนท์บนบอร์ดทดลองดังรูป



รูปที่ 7.12 ผลลัพธ์ที่เซเว่นเซกเมนท์

บทที่8

การนำไมโครซีโอเอสทูว์มาใช้งานบนระบบไมโครเบลส

8.1 ภาพรวม

เนื้อหาในบทนี้จะอธิบายถึงวิธีการนำเรียลไทม์โอเอสที่ชื่อว่าไมโครซีโอเอสทูว์มาทำงานบนระบบไมโครเบลส รวมถึงวิธีการเขียนโปรแกรมบนไมโครซีโอเอสทูว์เบื้องต้น

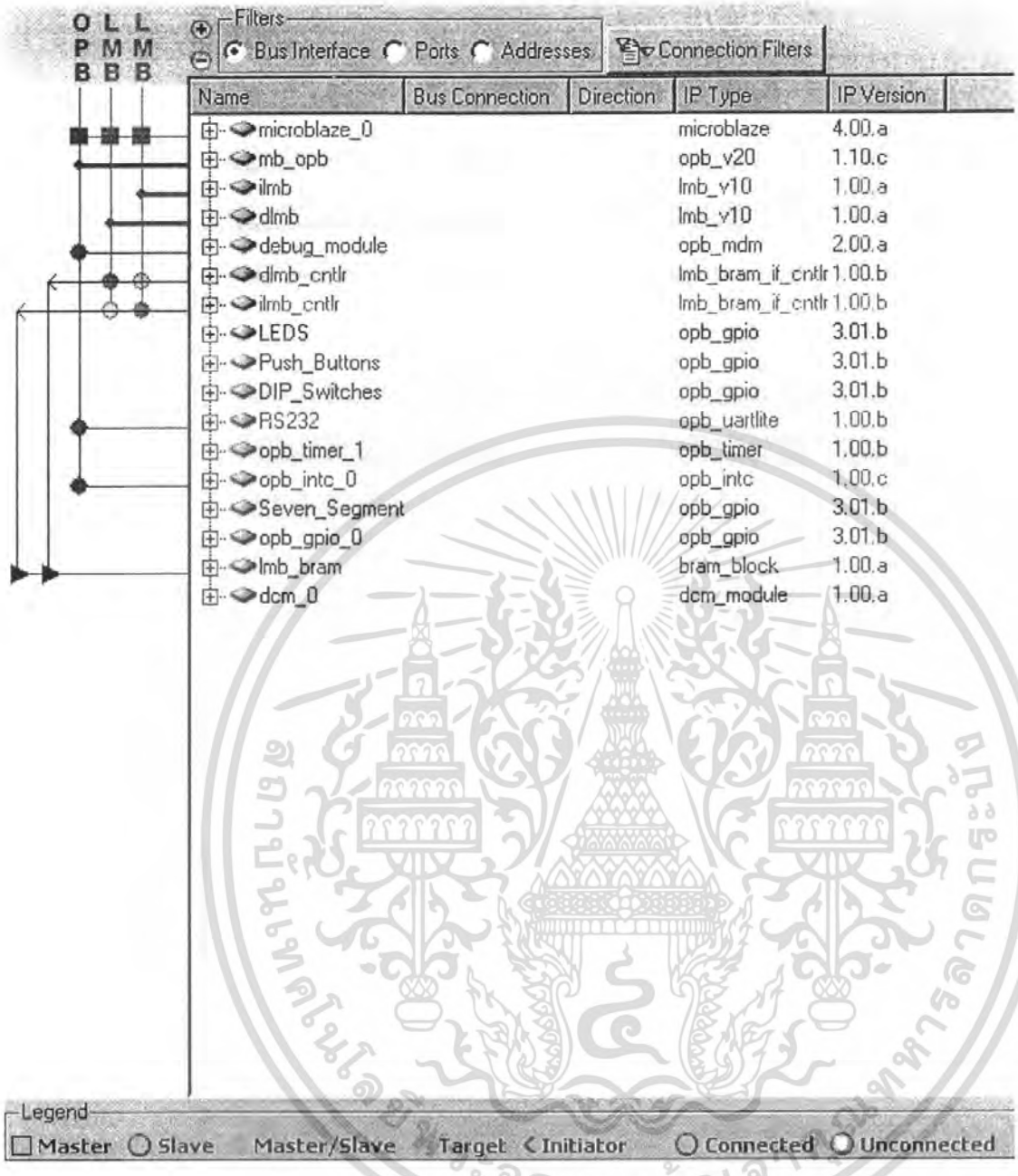
8.2 ไมโครซีโอเอสทูว์

สำหรับซอสโค๊ดของไมโครซีโอเอสทูว์นั้นถูกแบ่งออกเป็นสองส่วนได้แก่ ซอส และพอร์ท โดยซอสนั้นคือส่วนซอสโค๊ดของตัวโอเอสที่ไม่ขึ้นกับสถาปัตยกรรมของโพรเซสเซอร์ที่จะนำไปใช้งาน ส่วนพอร์ทนั้นคือซอสโค๊ดส่วนที่จะเชื่อมต่อโอเอสเข้ากับสถาปัตยกรรมของไมโครโพรเซสเซอร์ตัวที่จะใช้งาน

8.2.1 เตรียมความพร้อมของระบบก่อนใช้งานไมโครซีโอเอสทูว์

สำหรับระบบไมโครเบลสที่จะใช้งานไมโครซีโอเอสทูว์นั้นจำเป็นที่จะต้องมีไทม์เมอร์อย่างน้อยหนึ่งตัวที่ซัพพอร์ทการเกิดอินเทอร์รัพท์ ซึ่งในการทดลองนี้เราจะเปลี่ยนบอร์ดทดลองไปใช้บอร์ดที่มีชิพเวอร์เท็กซ์ทูว์เนื่องจากขนาดของบลอคแรมในเอฟพีจีเอของบอร์ดเดิมนั้นไม่เพียงพอต่อการลงไมโครซีโอเอสทูว์ สำหรับขั้นตอนการเตรียมระบบนั้นมีดังนี้

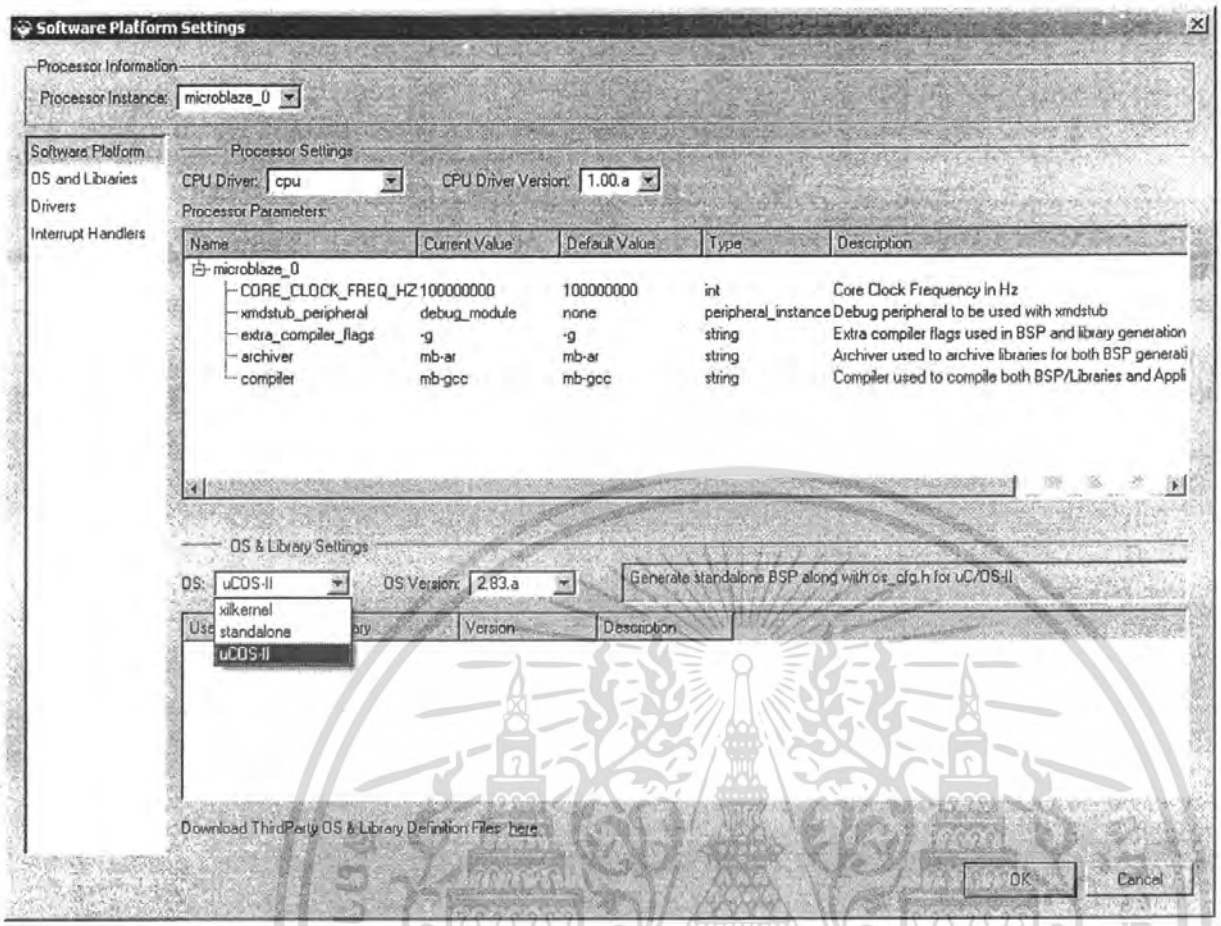
1. ทำการดาวโหลดซอสโคดของตัวไมโครซีโอเอสทูว์และพอร์ทของไมโครเบลสจากเว็บไซต์ http://www.micrium.com
2. ทำการคลายการบีบอัดไฟล์ทั้งหมดไปไว้ที่ไดรฟ์ซี
3. เปิดโปรแกรมเอ็กซ์พีเอสขึ้นมาและทำการสร้างระบบไมโครเบลสโดยใช้เบสซิสเต็มบิลเดอร์วิซาร์ดให้ระบบมีลักษณะดังนี้



รูปที่ 8.1ระบบที่จะนำไมโครซีโอเอสทูว์เข้ามาทำงาน

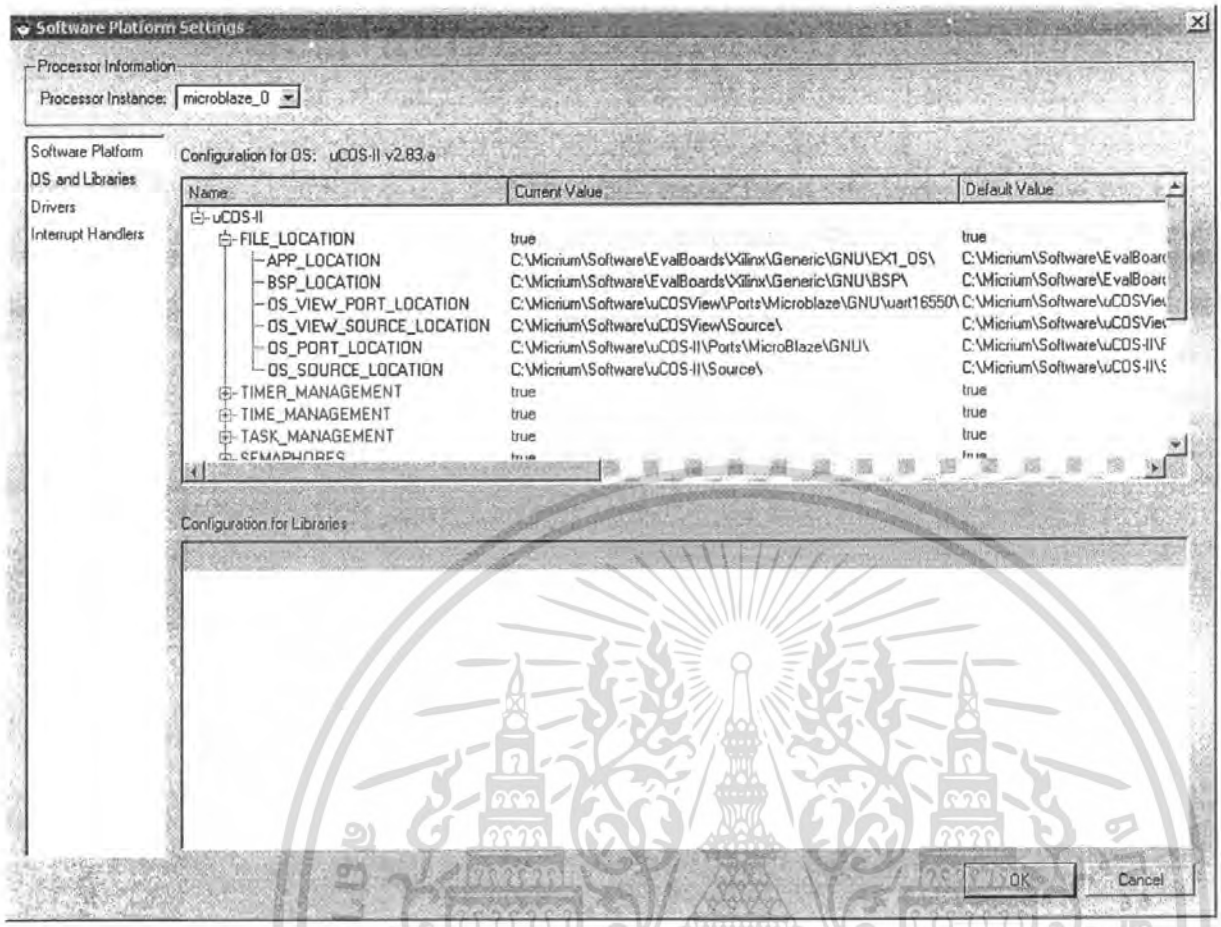
4. เปิดไปที่เมนู “Software > Software Platform Settings”

5. ในหน้าแรกด้านล่างตรง “OS” ให้เลือกเป็น “uCOS-II” ดังรูป



รูปที่ 8.2เลือก "OS" เป็น "uCOS-II"

6. ไปที่แถบ “OS and Libraries” ทำการแก้ “FILE_LOCATION” ให้ตรงตามภาพ



รูปที่ 8.3การแก้ไข"FILE_LOCATION"

7. ที่ “TASK_MANAGEMENT” แก้ “OS_TASK_CREATE_EN” ให้เป็น 1 เพื่อเปิดการใช้งานฟังก์ชัน “OSTaskCreate”

8. ที่ “MISCELLANEOUS” แก้ “OS_TICKS_PER_SEC” ให้เป็น 1000 เพื่อให้ความละเอียดของโอเอสทิคสูงขึ้นเนื่องจากซีพียูของเราทำงานที่ความถี่ค่อนข้างสูง

9. แก้ “OS_MAX_TASKS” จาก 5 ให้เป็น 20 เพื่อที่จะได้สร้างทาส์กได้มากกว่า 5 ทาส์ก

10. แก้ “OS_LOWEST_PRIO” จาก 10 ให้เป็น 30 เนื่องจากปริมาณทาส์กของเรามีมากขึ้น

11. แก้ “OS_MAX_EVENTS” จาก 2 ให้กลายเป็น 10 เพื่อที่เราจะได้สามารถสร้างอีเวนท์ได้มากขึ้น

12. ที่ “BSP_OPTIONS” ทำการแก้ “stdout” และ “stdin” ให้เป็น “debug_module” เพื่อให้ปริ๊นข้อความผ่านทางดีบักโมดูลได้

13. กด “OK”

14. ทำการเปิดไฟล์ "C:\MICRIUM\Software\EvalBoards\Xilinx\Generic\GNU\BSP\bsp.c" ขึ้นมาทำการแก้ไข

15. ทำการแก้ไข "BSP_INTC_DEVICE_ID, BSP_INTC_TIMER1_ID, BSP_INTC_ADDR, BSP_TIMER1_ADDR" ให้ตรงกับอุปกรณ์ของเราที่ประกาศไว้ในไฟล์ "xparameters.h" ไม่เช่นนั้นโอเอสจะไม่สามารถทำงานได้

```
23 /*
24 *********************************************************************************************************
25 *                                              CONSTANTS
26 *********************************************************************************************************
27 */
28
29  #define  BSP_INTC_DEVICE_ID      XPAR_OPB_INTC_0_DEVICE_ID
30  #define  BSP_INTC_TIMER1_ID      XPAR_OPB_INTC_0_OPB_TIMER_1_INTERRUPT_INTR
31
32  #define  BSP_INTC_ADDR           XPAR_OPB_INTC_0_BASEADDR
33  #define  BSP_TIMER1_ADDR         XPAR_OPB_TIMER_1_BASEADDR
34
35  #define  BSP_GPIO_ADDR           XPAR_LEDS_BASEADDR
36
37  #define  BSP_TMR_VAL             (XPAR_CPU_CORE_CLOCK_FREQ_HZ / OS_TICKS_PER_SEC)
38
```

รูปที่ 8.4แก้ไขไฟล์ "bsp.c"

16. ทำการสร้างและอัพเดทบิตสตรีม และดาวโหลดลงบอร์ดพร้อมทั้งทำ "LibGen" ให้เรียบร้อย

8.2.2 การเขียนโปรแกรมให้ไมโครซีโอเอสทูว์เริ่มทำงาน

1. เปิดโปรแกรมเอ็กซ์พีเอสเอสดีเคขึ้นมาแล้วทำการสร้างโปรเจคใหม่ให้ชี้มายังเอ็กซ์พีเอสโปรเจ็คที่ได้ทำไว้แล้ว

2. ทำการอินคลูดไฟล์ "includes.h" เข้ามาในโปรแกรมเพื่อให้สามารถใช้งานเอพีไอของไมโครซีโอเอสทูว์ได้

3. เรียกฟังก์ชัน "BSP_IntDisAll" เพื่อทำการปิดอินเทอร์รัพท์ทั้งหมด

4. เรียกฟังก์ชัน "OSInit" เพื่อเตรียมความพร้อมค่าต่างๆ ก่อนที่โอเอสจะเริ่มทำงาน

5. สร้างทาสก์ "AppTaskFirst" ขึ้นมาเพื่อเป็นทาสก์แรกของโอเอส

6. เรียกฟังก์ชัน "OSStart" เพื่อย้ายการทำงานไปยังโอเอส

```
#include "includes.h"
#include "xparameters.h"
#include <stdio.h>

static  OS_STK   AppTaskFirstStk [APP_TASK_FIRST_STK_SIZE];
static  void  AppTaskFirst (void *p_arg);
// Main Program (OS Startup)
int main()
{
    INT8U    err;
    // Make sure interrupts are disabled on interrupt controller
    BSP_IntDisAll();
    // Initialize uC/OS-II
    OSInit();
    // Create Startup Task
    OSTaskCreateExt(AppTaskFirst,
                    (void *)0,
                    &AppTaskFirstStk[APP_TASK_FIRST_STK_SIZE - 1],
                    APP_TASK_FIRST_PRIO,
                    APP_TASK_FIRST_ID,
                    &AppTaskFirstStk[0],
                    APP_TASK_FIRST_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

    OSTaskNameSet(APP_TASK_FIRST_PRIO, "App Task First", &err);
    // Start multitasking
    OSStart();
    return 0;
}
```

รูปที่ 8.5 โปรแกรมส่วน "main" ในการเริ่มการทำงานของโอเอส

7. ใน “AppTaskFirst” ทำการตั้งค่าเริ่มต้นของสแตทิสติกทาส์กโดยการเรียกฟังก์ชัน “OSStatInit” จากนั้นโอเอสได้ถือว่าถูกเริ่มต้นอย่างสมบูรณ์แล้ว ทำการเรียกฟังก์ชัน “myMain” เพื่อย้ายการทำงานไปยังโปรแกรมประยุกต์ที่เราจะเขียนขึ้น

```
static  void  AppTaskFirst (void *p_arg)
{
    INT8U err;
    (void)p_arg; // Prevent compiler warning by doing something with argument
    BSP_InitIO(); // Initialize the I/Os
#if OS_TASK_STAT_EN > 0
    OSStatInit(); // Initialize uC/OS-II's statistics
#endif
    //end of the startup process
    myMain();

}

void myMain(void)
{

}
```

รูปที่ 8.6การทำงานภายในทาส์กแรกของเรา

8. ทำการเขียนโปรแกรมให้พิมพ์ค่าตัวเลขออกมาในทุกๆวินาทีเพื่อทดสอบการทำงานของโอเอสของเราดังนี้

```
void myMain(void)
{
    unsigned int time = 0;
    while(1)
    {
        xil_printf("Time = %d\n",time);
        time++;
        OSTimeDlyHMSM(0,0,1,0);
    }
}
```

รูปที่ 8.7โปรแกรมพิมพ์ตัวเลขทุกวินาที

9. ตั้งค่าการรันโปรแกรมให้มารันที่โปรเจคใหม่ของเรา แล้วนำโปรแกรมไปทดสอบการทำงานดู แล้วลองดูที่คอนโซลจะได้ผลดังภาพ

รูปที่ 8.8ผลการทำงานของโปรแกรมนับเลขในทุกวินาที

10. ทำการเขียนฟังก์ชันทาส์กเพิ่ม โดยให้ทาส์กนั้นพิมพ์คำว่า “Another Task” ออกมาในทุกๆ สอง วินาทีดังนี้

```
static  void  AnotherTask (void *p_arg)
{
    (void)p_arg;
    while(1)
    {
        xil_printf("Another Task\n");
        OSTimeDlyHMSM(0,0,2,0);
    }
}
```

รูปที่ 8.9ทาส์กที่สองในโปรแกรม

11. ทำการสร้างทาส์กที่ได้เขียนไว้โดยเพิ่มคำสั่ง “OSTaskCreate” ลงใน “myMain”

```
void myMain(void)
{
    unsigned int time = 0;
    OSTaskCreate(AnotherTask,(void*)0,
    &AnotherTaskStk[APP_TASK_FIRST_STK_SIZE - 1],1);
    while(1)
    {
        xil_printf("Time = %d\n",time);
        time++;
        OSTimeDlyHMSM(0,0,1,0);
    }
}
```

รูปที่ 8.10การสร้างทาส์กที่สอง

12. นำโปรแกรมไปทดสอบการทำงานแล้วดูผลลัพธ์ที่คอนโซลจะได้ดังรูป

รูปที่ 8.11ผลลัพธ์ของโปรแกรมที่มีสองทาส์ก

บทที่9

ตัวอย่างโปรแกรมประยุกต์

9.1 ภาพรวม

เนื้อหาในบทนี้จะเป็นการออกแบบและการทดลองโปรแกรมประยุกต์ที่สร้างบนระบบไมโครเบลส โดยใช้เรียลไทม์โอเอส ไมโครซีโอเอสทูว์ โดยระบบที่จะทำนั้นจะทำการอ่านค่าจากตัวตรวจจับอุณหภูมิทั้งสิ้นสามตัวในทุกๆวินาทีแล้ววาดออกมาบนจอกราฟิคแอลซีดีในลักษณะของกราฟ และสามารถติดต่อกับพีซีผ่านทางพอร์ทอนุกรมในการอ่านข้อมูลอุณหภูมิ 100 วินาทีล่าสุดของเซ็นเซอร์แต่ละตัวออกมาได้

9.2 อุปกรณ์ภายนอกที่ใช้ในการทดลองนี้

9.2.1 เซ็นเซอร์วัดอุณหภูมิ

สำหรับในการทดลองนี้เราใช้เซ็นเซอร์วัดอุณหภูมิเบอร์ดีเอส1820 (DS1820) ของบริษัทดัลลาสเซมิคอนดัคเตอร์ ซึ่งมีอินเตอร์เฟสการเชื่อมต่อแบบวันไวร์(ONEWIRE) หรือใช้สายเส้นเดียวในการสื่อสาร

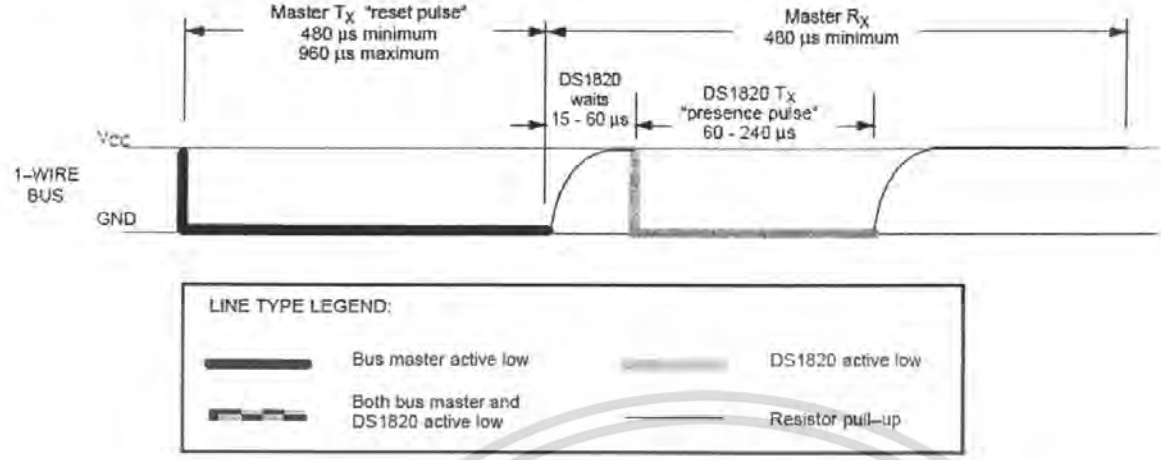
9.2.1.1 โปรโตคอลวันไวร์

9.2.1.1.1 รีเซ็ต

ในการสื่อสารบนโปรโตคอลวันไวร์นั้นจะเริ่มต้นด้วยการส่งสัญญาณรีเซ็ตจากมาสเตอร์ซึ่งลักษณะของสัญญาณรีเซ็ตนั้นเป็นดังนี้

มาสเตอร์ดึงขาสัญญาณให้เป็น 0 ประมาณ 480 – 960 ไมโครวินาที จากนั้นปล่อยให้เป็น 1 ประมาณ 15 – 60 ไมโครวินาที แล้วสลาฟจะตอบรับด้วยการดึงขาสัญญาณลงเป็น 0 ประมาณ 60 – 240 ไมโครวินาทีดังภาพ

INITIALIZATION PROCEDURE "RESET AND PRESENCE PULSES" Figure 11



รูปที่ 9.1สัญญาณรีเซตของโปรโตคอลวันไวร์

9.2.1.1.1 การเขียนและอ่านข้อมูล

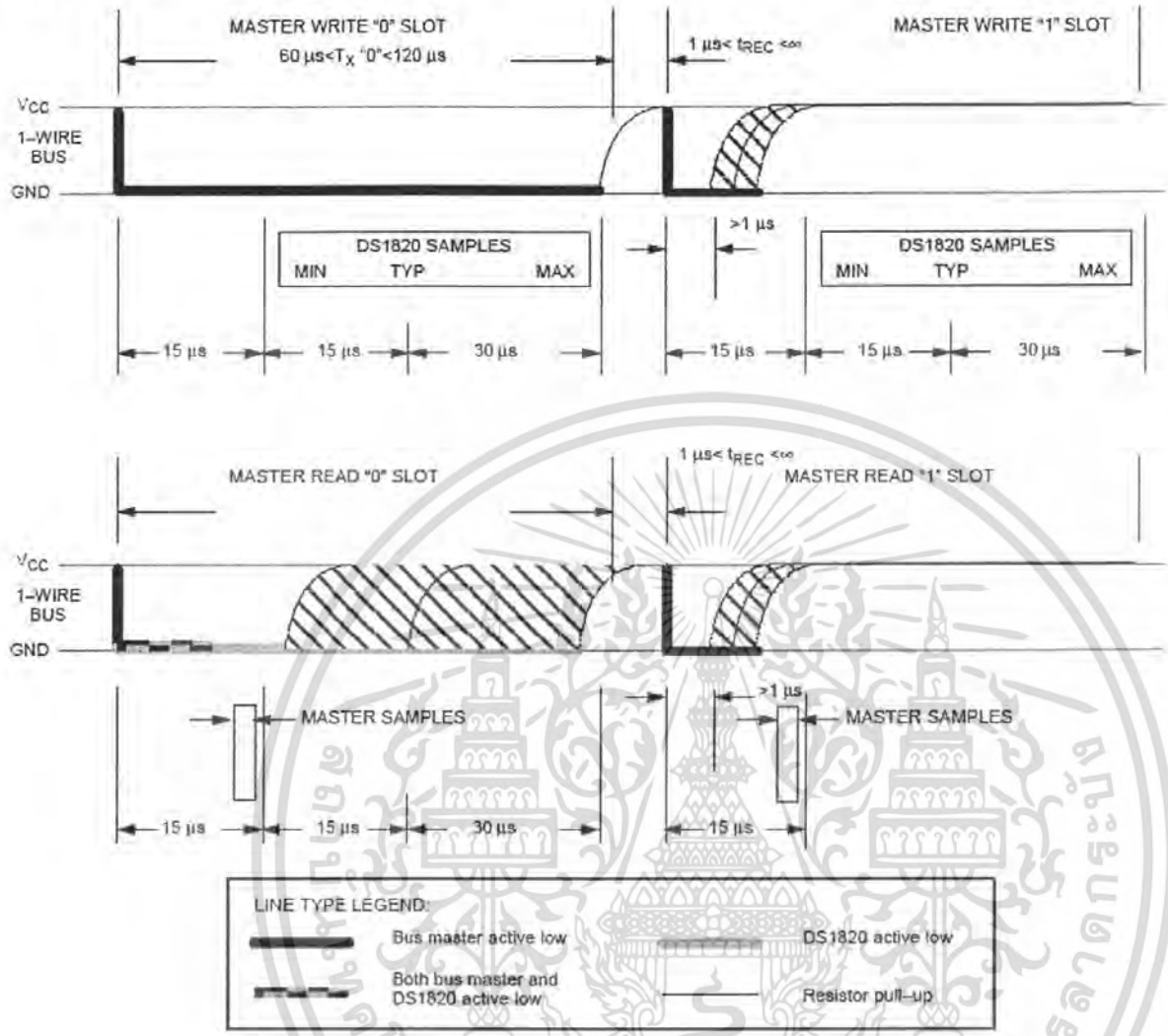
9.2.1.1.1.1 การเขียนข้อมูล

สำหรับการเขียนข้อมูลของมาสเตอร์บนโปรโตคอลวันไวร์นั้น จะเริ่มต้นด้วยการที่มาสเตอร์ดึงสายสัญญาณลงเป็น 0 อย่างน้อย 1 ไมโครวินาที จากนั้นภายใน 15 ไมโครวินาทีสลาฟจะทำการอ่านค่าภายใน 15 – 60 ไมโครวินาทีหลังจากนั้น ซึ่งมาเตอร์จะต้องเปลี่ยนขาสัญญาณให้เป็นไปตามข้อมูลที่ต้องการจะส่ง (0 ถ้าจะส่ง 0 และ 1 ถ้าต้องการจะส่ง 1)

9.2.1.1.1.2 การอ่านข้อมูล

สำหรับการอ่านข้อมูลของมาสเตอร์บนโปรโตคอลวันไวร์นั้นเริ่มต้นด้วยการที่มาสเตอร์ดึงสายสัญญาณลงเป็น 0 อย่างน้อย 1 ไมโครวินาที จากนั้นอีก 15 ไมโครวินาทีมาสเตอร์จะทำการอ่านข้อมูลจากสายสัญญาณ ซึ่งสลาฟจะเปลี่ยนข้อมูลในสายสัญญาณในช่วงนี้ และทำการรอจนครบ 60 ไมโครวินาทีแล้วจึงเริ่มการสื่อสารในไทม์สลอตถัดไป

READ/WRITE TIMING DIAGRAM Figure 12



รูปที่ 9.2การอ่านและเขียนข้อมูลบนโปรโตคอลวันไวร์

9.2.1.2 การอ่านอุณหภูมิจากดีเอส1820

1. เริ่มต้นจากการที่มาสเตอร์ส่งสัญญาณรีเซตไปบนบัสวันไวร์เพื่อให้อุปกรณ์พร้อมที่จะทำงาน

2. มาสเตอร์ทำการส่งข้อมูล 0xCC ซึ่งเป็นคำสั่งข้ามรอมไปให้กับตัวเซ็นเซอร์เนื่องจากบนบัสมีเซ็นเซอร์อุณหภูมิต่ออยู่เพียงอย่างเดียว

3. มาสเตอร์ทำการส่งข้อมูล 0x44 ซึ่งเป็นคำสั่งให้เซ็นเซอร์ทำการแปลงค่าอุณหภูมิ จากนั้นรออย่างน้อย 200 มิลลิวินาทีเพื่อรอให้เซ็นเซอร์ทำการแปลงอุณหภูมิให้เรียบร้อย

4. มาสเตอร์ทำการส่งสัญญาณรีเซตบนบัสวันไวร์อีกครั้ง จากนั้นทำการส่ง 0xCC และ 0xBE เพื่อบอกให้เซ็นเซอร์ส่งข้อมูลอุณหภูมิกลับมา

5. มาสเตอร์ทำการอ่านข้อมูลจากบัสซึ่งข้อมูลนี้จะมีขนาดแปดบิตและตัวเลขที่อ่านได้จะเป็น 2 เท่าของอุณหภูมิ (บิตสุดท้ายจะมีค่าเท่ากับ 0.5 องศา)

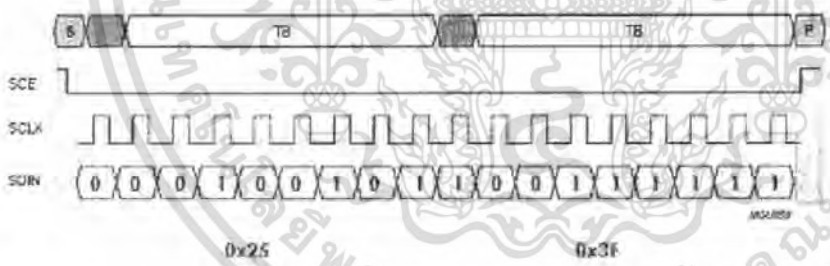
9.2.2 จอแอลซีดีแบบกราฟิค

จอแอลซีดีแบบกราฟิคที่เรานำมาใช้ในการทดลองนี้เป็นจอของโทรศัพท์มือถือโนเกียรุ่น 6100 ซึ่งใช้ชิพควบคุมของฟิลิปส์และมีการเชื่อมต่อโดยโปรโตคอลเอสพีไอ

9.2.2.1 โปรโตคอลเอสพีไอ

ในการติดต่อกับจอโนเกีย 6100 นั้นเราจะใช้โปรโตคอลเอสพีไอในการสื่อสาร โดยลักษณะของโปรโตคอลเอสพีไอนั้นเป็นดังนี้

เมื่อต้องการเริ่มการสื่อสารมาสเตอร์จะทำการดึงขาสัญญาณชิพเอ็นเอเบิล(SCE) ลงเป็น 0 เพื่อให้อุปกรณ์รับฟังหลังจากนั้นมาสเตอร์จะทำการส่งข้อมูลบิตแรกซึ่งเป็นการบอกว่าสิ่งที่กำลังจะส่งนี้เป็นข้อมูลหรือคำสั่ง (0 คือคำสั่ง 1 คือข้อมูล) โดยมาสเตอร์จะทำการเปลี่ยนแปลงขาเอสดีเอ (SDA) ให้เป็นข้อมูลที่ต้องการจะส่ง (1 หรือ 0) จากนั้นทำการส่งสัญญาณคลอดออกไปทางขาเอสซีแอล (SCL) หนึ่งลูก สลาฟจะอ่านข้อมูลที่มาสเตอร์ส่งให้นี้หลังจากได้รับสัญญาณทางเอสซีแอลแล้วเท่านั้น



รูปที่ 9.3การสื่อสารบนโปรโตคอลเอสพีไอ

9.2.2.2 การสั่งงานจอโนเกีย 6100

9.2.2.2.1 การสั่งให้จอเริ่มทำงาน

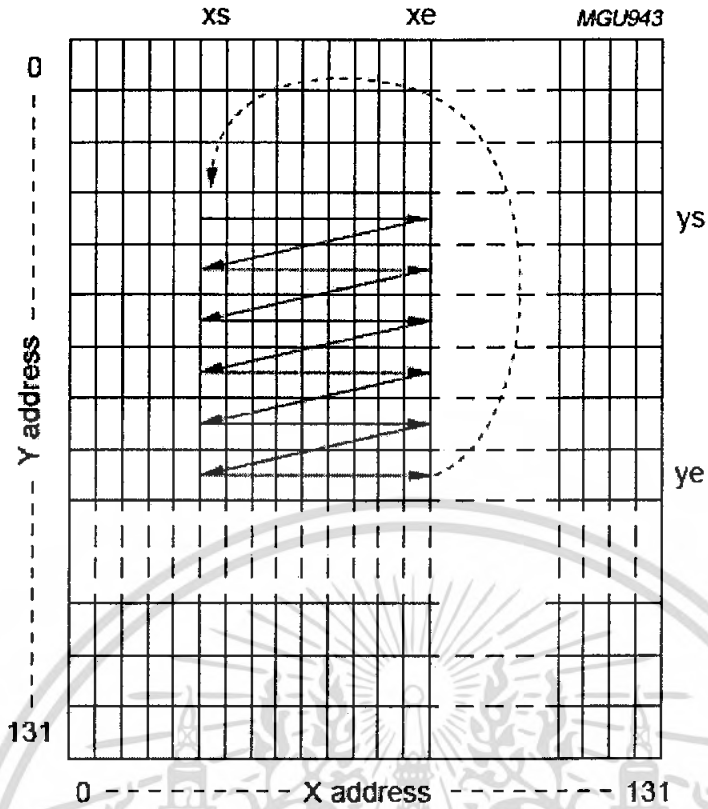
ในการสั่งให้จอโนเกีย 6100 เริ่มต้นทำงานนั้นมีขั้นตอนดังต่อไปนี้

1. ทำการส่งสัญญาณรีเซตไปยังจอ
2. เริ่มส่งคำสั่ง “SLEEPOUT” (0x11) ออกไปทางโปรโตคอลเอสพีไอ

3. ส่งคำสั่ง “NORON” (0x13) ผ่านโปรโตคอลเอสพีไอไปยังจอเพื่อบอกให้จอทำงานในโหมดปกติ
4. ส่งคำสั่ง “COLMOD” (0x3A) ไปยังจอเพื่อบอกว่าจะทำการตั้งค่าโหมดสี
5. ส่งข้อมูล 0x02 ไปยังจอเพื่อบอกให้จอทำงานที่โหมด 2 (สี 8 บิต)
6. ส่งคำสั่ง “MADCTL” (0x36) และข้อมูล 0x00 เพื่อตั้งค่าการเข้าถึงหน่วยความจำของจอ
7. ส่งคำสั่ง “SETCON” (0x25) และตามด้วยข้อมูลความสว่างของจอที่ต้องการ (ไม่เกิน) 0x3F ไปยังจอเพื่อปรับความสว่าง
8. ปิดการสื่อสารเอสพีไอ
9. เริ่มการสื่อสารเอสพีไอใหม่อีกครั้งและส่งคำสั่ง “DISPON” (0x29) ไปยังจอเพื่อเปิดจอภาพ
10. ปิดการสื่อสารเอสพีไอ และจอภาพพร้อมที่จะเริ่มทำงาน

9.2.2.2.2 การส่งภาพออกไปแสดงยังจอ

1. เริ่มต้นการสื่อสารเอสพีไอ จากนั้นทำการส่งคำสั่ง “CASET” (0x2A) ไปยังจอ เพื่อทำการตั้งค่าตำแหน่งคอลลัมที่ต้องการจะเริ่มเขียนภาพ
2. ส่งข้อมูล คอลลัมน์เริ่มต้น(x1) และคอลลัมน์สิ้นสุด(x2) เพื่อกำหนดขอบเขตในแนวแกนเอ็กซ์ที่จะทำการเขียนภาพ
3. ทำการส่งคำสั่ง “PASET” (0x2B) ไปยังจอภาพเพื่อตั้งค่าตำแหน่งแถวที่ต้องการจะเริ่มเขียนภาพ
4. ส่งข้อมูลแถวเริ่มต้น(y1) และตำแหน่งแถวสิ้นสุด(y2) เพื่อกำหนดขอบเขตในแนวแกนวายที่จะทำการเขียนภาพ
5. หลังจากที่ทำการส่ง “CASET” และ “PASET” เรียบร้อยแล้วข้อมูลที่เราส่งไปจะไปแสดงที่ตำแหน่งในขอบเขตที่ตั้งค่าไว้ทีละจุดดังภาพ



รูปที่ 9.4การส่งภาพไปแสดงผลที่จอ

6. ทำการส่งคำสั่ง "RAMWR" (0x2C) เพื่อเริ่มการเขียนภาพในขอบเขตที่กำหนดไว้

7. ทำการส่งข้อมูลจุดภาพโดยข้อมูล 1 ไบต์จะหมายถึงค่าสีแปดบิตของจุด 1 จุด

9.3 ทาส์กในระบบ

ในระบบนี้ประกอบไปด้วยทาส์กทั้งสิ้น 6 ทาส์กด้วยกันโดยแต่ละทาส์กมีหน้าที่ของตนเองดังนี้

9.3.1 AppTaskFirst

ทำหน้าที่ในการตั้งค่าเริ่มต้นของระบบ ทำการสร้างทาส์กอื่นๆขึ้นมา และทำการแสดงผลภาระของซีพียูในหน่วยเปอร์เซ็นต์ออกทางเซเว่นเซกเมนท์

9.3.2 TempLoggerTask1(2,3)

เป็นทาส์กสามทาส์กที่ทำหน้าที่ในการอ่านค่าจากเซ็นเซอร์วัดอุณหภูมิแต่ละตัว(ทาส์กละหนึ่งตัว) ในทุกๆ วินาที และดูแลการจัดเก็บอุณหภูมิใน 100 วินาทีล่าสุดในหน่วยความจำ

9.3.3 DispTask

เป็นทาส์กที่คอยควบคุมการแสดงผลของจอกราฟิคแอลซีดีโนเกีย 6100 คอยทำการคำนวนตำแหน่งในการวาดกราฟบนจอ และควบคุมการลบและวาดภาพบนจอทั้งหมด

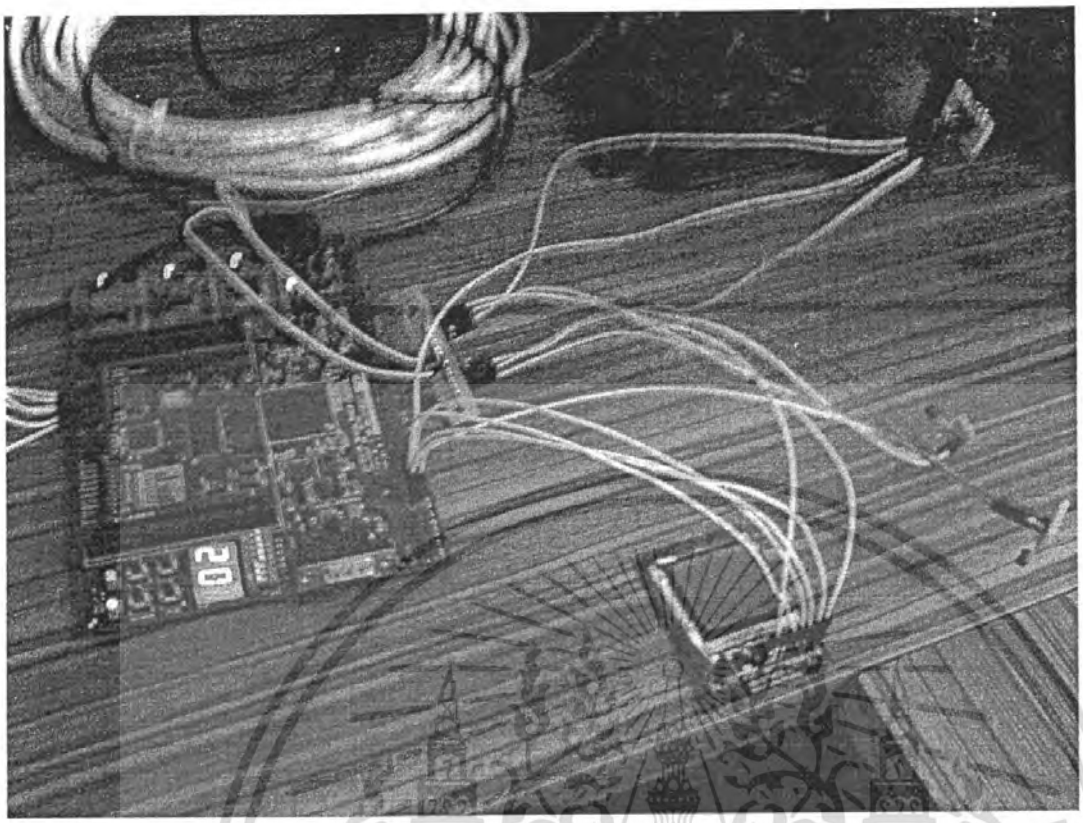
9.3.4 UARTCmdTask

ทำหน้าที่ในการจัดการรับข้อมูลจากพอร์ทอนุกรมและทำการส่งเอคโคกลับไปยังคอนโซลฝั่งพีซีและประมวลผลคำสั่งที่ได้มาทางพอร์ทอนุกรม

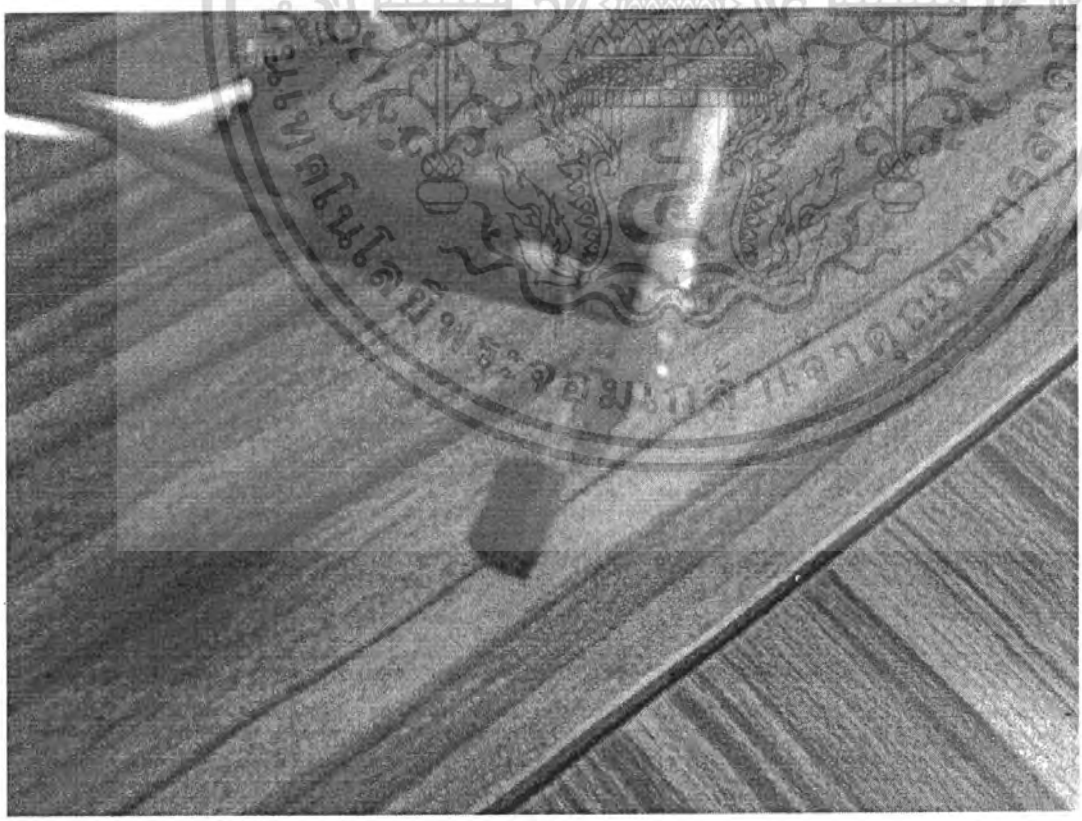
9.4 ลำดับการทำงานของโปรแกรม

1. ทำการตั้งค่าเริ่มต้นต่างๆ และสร้างทาส์ก AppTaskFirst ขึ้นมา จากนั้นส่งการทำงานต่อให้กับโอเอส
2. โอเอสจะเรียกทาส์ก AppTaskFirst ขึ้นมาทำงาน AppTaskFirst จะทำการสร้างทาส์ก TempLoggerTask ทั้งสามทาส์กขึ้นมา จากนั้นจะรออีเวนท์แฟลกที่บอกว่าทาส์กทั้งสามนั้นได้ทำการเตรียมการเรียบร้อยแล้ว
3. หลังจากได้รับอีเวนท์แฟลกครบทั้งสามตัวแล้ว AppTaskFirst จะทำการสร้าง DispTask ขึ้นมาเพื่อจัดการการแสดงผลที่จอภาพ จากนั้นทำการสร้าง UARTCmdTask เพื่อรอรับคำสั่งจากผู้ใช้งานระบบ
4. แต่ละทาส์กทำหน้าที่ของตัวเองไปตามที่ได้กล่าวไว้แล้ว

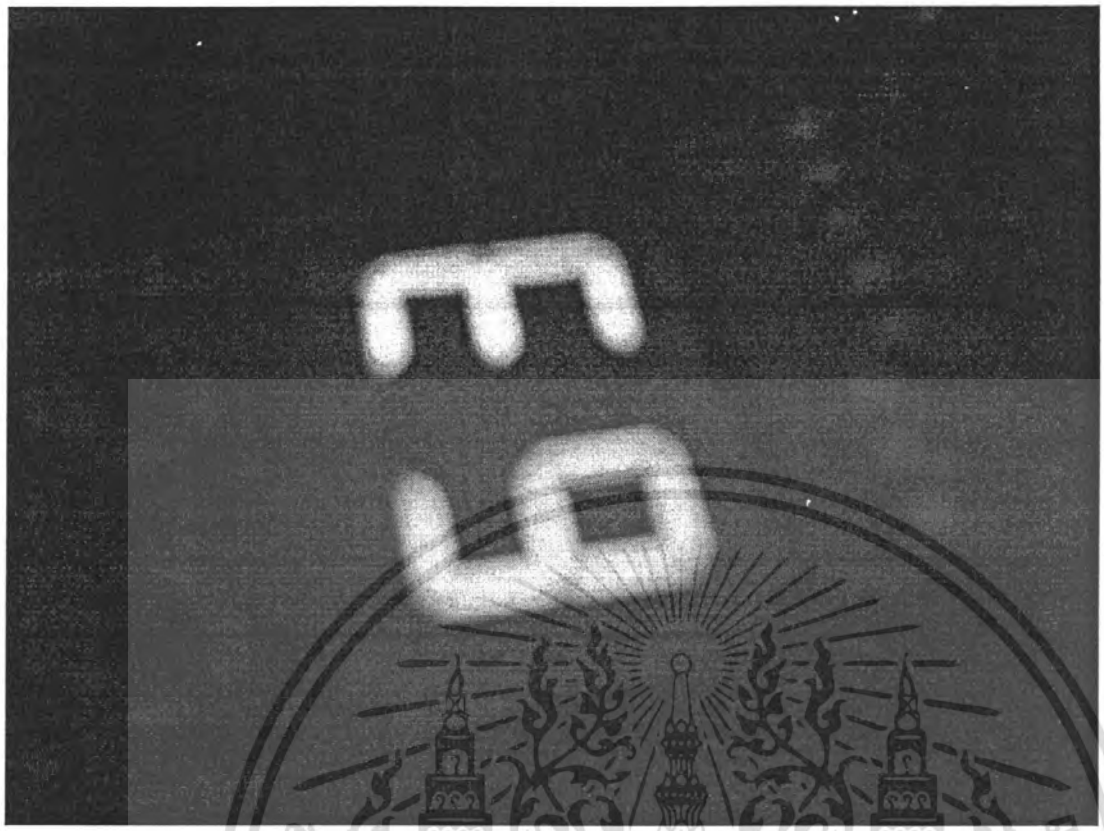
9.5 ภาพรวมการทำงานของระบบ



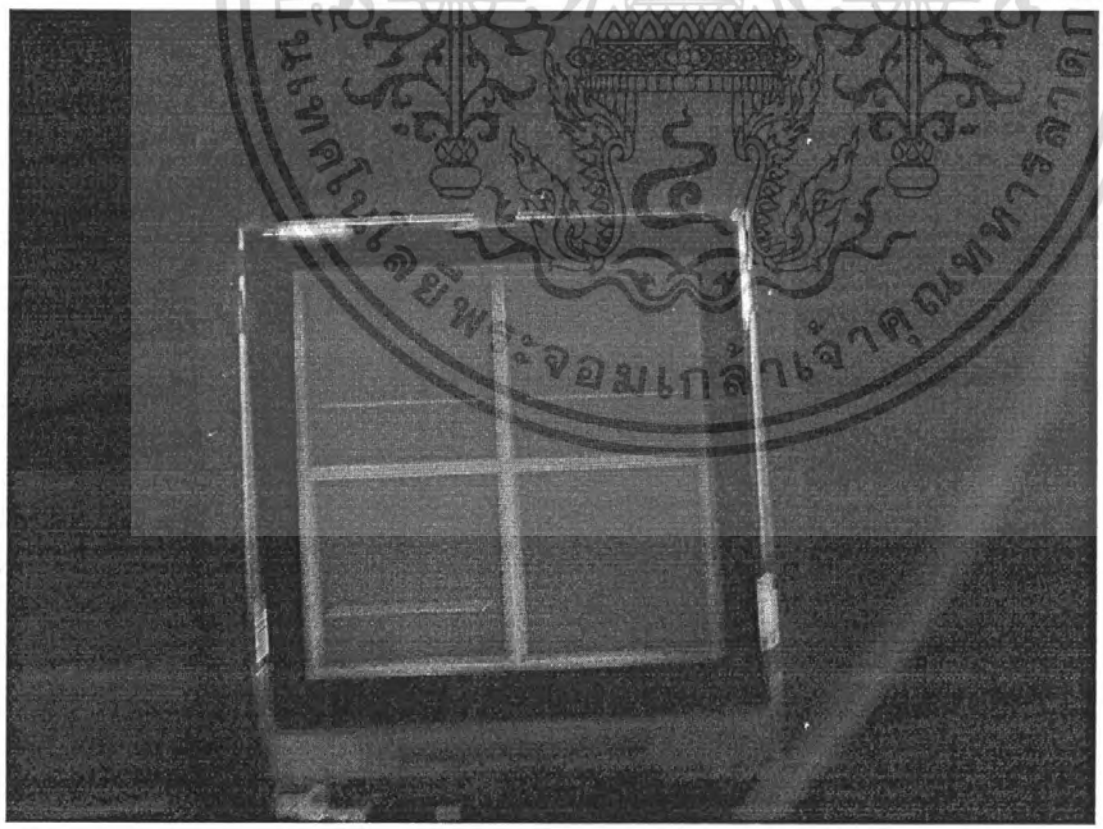
รูปที่ 9.5ภาพรวมของระบบ



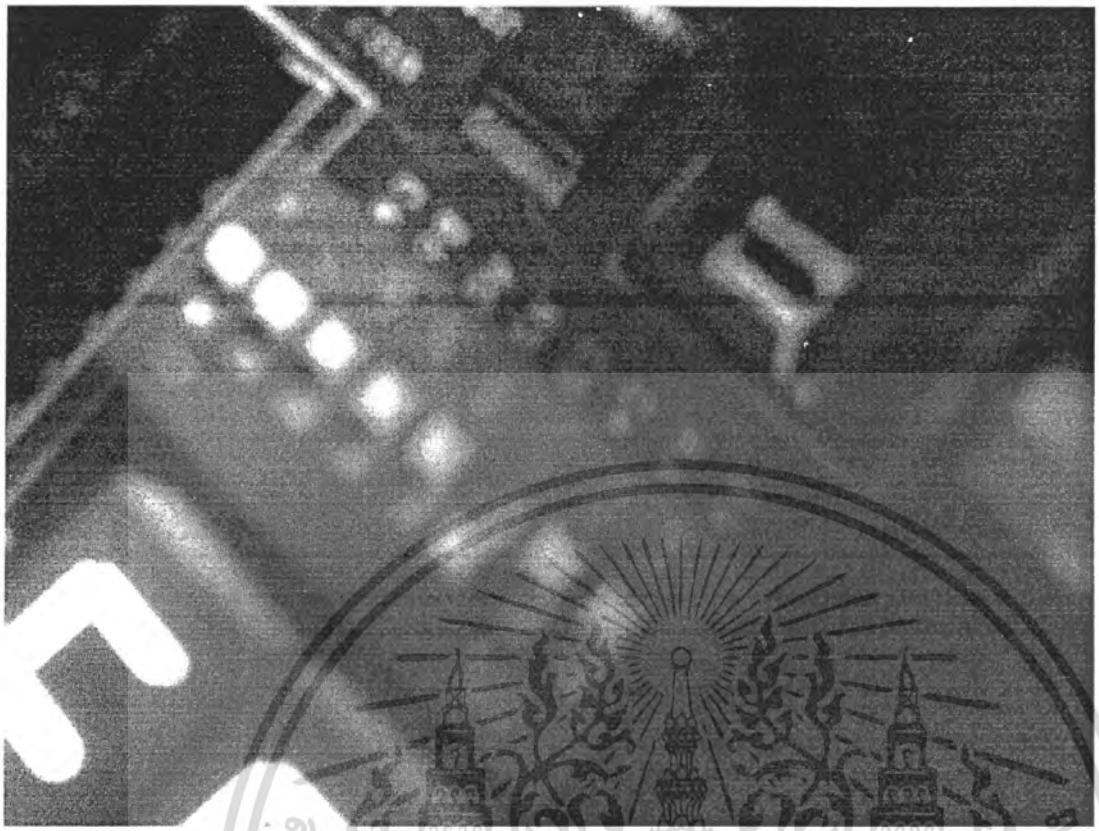
รูปที่ 9.6เซ็นเซอร์วัดอุณหภูมิ



รูปที่ 9.7ตัวเลขแสดงภาระของซีพียู

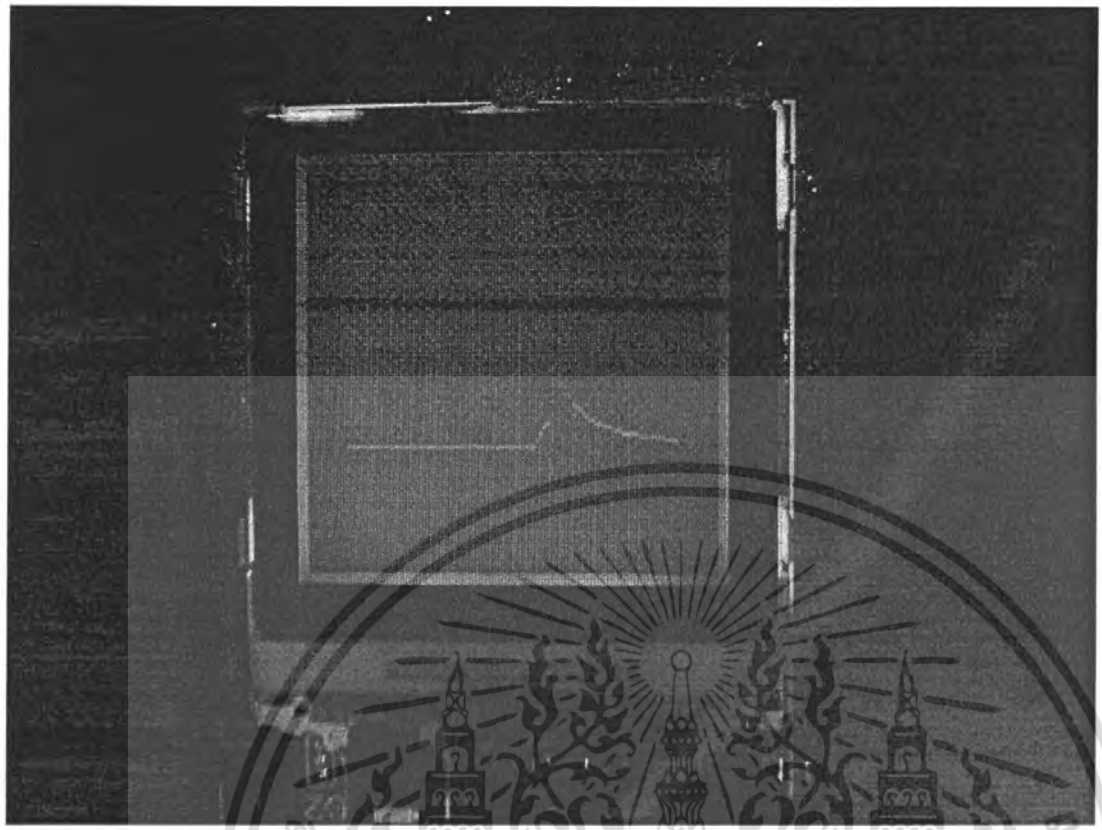


รูปที่ 9.8กราฟแสดงอุณหภูมิของเซ็นเซอร์ทั้งสามตัว



รูปที่ 9.9สวิตช์เลือกให้แสดงผลเซ็นเซอร์ตัวที่สามสามตัวเดียว

รูปที่ 9.10เพิ่มอุณหภูมิที่เซ็นเซอร์ตัวที่สาม



รูปที่ 9.11กราฟอุณหภูมิของเซ็นเซอร์ตัวที่สาม

Tera Term - COM3 VT

File Edit Setup Control Window Help

```
$GET,3
$DATA,3,27,27,27,27,27,27,27,27,27,27,27,27,27,27,28,28,28,28,28,28,28,28,28,28,
28,28,28,28,28,28,29,29,29,29,29,29,29,29,30,30,30,30,30,31,31,31,32,32,32,33,33
,34,34,35,36,37,38,39,41,42,44,45,48,50,49,33,32,30,28,26,26,26,26,26,26,26,26,2
6,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26
```

รูปที่ 9.12ผลการอ่านค่าเซ็นเซอร์ตัวที่สามทางพอร์ทอนุกรม

บรรณานุกรม

Jean J. Labrosse, 2002, **MicroC/OS-II The Real-Time Kernel Second Edition**,

CMPBooks,San Francisco, New York, Lawrence

"MicroBlaze Processor Reference Guide" [Online]

Available: http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf

"On-Chip Peripheral Bus Architecture Specifications" [Online]

Available: http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/

9A7AFA74DAD200D087256AB30005F0C8/$file/OpbBus.pdf

"Designing Custom OPB Slave Peripherals for MicroBlaze" [Online]

Available: http://www.xilinx.com/ipcenter/processor_central/microblaze/doc/opb_tutorial.pdf

"ภาคผนวก Sapatan3 (ปรับปรุงล่าสุด 14 ส.ค. 49)" [Online]

Available: http://www.kmitl.ac.th/~ksjirasa/Laboratory/DigitalLab/_Sapatan3BM.pdf