USB TO RS232 CONVERTER

โดย

นาย ธนิษฐ์ ดงคาสวรรค์ รหัสประจำตัว 45010333

นาย เก้าวงค์ วิศิษฏ์สิน รหัสประจำตัว 45010068

อาจารย์ที่ปรึกษา

ดร. กสิน วิเชียรชม

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

สาขาอิเล็กทรอนิกส์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2548

ปริญญานิพนธ์ปีการศึกษา 2548

ภาควิชา อิเล็คทรอนิกส์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง USB to RS-232 Convertor

ผู้จัดทำ

1. นาย ธนิษฐ์ คงคาสวรรค์ 45010333
2. นาย เก้าวงค์ วิศิษฏ์สิน 45010068

.....อาจารย์ที่ปรึกษา

(ดร. กสิน วิเชียรชม)

USB TO RS232 CONVERTER

นาย เก้าวงค์ วิศิษฏ์สิน
นาย ธนิษฐ์ คงคาสวรรค์
ดร.กสิน วิเชียรชม อาจารย์ที่ปรึกษา
ปีการศึกษา 2548

บทคัดย่อ

รายงานนี้อธิบายถึงการศึกษาและออกแบบวงจรรวมของระบบดิจิตอล ซึ่งประกอบด้วยฮาร์ดแวร์ที่ทำหน้าที่เปลี่ยนรูปแบบข้อมูลอนุกรมที่เป็น USB ให้ไปเป็นรูปแบบข้อมูลอนุกรมที่เป็นแบบ RS-232 ซึ่งในการออกแบบวงจรใช้วิธีเขียนด้วยภาษา VHDL แล้วสังเคราะห์วงจรและทดสอบการทำงานด้วยโปรแกรม ModelSim ผลการจำลองการทำงานวงจรที่ออกแบบสามารถรองรับการส่งถ่ายข้อมูลจาก USBในแบบบัลก์ (Bulk Mode) ไปสู่ ข้อมูลในแบบ RS-232 ที่อัตราการส่ง 9600 บิทต่อวินาที

USB TO RS232 CONVERTER

Mr. Kaowong Visissin
Mr. Thanit Kongkasawan
Dr. Kasin Vichianchom (Advisor)
Education Year 2005

Abstract

This report is described the study and design of a digital system module called USB-to-RS232 Converter. The module is able to convert data stream from a bulk transfer mode of the USB protocol to the RS-232 protocol. The design was written in VHDL. It was synthesis and functionally verified with ModelSim®. The simulation results show that it is capable to transfer data from USB format to RS-232 at data rate 9600 bps.

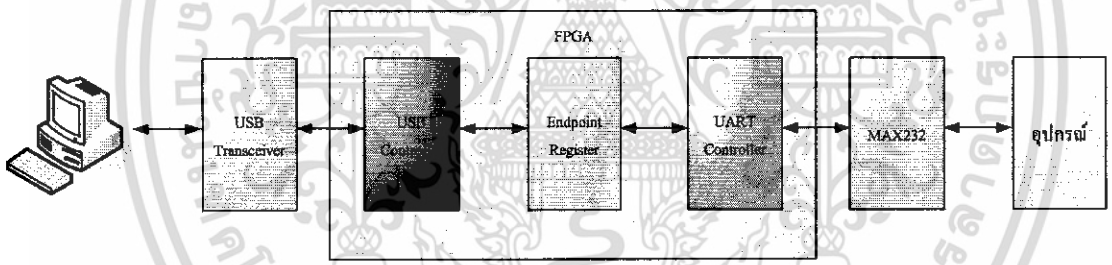
สารบัญ

บทที่ 1
บทนำ

ในปัจจุบันระบบบัส USB (Universal Serial Bus) ได้เข้ามามีบทบาทในการใช้งานอุปกรณ์รอบข้างของคอมพิวเตอร์มากขึ้นทุกขณะ ทั้งนี้เนื่องจากระบบบัสชนิดนี้มีความสะดวกในการใช้งานและมีความยืดหยุ่นในด้านการเชื่อมต่อสูง นอกจากนี้การเชื่อมต่ออุปกรณ์ USB ยังสามารถทำได้ขณะที่เครื่องคอมพิวเตอร์กำลังทำงานอยู่ ซึ่งการใช้งานลักษณะนี้คือรูปแบบปลั๊กแอนด์เพลย์ (Plug and Play) อย่างแท้จริง จุดเด่นอีกอย่างหนึ่งคือมีความเร็วในการเชื่อมต่อสูง (1.5 เมกะบิตต่อวินาที สำหรับอุปกรณ์โลว์สปีด, 12 เมกะบิตต่อวินาที สำหรับอุปกรณ์ฟูลสปีด และ 480 เมกะบิตต่อวินาที สำหรับอุปกรณ์ไฮสปีด)

แต่ในปัจจุบันยังคงมีการใช้การส่งข้อมูลในระบบบัสแบบอนุกรม ด้วยเหตุนี้เราจึงมีความคิดที่จะทำการออกแบบ IC ที่ทำหน้าที่ในการเชื่อมต่อหรือพูดคุยกันระหว่างระบบบัส USB และระบบบัสแบบอนุกรมขึ้น เพื่อให้อุปกรณ์ที่ใช้ระบบบัสแบบอนุกรมสามารถแรกเปลี่ยนข้อมูลกับระบบบัสที่เป็น USB ได้

สำหรับการออกแบบเราแบ่งออกเป็นส่วนต่างๆ ดังนี้



รูปที่ 1.1 Block Diagram ของวงจรที่ใช้งาน

1. USB Transceiver เป็นเสมือนตัวกลางในการเชื่อมต่อระหว่างพอร์ต USB กับ FPGA ที่ออกแบบ

2. USB Controller ทำหน้าที่ในการแปลงโปรโตคอล USB แล้วทำการแยกข้อมูลไปเก็บไว้ใน Endpoint Register

3. Endpoint Register ทำหน้าที่ในการเก็บข้อมูลที่ได้รับจาก USB Controller และ UART Controller เพื่อรอในการเอาไปเรียกใช้ต่อ

4. UART Controller ทำหน้าที่ควบคุมการรับส่งข้อมูลที่เก็บอยู่ในเอนด์พอยน์กับอุปกรณ์ และทำการเข้ารหัสข้อมูลให้เป็นโปรโตคอลแบบ RS-232

5. MAX232 ทำหน้าที่เปลี่ยนระดับแรงดันที่มาจาก FPGA ให้เป็นระดับมาตรฐาน EIA เพื่อส่งไปยังอุปกรณ์ RS-232 และเปลี่ยนระดับแรงดันจาก RS-232 ให้เป็นระดับลอจิกเพื่อส่งไปยัง FPGA

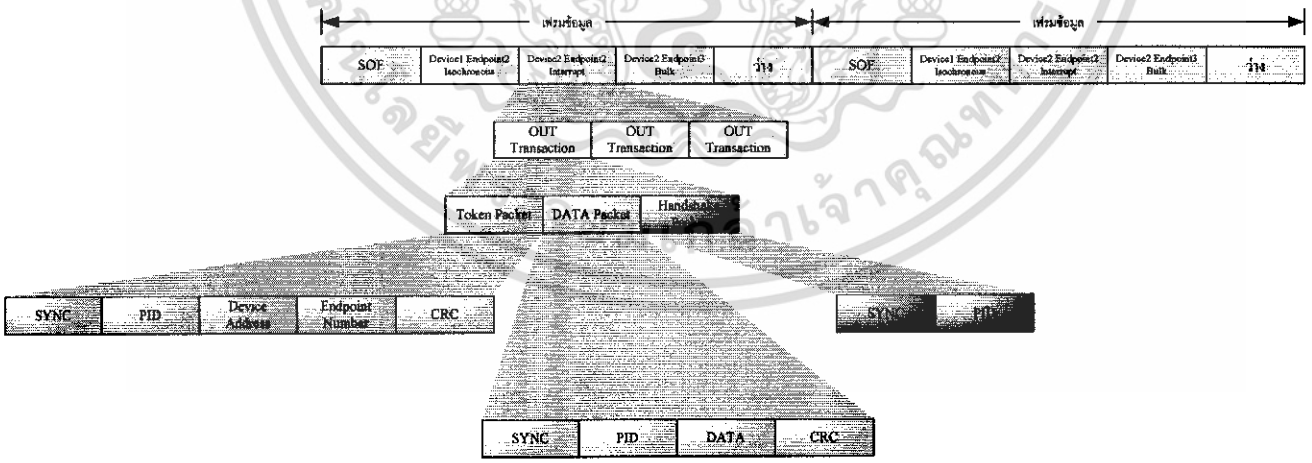
บทที่ 2
ทฤษฎีพื้นฐาน

การส่งถ่ายข้อมูลบนระบบบัส USB พื้นฐาน

กลไกการส่งข้อมูลบนระบบบัส USB ค่อนข้างยุ่งยากพอสมควร โดยทั่วไปแล้วอาจไม่ต้องเรียนรู้รายละเอียดของมันอย่างลึกซึ้งมากนักเนื่องจากส่วนใหญ่งานที่เกี่ยวข้องกับการนส่งถ่ายข้อมูลนั้นคอนโทรลเลอร์ชิปจะเป็นตัวที่จัดการให้ แต่การทำความเข้าใจในกระบวนการส่งถ่ายข้อมูลบ้างก็จะทำให้สามารถตัดสินใจได้ว่าจะใช้การส่งถ่ายข้อมูลชนิดไหน อีกทั้งยังช่วยในการเขียนเฟิร์มแวร์สำหรับคอนโทรลเลอร์ชิปและค้นหาบักที่เกิดขึ้นเมื่อเราได้สร้างวงจรและเขียนโค้ดขึ้นมา

2.1 การจัดการข้อมูลบนระบบบัส USB

การสื่อสารบนระบบบัส USB เป็นการสื่อสารที่มีโฮสเป็นศูนย์กลาง ซึ่งโฮสมีหน้าที่ทำให้การส่งถ่ายข้อมูลที่อยู่บนบัสเกิดขึ้นเร็วที่สุดเท่าที่จะเป็นได้โดยการแบ่งเวลาออกเป็นช่วงเล็กๆ เรียกว่าเฟรม (Frame) และสำหรับโหมดไฮสปีด (Hi-speed) จะแบ่งเป็นไมโครเฟรม (Micro frame) โดยโฮสจะแบ่งช่วงเวลาสำหรับโลว์สปีด (Low-speed) และฟูลสปีด (Full-speed) แต่ละเฟรมจะมีค่าเท่ากับ 1 มิลลิวินาที สำหรับไฮสปีดจะแบ่งเฟรมออกเป็น 8 ส่วน โดยให้ชื่อแต่ละส่วนว่าไมโครเฟรมและมีช่วงเวลาเท่ากับ 125 ไมโครวินาที แต่ละเฟรมหรือไมโครเฟรมจะเริ่มต้นด้วยตัวเริ่มเฟรมหรือเรียกสั้นๆ ว่า SOF (Start-Of-Frame)



รูปที่ 2.1 การจัดข้อมูลภายในเฟรมของ USB

การส่งถ่ายข้อมูลแต่ละชนิดประกอบด้วยการสื่อสารที่เรียกว่า ทรานแซคชัน (Transaction) มากกกว่า 1 ชุด สำหรับการส่งถ่ายแบบคอนโทรลจะมีทรานแซคชันมากกว่า 1 ชุด

เอนด์พอยน์ IN เราก็จะมีเอน์พอยน์ OUT เพื่อสำหรับ

เสมอเนื่องจากมันจะมีขั้นตอนหรือเสตจมากกว่า 1 เสตจ ซึ่งแต่ละเสตจอาจประกอบด้วยทรานแซคชันมากกว่า 1 ชุด ส่วนการส่งถ่ายข้อมูลแบบอื่นๆ จะใช้ทรานแซคชันหลายชุดก็ต่อเมื่อมันมีข้อมูลมากเกินกว่าที่จะใส่ลงไปในทรานแซคชันเดียวได้ ทรานแซคชันของการส่งถ่ายข้อมูลอาจจะทำเสร็จภายใน 1 เฟรมหรือไมโครเฟรม หรืออาจต้องแบบข้อมูลเป็นส่วนๆ แล้วส่งไปกับเฟรมหรือไมโครเฟรมหลายๆ ชุด ขึ้นอยู่กับการจัดเวลาของโฮสให้ทรานแซคชันและความเร็วการตอบสนองของอุปกรณ์

เนื่องจากการส่งข้อมูลทุกครั้งจะต้องใช้เส้นทางร่วมกัน ดังนั้นทรานแซคชันแต่ละชุดต้องมีแอดเดรสอุปกรณ์ใส่ลงไปด้วย อุปกรณ์ทุกตัวจะมีแอดเดรสของมันเองที่ไม่ซ้ำกับตัวอื่นซึ่งกำหนดมาจากโฮส ทรานแซคชันแต่ละชุดจะเริ่มต้นขึ้นเมื่อโฮสส่งบล็อกข้อมูลซึ่งมีแอดเดรสอุปกรณ์และตำแหน่งเฉพาะหรือเอนด์พอยน์ภายในอุปกรณ์ที่ใช้รับข้อมูลรวมอยู่ด้วย ทุกสิ่งทุกอย่างที่ออกมาจากอุปกรณ์จะเป็นการตอบสนองต่อการร้องขอที่รับมาจากโฮสเพื่อเป็นการส่งข้อมูลหรือสถานะของอุปกรณ์กลับออกไป

2.2 องค์ประกอบการส่งถ่ายข้อมูล

การส่งถ่ายข้อมูลแต่ละชนิดบนระบบบัส USB ประกอบขึ้นจากทรานแซคชัน (Transaction) ซึ่งแต่ละทรานแซคชันสร้างมาจากแพ็กเก็ต (Packet) โดยที่แต่ละแพ็กเก็ตจะบรรจุข้อมูลเอาไว้ แต่ก่อนจะกล่าวรายละเอียดของทรานแซคชันนั้นเราต้องทำความเข้าใจกับองค์ประกอบพื้นฐานที่สำคัญของการสื่อสารก่อน นั่นคือ เอนด์พอยน์ และ ไปป์

2.2.1 ดีไวท์เอนด์พอยน์ (Device Endpoint)

การสื่อสารทุกครั้งข้อมูลจะต้องเดินทางออกมาจากดีไวท์เอนด์พอยน์หรือถูกส่งเข้าไปในดีไวท์เอนด์พอยน์ ซึ่งเอนด์พอยน์ก็คือบัฟเฟอร์ซึ่งใช้สำหรับเก็บไบท์ข้อมูล ตามปกติแล้วจะเป็นกลุ่มของหน่วยความจำหรือรีจิสเตอร์ซึ่งอยู่ภายในตัวคอนโทรลเลอร์ชิป ข้อมูลที่อยู่ภายในดีไวท์เอนด์พอยน์อาจเป็นข้อมูลที่รับเข้ามาหรืออาจจะเป็นข้อมูลที่รอเพื่อจะส่งออกไป สำหรับด้านโฮสนั้นจะไม่มีเอนด์พอยน์แต่จะมีบัฟเฟอร์สำหรับเก็บข้อมูลที่รับเข้ามาและสำหรับข้อมูลที่พร้อมจะส่งออกไป โดยโฮสจะทำตัวเป็นเหมือนจุดเริ่มต้นของการสื่อสารกับดีไวท์เอนด์พอยน์

ในข้อกำหนด USB ได้ให้คำจำกัดความของเอนด์พอยน์ไว้ว่า “เป็นส่วนหนึ่งของอุปกรณ์ซึ่งสามารถกำหนดตำแหน่งให้ไม่ซ้ำกับอุปกรณ์ตัวอื่นได้ ซึ่งมันเป็นได้ทั้งแหล่งจ่ายและตัวรับข้อมูลของการสื่อสารที่อยู่ระหว่างโฮสกับอุปกรณ์” จากคำจำกัดความนี้ชี้ให้เห็นว่า การสื่อสารข้อมูลเอนด์พอยน์นั้นสามารถทำได้ในทิศทางเดียวเท่านั้น แต่สำหรับคอนโทรลเลอร์เอนด์พอยน์จะเป็นกรณีพิเศษซึ่งมีการสื่อสารในลักษณะสองทิศทาง นั่นหมายความว่า ถ้าเรามีคอนโทรลเลอร์เอนด์พอยน์ IN เราก็จะมีคอนโทรลเลอร์เอนด์พอยน์ OUT โดยอัตโนมัติ

เอนด์พอยน์แต่ละตัวต้องการแอดเดรสที่ไม่ซ้ำกับตัวอื่น ซึ่งแอดเดรสเหล่านี้ประกอบไปด้วยหมายเลขเอนด์พอยน์และทิศทาง หมายเลขเอนด์พอยน์อาจมีค่าอยู่ในช่วง 0 ถึง 15 สำหรับทิศทางจะมองจากโฮสเป็นหลัก (IN คือ มีทิศทางไปยังโฮส และ OUT คือ มีทิศทางออกจากโฮส) สำหรับการส่งข้อมูลแบบคอนโทรลเป็นการส่งแบบสองทิศทาง ดังนั้นมันจึงประกอบไปด้วยคู่ของเอนด์พอยน์ IN และ OUT ซึ่งหมายเลขเอนด์พอยน์ที่ใช้ร่วมกันก็คือเอนด์พอยน์ศูนย์

อุปกรณ์ทุกตัวต้องมีเอนด์พอยน์ศูนย์ซึ่งถูกตั้งมาให้เป็นคอนโทรลเอนด์พอยน์ ตามปกติแล้วเรามักใช้เพียงตัวเดียว แต่ถ้าเราต้องการใช้มากกว่านั้นก็สามารถทำได้

การส่งถ่ายข้อมูลชนิดอื่นๆ จะสื่อสารข้อมูลในทิศทางเดียวเท่านั้น ซึ่งแอดเดรสเอนด์พอยน์หนึ่งๆ สามารถกำหนดให้เป็นได้ทั้ง IN และ OUT แต่มันจะอยู่แยกกัน เช่น EP1 IN (เป็นเอนด์พอยน์ 1 สำหรับส่งข้อมูลไปยังโฮส) และ EP1 OUT (เป็นเอนด์พอยน์ 1 สำหรับส่งข้อมูลออกจากโฮส) ได้ด้วย

อุปกรณ์โลว์สปีด นอกจากเอนด์พอยน์ศูนย์แล้ว อุปกรณ์จะมีเพียง 2 เอนด์พอยน์เท่านั้นโดยการจับคู่กันด้านทิศทาง เช่น EP1 IN กับ EP1 OUT หรือ EP1 IN กับ EP2 IN

อุปกรณ์ฟูลสปีด นอกจากเอนด์พอยน์ศูนย์แล้ว อุปกรณ์จะมีให้ 30 เอนด์พอยน์ คือมีค่าแอดเดรสตั้งแต่ 1 ถึง 15 ซึ่งแต่ละแอดเดรสสนับสนุนทั้ง IN และ OUT

ทุกๆ ทรานแซคชันที่อยู่บนบัสจะมีการใส่หมายเลขเอนด์พอยน์และโค้ดที่บอกทิศทางการไหลของข้อมูล หรืออาจมีไว้เพื่อบอกว่าทรานแซคชันนี้เป็นจุดเริ่มต้นของการส่งถ่ายข้อมูลแบบคอนโทรล โค้ดดังกล่าวได้แก่ IN, OUT และ SETUP ดังตารางที่ 2.1

ตารางที่ 2.1 โค้ดบอกทิศทางการไหลข้อมูลของทรานแซคชัน

ชนิดของทรานแซคชัน	แหล่งของข้อมูล	ชนิดของการส่งถ่ายข้อมูลที่ใช้ทรานแซคชันชนิดนี้	สิ่งที่อยู่ภายในทรานแซคชัน
IN	อุปกรณ์	ทุกชนิด	ข้อมูลทั่วไป
OUT	โฮส	ทุกชนิด	ข้อมูลทั่วไป
SETUP	โฮส	คอนโทรล	คำร้องขอ

สำหรับทรานแซคชัน IN และ OUT จะยึดมุมมองจากโฮสเป็นหลัก นั่นคือถ้าเป็นทรานแซคชัน IN ข้อมูลจะเดินทางจากอุปกรณ์รอบข้างไปยังโฮส แต่ถ้าเป็นทรานแซคชัน OUT ข้อมูลจะเดินทางจากโฮสไปยังอุปกรณ์รอบข้าง

สำหรับทรานแซคชัน SETUP ข้อมูลจะเริ่มเดินทางจากโฮสไปอุปกรณ์รอบข้างเช่นกัน แต่ทรานแซคชัน SETUP จะเป็นกรณีพิเศษเนื่องจากทรานแซคชันนี้เป็นตัวเริ่มการส่งถ่ายข้อมูลคอนโทรล อุปกรณ์ทุกตัวต้องเข้าใจทรานแซคชันนี้ เพื่อให้มันทราบถึงวิธีการแปลความหมาย

ข้อมูลที่อยู่ภายใน และทรานแซคชันนี้อุปกรณ์จะเป็นฝ่ายรับเสมอ และการส่งถ่ายข้อมูลคอนโทรลจะต้องใช้ทรานแซคชัน SETUP เท่านั้น

ในแต่ละทรานแซคชันจะมีแอดเดรสของอุปกรณ์และเอนด์พอยน์บรรจุไว้ เมื่ออุปกรณ์ได้รับทรานแซคชัน OUT หรือ SETUP ซึ่งบรรจุแอดเดรสของอุปกรณ์ตัวนั้นเอาไว้ ฮาร์ดแวร์ของมันจะเก็บข้อมูลไว้ในเอนด์พอยน์ที่เหมาะสม และทำการกระตุ้นอินเตอร์รัพท์จากนั้นโปรแกรมบริการอินเตอร์รัพท์ในตัวอุปกรณ์จะประมวลผลข้อมูลที่รับเข้ามาและทำงานตามที่ทรานแซตชันนั้นต้องการ ในกรณีที่อุปกรณ์ได้รับทรานแซคชัน IN ซึ่งบรรจุแอดเดรสที่ตรงกับอุปกรณ์ตัวนั้นและถ้ามันพร้อมที่จะส่งข้อมูลไปยังโฮส ฮาร์ดแวร์จะส่งข้อมูลจากเอนด์พอยน์ที่ระบุมาลงไปในบัสและทำการกระตุ้นอินเตอร์รัพท์ จากนั้นโปรแกรมบริการอินเตอร์รัพท์ที่อยู่ในอุปกรณ์จะทำงานต่างๆ ที่จำเป็นเพื่อให้พร้อมสำหรับทรานแซคชัน IN ถัดไป

2.2.2 ไปป์ (Pipe)

ก่อนที่จะทำการส่งถ่ายข้อมูลนั้น โอสจะต้องทำการสร้างไปป์ขึ้นมาก่อน ไปป์ในระบบบัสไม่ใช่วัตถุทางกายภาพแต่เป็นการเชื่อมต่อกันทางลอจิกระหว่างโฮสกับเอนด์พอยน์ ซึ่งใช้งานร่วมกันระหว่างเอนด์พอยน์ของอุปกรณ์และซอฟแวร์ของโฮสคอนโทรลเลอร์

โฮสจะทำการสร้างไปป์ขึ้นในกรณีที่มีการจ่ายไฟให้แก่ระบบ หรือมีการเสียบอุปกรณ์เข้ากับระบบบัสและในกรณีที่มีการร้องขอข้อมูลคอนฟิกกูเรชันจากอุปกรณ์ และโฮสจะทำลายไปป์ที่ไม่ได้ใช้ออกไปเมื่อถอดอุปกรณ์ออกจากระบบบัส นอกจากนี้โฮสอาจร้องขอไปป์ชุดใหม่ หรือทำลายไปป์ที่ไม่ได้ใช้แล้วออกไปที่เวลาใดๆ ได้เมื่อมีคอนฟิกกูเรชันหรืออินเตอร์เฟสชุดใหม่ของอุปกรณ์ อุปกรณ์ USB ทุกตัวจะมีไปป์ที่ชื่อว่าดีฟอลต์คอนโทรลไปป์ (Default Control Pipe) ซึ่งประกอบขึ้นจากเอนด์พอยน์ 2 ชุดคือ EP0 IN และ EP0 OUT โดยไปป์ที่ใช้นี้จะใช้สำหรับตั้งค่าให้กับอุปกรณ์

ข้อมูลคอนฟิกกูเรชันที่โฮสรับเข้ามาจะมีดีสคริปเตอร์ของเอนด์พอยน์แต่ละชุดซึ่งอุปกรณ์ต้องการใช้ โดยเอนด์พอยน์แต่ละชุดจะแจ้งถึงข้อมูลเอนด์พอยน์นั้นๆ ให้แก่โฮสเพื่อใช้ในการสื่อสารกับมัน ซึ่งข้อมูลนี้ประกอบด้วยแอดเดรสของเอนด์พอยน์, ชนิดของการส่งถ่ายข้อมูลที่ใช้, ขนาดสูงสุดของดาต้าแพ็กเก็ต และช่วงเวลาที่ต้องการสำหรับส่งถ่ายข้อมูล

ในบางกรณีโฮสจะรับคำร้องขอคอนฟิกกูเรชันหลังจากแน่ใจแล้วว่าบัสมีแบนวิดธ์ว่างเพียงพอที่จะทำการส่งข้อมูลในอัตราที่ต้องการ ซึ่งกรณีนี้จะเกิดขึ้นเมื่อคอนฟิกกูเรชันนั้นร้องขอไปป์สำหรับเป็นสื่อในการส่งถ่ายข้อมูลแบบไอโซโครนัสที่มีการรับประกันอัตราการส่งข้อมูล และการส่งแบบอินเตอร์รัพท์ที่รับประกันค่าเวลาสูงสุดระหว่างทรานแซคชัน ซึ่งในกรณีนี้โฮสจะทำการตรวจสอบแบนด์วิดธ์ก่อนที่สร้างไปป์ขึ้น ถ้าแบนด์วิดธ์ของระบบบัสมีเพียงพอโฮสจะรับคำร้องขอคอนฟิกกูเรชันนั้น ซึ่งวิธีการนี้จะทำให้มั่นใจว่าการส่งถ่ายข้อมูลจะใช้เวลาตามที่มันต้องการ แต่ถ้าแบนด์วิดธ์ไม่เพียงพอโฮสจะทำการปฏิเสธคำร้องขอนั้นไป และซอฟแวร์ที่ทำการร้องขอจะต้อง

พยายามทำการร้องขอใหม่อีกครั้ง ซึ่งอาจต้องรอจนกระทั่งแบนด์วิดธ์พอเพียง หรือทำการเลือกคอนฟิกกุเรชันใหม่ที่มีแบนด์วิดธ์น้อยกว่า สำหรับไปป์ซึ่งสำหรับการร้องขอการส่งถ่ายข้อมูลที่ไม่ต้องการการรับประกันด้านเวลา โฮสจะไม่ทำการตรวจสอบแบนด์วิดธ์ที่เหลืออยู่แต่มันเพียงตอบรับว่าจะจัดการส่งข้อมูลนั้นในแบนด์วิดธ์ที่เหลืออยู่ให้ดีที่สุดเท่าที่จะทำได้

ในข้อกำหนด USB ได้กำหนดไปป์ไว้ 2 ชนิด ได้แก่ เมสเสจไปป์ และสตรีมไปป์ซึ่งในการใช้งานจะสอดคล้องกับทิศทางการเดินทางของข้อมูลว่าเป็นหนึ่งหรือสองทิศทาง โดยเมสเสจไปป์เป็นไปป์สองทิศทางซึ่งใช้กับการส่งถ่ายข้อมูลแบบคอนโทรลเพียงชนิดเดียวเท่านั้น ส่วนสตรีมไปป์เป็นไปป์แบบทิศทางเดียวซึ่งใช้กับการส่งถ่ายข้อมูลชนิดอื่นๆ นอกเหนือจากการส่งถ่ายข้อมูลแบบคอนโทรล

2.3 การเริ่มส่งถ่ายข้อมูล

เมื่อดีไวซ์ไดร์ฟเวอร์ซึ่งอยู่ที่โฮสต้องการติดต่อสื่อสารกับอุปกรณ์ มันจะเริ่มเข้าสู่กระบวนการส่งถ่ายข้อมูล ในข้อกำหนดได้กำหนดให้การส่งถ่ายข้อมูลเป็นกระบวนการของการสร้างและการกระทำตามคำร้องขอที่เกิดจากการสื่อสาร

ในการเริ่มต้นการสื่อสารแอพพลิเคชันซอฟแวร์จะเรียกฟังก์ชัน API เพื่อร้องขอการส่งถ่ายข้อมูลจากดีไวซ์ไดร์ฟเวอร์ แอพพลิเคชันซอฟแวร์สามารถร้องขอข้อมูลจากอุปกรณ์หรือเป็นตัวจ่ายข้อมูลให้กับอุปกรณ์ก็ได้ เมื่อแอพพลิเคชันซอฟแวร์ร้องขอการส่งถ่ายข้อมูลระบบปฏิบัติการจะผ่านคำร้องไปยังดีไวซ์ไดร์ฟเวอร์ที่เหมาะสม ซึ่งจะส่งคำร้องเหล่านี้ไปยังไดร์ฟเวอร์อื่นๆ ที่อยู่ในระบบและส่งต่อไปยังโฮสคอนโทรลเลอร์ จากนั้นโฮสคอนโทรลเลอร์จึงเริ่มทำการส่งถ่ายข้อมูล

ในบางกรณีไดร์ฟเวอร์จะถูกตั้งค่าไว้เพื่อให้ร้องขอการส่งถ่ายข้อมูลทุกช่วงเวลา และแอพพลิเคชันซอฟแวร์จะอ่านข้อมูลที่รับเข้ามาหรือจ่ายออกไป ส่วนการส่งถ่ายข้อมูลชนิดอื่น เช่น การส่งถ่ายข้อมูลที่ใช้ในกระบวนการอีนิวเมอร์เรท จะเริ่มต้นขึ้นเมื่อมันตรวจพบอุปกรณ์ที่เพิ่มเข้ามาในระบบ

2.4 รูปแบบของแพ็กเก็ต

ภายใน USB แพ็กเก็ตประกอบไปด้วยฟิลด์ต่างๆ ซึ่งแต่ละฟิลด์จะมีข้อมูลเฉพาะตัวตามชนิดของมัน ชนิดของฟิลด์เหล่านี้ได้แก่ SYNC, PID, Address, Endpoint, หมายเลขเฟรม, Data หรือข้อมูล และ CRC ดังตารางที่ 2.2

2.4.1 ฟิลด์ SYNC

แพ็กเก็ตทุกชนิดต้องเริ่มต้นด้วยฟิลด์ SYNC เสมอฟิลด์นี้มีขนาด 8 บิต ใช้สำหรับเข้าจังหวะสัญญาณนาฬิกาของตัวรับและตัวส่ง

ตารางที่ 2.2 ฟิลด์ต่างๆ ภายในแพ็กเก็ต

ชื่อ	ขนาด (บิต)	ชนิดของแพ็กเก็ต	จุดประสงค์การใช้งาน
SYNC	8	ทุกชนิด	ใช้เริ่มแพ็กเก็ตและเข้าจังหวะ
PID	8	ทุกชนิด	แสดงชนิดของแพ็กเก็ต
Address	7	IN, OUT, Setup	แสดงแอดเดรสของฟังก์ชัน
Endpoint	4	IN, OUT, Setup	แสดงเอนด์พอยน์
หมายเลขเฟรม	11	SOF	แสดงเฟรม
Data	0 ถึง 8192 (1024 ไบต์) สำหรับฮาร์ดแวร์ 2.0 และ 0 ถึง 8184 (1023 ไบต์) สำหรับฮาร์ดแวร์ 1.X	DATA0, DATA1	ข้อมูล
CRC	5 หรือ 16	IN, OUT, Setup, DATA0, DATA1	ตรวจจับความผิดพลาด

2.4.2 ฟิลด์ PID (Packet Identifier Field)

ฟิลด์ PID มีขนาด 8 บิต โดยบิต 0 ถึง 3 เป็นตัวบอกชนิดของแพ็กเก็ต และบิต 4 ถึง 7 จะเป็นอินเวิร์สของบิต 0 ถึง 3 ซึ่งใช้สำหรับตรวจสอบความผิดพลาดดังรูปที่ 2.2 ในข้อกำหนด USB ได้กำหนดโค้ดของ PID ไว้ 16 ตัว แยกเป็นกลุ่มของโทเค็น, ดาต้า, แฮนด์เช็ก และแพ็กเก็ตพิเศษ

รูปที่ 2.2 รูปแบบของ PID

2.4.3 ฟิลด์ ADDR (Address Field)

ฟิลด์นี้ใช้สำหรับระบุว่าแพ็กเก็ตนั้นๆ ถูกกำหนดให้ใช้สำหรับอุปกรณ์ตัวไหน เนื่องจากมีความยาวฟิลด์เท่ากับ 7 บิต จึงทำให้สามารถสนับสนุนอุปกรณ์ได้ 127 ตัว สำหรับแอดเดรส 0 จะไม่มีการใช้งาน อุปกรณ์บางตัวที่ยังไม่ได้กำหนดแอดเดรสจะต้องตอบสนองต่อแพ็กเก็ตที่ถูกส่งไปยังแอดเดรส 0

2.4.4 ฟิลด์ ENDP (Endpoint Field)

ฟิลด์นี้มีขนาด 4 บิต ใช้สำหรับบอกถึงหมายเลขเอนด์พอยน์ที่อยู่ภายในอุปกรณ์ ซึ่งอุปกรณ์โลว์สปีดสามารถมีหมายเลขเอนด์พอยน์ได้ไม่เกิน 3 หมายเลข ส่วนอุปกรณ์ฟูลสปีดและไฮสปีดสามารถมีได้ถึง 16 หมายเลข

ตารางที่ 2.3 ชนิดและค่าของ PID

ชนิดของ PID	ค่า PID	คำอธิบาย
โทเค็น	0001	โทเค็น OUT
	1001	โทเค็น IN
	0101	โทเค็น SOF
	1101	โทเค็น SETUP
ดาต้า	0011	DATA0
	1011	DATA1
	0111	DATA2
	1111	MDATA
แฮนด์เช็ก	0010	แฮนด์เช็ก ACK
	1010	แฮนด์เช็ก NAK
	1110	แฮนด์เช็ก STALL
	0110	NYET (No Respond Yet)
พิเศษ	1100	PREamble
	1100	ERR
	1000	Split
	0100	Ping

2.4.5 ฟิลด์หมายเลขเฟรม (Frame Number Field)

มีขนาด 11 บิต ทำหน้าที่เป็นตัวระบุหมายเลขเฟรม โฮสจะส่งแพ็กเก็ตนี้ไปใน SOF ซึ่งเป็นตัวเริ่มต้นเฟรมหรือไมโครเฟรม ตัวเลขมีค่าอยู่ระหว่าง 0 ถึง 7FFh (ถ้ามีค่าเกิน 7FFh ตัวนับจะกลับมาที่ 0 ใหม่) โฮสแบบฟูลสปีดจะใช้ตัวนับ 11 บิตซึ่งมีค่าเพิ่มขึ้น 1 ครั้งต่อเฟรม ส่วนโฮสแบบไฮสปีดจะใช้ตัวนับ 14 บิต และมีค่าเพิ่ม 1 ครั้งต่อไมโครเฟรม แต่จะมีเพียงบิต 3 - 13 ของตัวนับไมโครเฟรมเท่านั้นที่จะถูกส่งลงไปในฟิลด์หมายเลขเฟรมนี้ นั่นก็หมายความว่าจะเพิ่มขึ้น 1 ครั้งต่อ 1 เฟรม หรือ 8 ไมโครเฟรมนั่นเอง

2.4.6 ฟิลด์ดาต้า (Data Field)

ฟิลด์ดาต้ามีขนาดอยู่ระหว่าง 0 ถึง 1024 ไบต์ ขึ้นอยู่กับชนิดการส่งถ่ายข้อมูล, ปริมาณของข้อมูลในทรานแซคชัน, เวอร์ชันของ USB ที่อุปกรณ์ตัวนั้นสนับสนุน

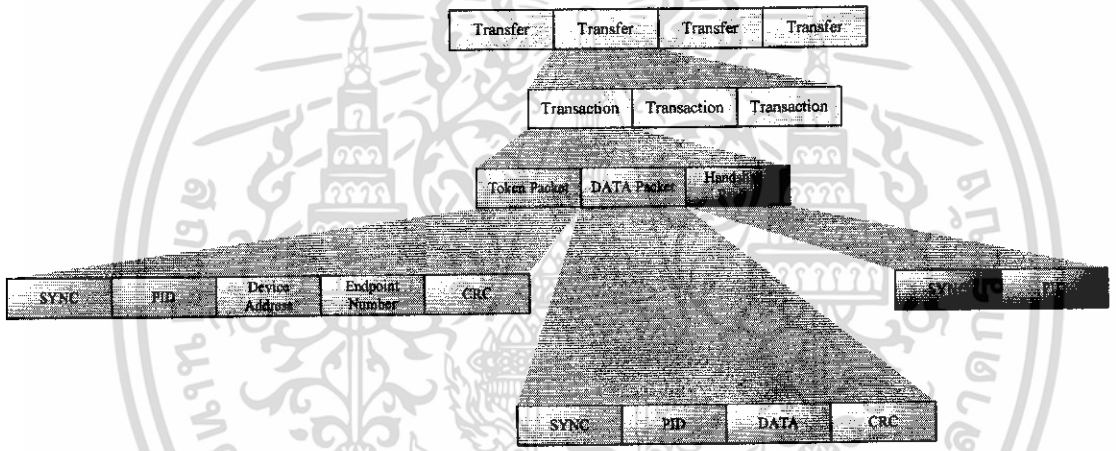
2.4.7 ฟิลด์ CRC (CRC Field)

ฟิลด์ CRC (Cyclic Redundancy Check) ใช้สำหรับตรวจสอบความผิดพลาดของการส่งถ่ายข้อมูล โดยฮาร์ดแวร์ตัวส่งจะแทรกบิต CRC และทางด้านรับก็จะมีฮาร์ดแวร์ที่คำนวณในลักษณะเดียวกับกับด้านส่ง สำหรับโทเค็นแพ็กเก็ตซึ่งครอบคลุมฟิลด์แอดเดรสและเอนด์พอยน์จะมีขนาด 5 บิต ส่วนในกรณีที่ใช้กับดาต้าแพ็กเก็ตจะมีขนาด 16 บิต

2.5 ทรานแซคชัน (Transactions)

ส่วนประกอบของทรานแซคชัน แสดงดังรูปที่ 2.3 ส่วนในตารางที่ 2.4 เป็นรายการองค์ประกอบซึ่งรวมกันเป็นการส่งถ่ายข้อมูลทั้ง 4 แบบ

การส่งถ่ายข้อมูลแต่ละชนิดจะสร้างขึ้นจากทรานแซคชัน 1 ทรานแซคชันหรือมากกว่านั้น และแต่ละทรานแซคชันจะประกอบด้วยแพ็กเก็ต 1, 2 หรือ 3 แพ็กเก็ต



รูปที่ 2.3 องค์ประกอบของทรานแซคชัน

ทรานแซคชันทั้ง 3 ชนิดนี้เกิดขึ้นจากจุดประสงค์การใช้งานและทิศทางการไหลของข้อมูล สำหรับเซ็ตอัพทรานแซคชันจะใช้สำหรับส่งคำร้องขอของการส่งถ่ายข้อมูลคอนโทรลไปยังอุปกรณ์ ส่วนทรานแซคชัน IN เป็นการรับข้อมูลจากอุปกรณ์ และทรานแซคชัน OUT จะใช้สำหรับส่งข้อมูลต่างๆ ไปยังอุปกรณ์ ในข้อกำหนด USB ได้กำหนดให้ทรานแซคชันเป็นการส่งบริการหรือเซอร์วิซ (Service) ไปยังเอนด์พอยน์ ซึ่งเซอร์วิซในที่นี้หมายถึงการส่งก้อนข้อมูลจากโฮสไปยังอุปกรณ์ หรือการร้องขอของอุปกรณ์และการรับก้อนข้อมูลจากอุปกรณ์

ในแต่ละทรานแซคชันประกอบด้วย การแสดงชนิดทรานแซคชัน (Identify), การตรวจสอบความผิดพลาด, สถานะ หรือสเตตัส และข้อมูลคอนโทรล รวมถึงข้อมูลใดๆ ก็ตามที่ต้องการส่งถ่ายกัน การส่งถ่ายข้อมูลให้สมบูรณ์อาจต้องใช้เฟรมหลายเฟรมแต่เนื่องจากทรานแซคชันเป็นการสื่อสารเดี่ยวที่ต้องต่อเนื่องอย่างสมบูรณ์ ดังนั้นจึงไม่มีการสื่อสารใดๆ บนระบบบัสที่สามารถหยุดทรานแซคชันไว้กลางทางได้

ตารางที่ 2.4 องค์ประกอบของการส่งถ่ายข้อมูลแต่ละชนิด

ชนิดของการส่งถ่ายข้อมูล	สเตจ (0 ทรานแซคชัน หรือมากกว่านั้น)	เฟส (แพ็กเก็ต) สำหรับแพ็กเก็ตโลว์สปีดจะตามด้วยแพ็กเก็ต PRE ในแต่ละดาวน์สตรีม
คอนโทรล	เซ็ตอัพ	โทเค็น
		ดาต้า
		แฮนด์เช็ก
	ดาต้า (IN หรือ OUT) (อาจมีหรือไม่มีก็ได้)	โทเค็น
		ดาต้า
		แฮนด์เช็ก
	สเตตัส (IN หรือ OUT)	โทเค็น
		ดาต้า
		แฮนด์เช็ก
บัลก์	ดาต้า (IN หรือ OUT)	โทเค็น
		ดาต้า
		แฮนด์เช็ก
อินเตอร์รัพท์	ดาต้า (IN หรือ OUT)	โทเค็น
		ดาต้า
		แฮนด์เช็ก
ไอโซโครนัส	ดาต้า (IN หรือ OUT)	โทเค็น
		ดาต้า

อุปกรณ์ต้องสามารถตอบสนองด้วยข้อมูลหรืออาจเป็นข้อมูลทางสถานะที่ถูกร้องขอมาในทรานแซคชันให้เร็วที่สุด เฟิร์มแวร์ที่อยู่ภายในอุปกรณ์อาจมีการจัดเตรียมเอนด์พอยน์สำหรับตอบสนองต่อการร้องขอของทรานแซคชันเอาไว้แล้ว แต่ฮาร์ดแวร์จะเป็นตัวจัดการตอบสนองต่อคำร้องขอเมื่อมันถูกส่งมาถึง

การส่งถ่ายข้อมูลด้วยข้อมูลจำนวนน้อยๆ อาจใช้ทรานแซคชันเพียงครั้งเดียว แต่ถ้าข้อมูลมีขนาดใหญ่ การส่งถ่ายข้อมูลอาจต้องแบ่งข้อมูลออกเป็นส่วนๆ แล้วส่งไปกับทรานแซคชันหลายๆ ครั้ง

2.5.1 ทรานแซคชันเฟส (Transaction Phase)

ภายในทรานแซคชันแต่ละครั้งอาจประกอบไปด้วยส่วนซึ่งเกิดขึ้นเป็นลำดับ หรือเรียกว่าเฟส (Phase) ได้มากถึง 3 เฟส ได้แก่ โทเค็น, ดาต้า และแฮนด์เช็ก โดยแต่ละเฟสประกอบด้วยแพ็กเก็ต 1 หรือ 2 แพ็กเก็ต ซึ่งแพ็กเก็ตก็คือบล็อกของข้อมูลที่กำหนดรูปแบบเอาไว้ แพ็กเก็ตทุกชุดจะเริ่มต้นด้วย หมายเลขแพ็กเก็ต (PID) ส่วนที่อยู่ถัดจาก PID ขึ้นอยู่กับชนิดของทรานแซคชัน ซึ่ง

อาจเป็นเอนด์พอยน์แอดเดรส, ดาต้า, ข้อมูลสเตตัส หรือหมายเลขเฟรมร่วมกับบิตตรวจสอบความผิดพลาด

ในส่วนของโทเค็นเฟสนั้นโฮสจะส่งคำร้องขอการสื่อสารไปในโทเค็นแพ็กเก็ต โดยที่ส่วน PID จะบอกถึงชนิดของทรานแซคชัน เช่น Setup, IN, OUT หรือ SOF (Start-Of-Frame)

ในส่วนของดาต้าเฟส โฮสหรืออุปกรณ์ส่งถ่ายข้อมูลชนิดใดๆ ก็ได้ลงในดาต้าแพ็กเก็ต โดยส่วน PID จะเป็นตัวบอกถึงดาต้าทอกเกิล (Data Toggle) ซึ่งใช้ในการบอกตำแหน่งของข้อมูลในกรณีที่มีดาต้าแพ็กเก็ตหลายชุด

ในแฮนด์เช็กเฟส โฮสหรืออุปกรณ์จะส่งข้อมูลสถานะ หรือแฮนด์เช็ก ลงในแฮนด์เช็กแพ็กเก็ต โดย PID จะเก็บโค้ดสถานะเอาไว้ (ACK, NAK, STALL, NYET) บางครั้งในข้อกำหนด USB ใช้คำว่าสเตตัสเฟสและสเตตัสแพ็กเก็ตเพื่ออ้างอิงถึงแฮนด์เช็กเฟสหรือแฮนด์เช็กแพ็กเก็ต

นอกจากนี้โทเค็นเฟสอาจใช้ในการส่งแพ็กเก็ตเริ่มเฟรมหรือ SOF ซึ่งเป็นตัวอ้างอิงทางด้านเวลาโดยในระบบบัสฟูลสปีดโฮสจะส่งมาทุกๆ 1 มิลลิวินาที และในระบบไฮสปีดโฮสจะส่งออกมาทุกๆ 125 ไมโครวินาที ภายในแพ็กเก็ตนี้ยังบรรจุหมายเลขเฟรมซึ่งเป็นตัวบอกถึงการนับเฟรม ซึ่งไมโครเฟรม 8 ชุดที่อยู่ภายในเฟรมเดียวกันจะมีหมายเลขเดียวกันทั้งหมด นอกจากนี้ SOF ยังเป็นตัวที่ช่วยรักษาอุปกรณ์จากการเข้าสู่สภาวะซัซเพนด์เมื่อไม่มีการเคลื่อนไหวของข้อมูลในระบบบัสอีกด้วย

อุปกรณ์โลว์สปีดจะไม่มองแพ็กเก็ต SOF แต่ฮับที่ต่อกับอุปกรณ์จะใช้สัญญาณจบแพ็กเก็ต (End-Of-Frame : EOP) ส่งไปเฟรมละหนึ่งครั้งแทน นอกจากนี้อุปกรณ์โลว์สปีดยังใช้ EOP เป็นตัวที่ช่วยรักษาอุปกรณ์จากการเข้าสู่สภาวะซัซเพนด์อีกด้วย

จากตารางที่ 2.5 สำหรับ PID ชนิดพิเศษ 4 ตัวนั้น ตัวหนึ่งใช้กับอุปกรณ์โลว์สปีดเท่านั้น อีกตัวหนึ่งใช้กับไฮสปีดเพียงอย่างเดียว และอีกสองตัวที่เหลือจะถูกใช้เมื่อฮับ 2.0 ของอุปกรณ์โลว์สปีดหรือฟูลสปีดต้องการติดต่อสื่อสารกับโฮสที่ไฮสปีด

PID ชนิดพิเศษสำหรับโลว์สปีดคือ PRE ซึ่งจะมีโค้ดที่เป็นตัวบอกล่วงหน้าให้แก่ฮับได้ทราบว่าแพ็กเก็ตถัดไปท่ะเข้ามาเป็นโลว์สปีดและฮับจะเปิดการสื่อสารกับอุปกรณ์โลว์สปีดที่ต่ออยู่ภายในระบบ บนระบบบัสโลว์สปีดและฟูลสปีดนั้น PRE จะเป็นส่วนที่ถูกส่งออกมาก่อนแพ็กเก็ตทุกชุด (โทเค็น, ดาต้า และแฮนด์เช็ก) ที่ส่งไปอุปกรณ์โลว์สปีด ส่วนระบบไฮสปีดจะทำการเข้ารหัส PRE เข้าไปในแพ็กเก็ต SPLIT ดังนั้นมันจึงไม่สามารถส่งแยกออกมาต่างหากได้ สำหรับแพ็กเก็ตโลว์สปีดที่ส่งโดยอุปกรณ์นั้นไม่จำเป็นต้องใช้ PRE

ส่วน PID ที่มีใช้เฉพาะอุปกรณ์ไฮสปีด คือ PING โฮสจะส่ง PING ออกมาเพื่อหาว่าดีไวซ์เอนพอยน์แบบไฮสปีดอยู่ในภาวะที่พร้อมหรือไม่ก่อนที่จะมีการส่งดาต้าแพ็กเก็ตถัดไปซึ่งเป็นการส่งถ่ายข้อมูลแบบบัลก์หรือคอนโทรลที่มีดาต้าแพ็กเก็ตหลายชุด

ตารางที่ 2.5 ข้อมูลเกี่ยวกับทรานแซคชันที่ได้จาก PID

ชนิดของแพ็กเก็ต	ชื่อ PID	ค่า	ใช้กับชนิดของการส่งถ่ายข้อมูล	แหล่งกำเนิด	ความเร็วบัส	คำอธิบาย
โทเคิน (แสดงชนิดของทรานแซคชัน)	OUT	0001	ทุกชนิด	โฮส	ทุกความเร็ว	เอนด์พอยน์แอดเดรสสำหรับทรานแซคชัน OUT (โฮสไปอุปกรณ์)
	IN	1001	ทุกชนิด	โฮส	ทุกความเร็ว	เอนด์พอยน์แอดเดรสสำหรับทรานแซคชัน IN (อุปกรณ์ไปโฮส)
	SOF	0101	SOF	โฮส	ทุกความเร็ว	บอกตำแหน่งเริ่มเฟรมและหมายเลขเฟรม
	SETUP	1101	คอนโทรล	โฮส	ทุกความเร็ว	เอนด์พอยน์แอดเดรสสำหรับทรานแซคชัน SETUP
ดาต้า (มีข้อมูลและโค้ดสถานะ)	DATA0	0011	ทุกชนิด	โฮส, อุปกรณ์	ทุกความเร็ว	ดาต้าทอกเกิล, ลำดับข้อมูล
	DATA1	1011	ทุกชนิด	โฮส, อุปกรณ์	ทุกความเร็ว	ดาต้าทอกเกิล, ลำดับข้อมูล
	DATA2	0111	ไอโซโครนัส	โฮส, อุปกรณ์	ไฮสปีด	ลำดับข้อมูล
	MDATA	1111	ไอโซโครนัส, อินเตอร์รัพท์	โฮส, อุปกรณ์	ไฮสปีด	ลำดับข้อมูล
แฮนด์เช็ก (มีโค้ดสถานะ)	ACK	0010	ทุกชนิด	โฮส, อุปกรณ์	ทุกความเร็ว	ผู้รับได้รับข้อมูลที่ปราศจากความผิดพลาด
	NAK	1010	คอนโทรล, บัลก์, อินเตอร์รัพท์	อุปกรณ์	ทุกความเร็ว	ผู้รับไม่สามารถรับข้อมูล หรือผู้ส่งไม่สามารถส่งข้อมูล หรือไม่มีข้อมูลที่จะส่ง
	STALL	1110	คอนโทรล, บัลก์, อินเตอร์รัพท์	อุปกรณ์	ทุกความเร็ว	ไม่มีการสนับสนุนคำร้องขอ คอนโทรลหรือเอนด์พอยน์ถูกทำให้ฮอลท์ (Halt)
	NYET	0110	คอนโทรล Write, บัลก์ OUT, สปลิตทรานแซคชัน	อุปกรณ์	ไฮสปีด	อุปกรณ์ได้รับดาต้าแพ็กเก็ตที่ไม่มีข้อผิดพลาด แต่ขณะนั้นมันยังไม่พร้อม หรือขณะนั้นฮับยังทำสปลิตดาต้าไม่สมบูรณ์
ชนิดพิเศษ	PRE	1100	คอนโทรล, บัลก์, อินเตอร์รัพท์	โฮส	ฟูลสปีด	ตัวบอกล่วงหน้าที่โฮสส่งออกมาเพื่อบอกว่าแพ็กเก็ตต่อไปจะเป็นโลว์สปีด
	ERR	1100	ทุกชนิด	ฮับ	ไฮสปีด	ถูกส่งกลับมาจากฮับเพื่อเป็นการรายงานความผิดพลาดของโลว์หรือฟูลสปีดในการทำสปลิตทรานแซคชัน
	SPLIT	1000	ทุกชนิด	โฮส	ไฮสปีด	นำหน้าโทเค็นแพ็กเก็ตเพื่อบอกว่ามันเป็นสปลิตทรานแซคชัน
	PING	0100	คอนโทรล Write, บัลก์ OUT	โฮส	ไฮสปีด	เช็คการไม่ว่างของทรานแซคชันบัลก์ OUT และคอนโทรล Write หลังจาก NYET
	สงวน	0000	-	-	-	สำหรับการใช้ในอนาคต

สำหรับ SPLIT เป็นตัวที่ใช้บอกว่าโทเค็นแพ็กเก็ตนั้นๆ เป็นส่วนหนึ่งของทรานแซคชันที่เกิดจากการแบ่งออกเป็นส่วนๆ หรือเรียกว่าสปลิตทรานแซคชัน (Split Transaction) ทางด้านโฮสและฮับ 2.0 จะขนส่งข้อมูลที่เป็นโลว์สปีดและฟูลสปีดที่เป็นความเร็วไฮสปีดเพื่อให้การใช้เวลาของบัสมีประสิทธิภาพมากขึ้น เมื่อโฮสเริ่มทรานแซคชันซึ่งมีเป้าหมายของการส่งไปยังอุปกรณ์โลว์สปีดและฟูลสปีด ฮับ 2.0 ที่อยู่ใกล้กับอุปกรณ์ตัวเป้าหมายมากที่สุดจะรับผิดชอบในการทำทรานแซคชันกับอุปกรณ์ให้สมบูรณ์รวมถึงการเก็บข้อมูลหรือสถานะใดๆ ที่ส่งออกกลับมาและรายงานกลับไปยังทรานแซคชัน 1 ครั้งหรือมากกกว่านั้น ด้วยวิธีการนี้ระบบบัสทั้งระบบจะไม่ต้องรอให้ทรานแซคชันที่ความเร็วต่ำกว่าเสร็จสมบูรณ์ ซึ่งทรานแซคชันพิเศษระหว่างฮับกับโฮสนี้จะเรียกว่าสปลิตทรานแซคชัน หรือการแบ่งทรานแซคชัน

ในส่วนของ ERR จะมีใช้เฉพาะในสปลิตทรานแซคชันเท่านั้น ฮับ 2.0 จะใช้ PID นี้เพื่อรายงานความผิดพลาดให้แก่โฮสในทรานแซคชันโลว์สปีดและฟูลสปีด ซึ่ง ERR และ PRE จะมีค่าเหมือนกัน แต่จะสับสนกันเนื่องจากฮับจะไม่มีการส่ง PRE ไปโฮส หรือส่ง ERR ไปยังอุปกรณ์

2.5.2 ลำดับของแพ็กเก็ต

ทรานแซคชันทุกๆ ครั้งจะมีแพ็กเก็ตบรรจุอยู่ โดยโฮสจะเป็นตัวสร้างแพ็กเก็ตชนิดนี้เสมอ ซึ่งทรานแซคชันที่สร้างขึ้นนี้จะประกอบไปด้วยการแสดงชนิดของแพ็กเก็ต, อุปกรณ์เอนด์พอยน์ที่เป็นตัวรับ และทิศทางของข้อมูลใดๆ ซึ่งทรานแซคชันจะทำการส่งถ่ายข้อมูล ในกรณีที่ทรานแซคชันนั้นๆ เป็นแบบโลว์สปีดที่ถูกส่งไปบนระบบบัสแบบฟูลสปีดก็จะต้องมีการส่งแพ็กเก็ต PRE นำหน้าโทเค็นออกมา และถ้าทรานแซคชันนั้นๆ เป็นสปลิตทรานแซคชัน ทรานแซคชันนั้นก็ต้องมีแพ็กเก็ต SPLIT ถูกส่งนำหน้าโทเค็นแพ็กเก็ตออกมาเสมอ

แพ็กเก็ตถัดไปที่ถูกส่งตามโทเค็นแพ็กเก็ตออกมาได้แก่ ดาต้าแพ็กเก็ต แต่แพ็กเก็ตนี้อาจมีหรือไม่มีก็ได้ขึ้นอยู่กับชนิดของการส่งถ่ายข้อมูล และในขณะนั้นอุปกรณ์มีข้อมูลที่จะส่งออกไปหรือไม่ โดยทิศทางที่กำหนดอยู่ในโทเค็นแพ็กเก็ตจะเป็นตัวบอกว่าโฮสหรืออุปกรณ์ที่จะเป็นตัวส่งดาต้าแพ็กเก็ต

ในการส่งถ่ายข้อมูลทุกชนิดยกเว้นไอโซโครนัส อุปกรณ์ที่ได้รับดาต้าแพ็กเก็ตจะส่งแฮนด์เช็กแพ็กเก็ตกลับออกมา ซึ่งมันจะบรรจุโค้ดที่แสดงถึงความสำเร็จหรือความผิดพลาดของทรานแซคชัน

2.5.3 การจัดเวลาในทรานแซคชัน

ภายในทรานแซคชันจะยอมให้เกิดการหน่วงเวลาระหว่างโทเค็นแพ็กเก็ต, ดาต้าแพ็กเก็ต และแฮนด์เช็กแพ็กเก็ตได้เล็กน้อย ซึ่งค่าความหน่วงเวลาที่ยอมให้เกิดขึ้นนี้ถูกออกแบบไว้ในกรณีที่

เกิดความหน่วงเวลาจากสายเคเบิลและค่าเวลาสวิตชิ่ง (Switching Time) รวมกับค่าเวลาอีกเล็กน้อยที่ฮาร์ดแวร์ใช้สำหรับการเตรียมการตอบสนองแพ็กเก็ตที่รับเข้ามา

ขนาดสูงสุดของแพ็กเก็ตในการส่งถ่ายข้อมูลและเอนด์พอยน์แต่ละชนิด จะเป็นตัวจำกัดปริมาณของข้อมูลซึ่งทรานแซคชันสามารถบรรจุเข้าไปได้ การส่งถ่ายข้อมูลหลายทรานแซคชันนั้นอาจต้องใช้จำนวนเฟรมหลายเฟรม แต่ไม่จำเป็นที่ต้องเป็นเฟรมต่อเนื่องกัน เช่นในการส่งถ่ายข้อมูลบัลก์ฟูลสปีดขนาด 512 ไบต์ ซึ่งในการส่งถ่ายข้อมูลชนิดนี้จะมีจำนวนไบต์สูงสุดสำหรับ 1 ทรานแซคชันเพียง 64 ไบต์ ดังนั้นการส่งถ่ายข้อมูลทั้งหมดจำเป็นต้องใช้อย่างน้อย 8 ทรานแซคชัน

2.5.4 การแบ่งทรานแซคชัน (Split Transaction)

ฮับ 2.0 จะติดต่อกับโฮส 2.0 ที่ไฮสปีดถ้าไม่มีฮับ 1.x ต่อคั่นอยู่ระหว่างมัน เมื่ออุปกรณ์แบบโลว์สปีดและฟูลสปีดถูกต่อเข้ากับฮับ 2.0 ฮับจะแปลงความเร็วของการสื่อสารให้เหมาะสมกับความต้องการของอุปกรณ์ ซึ่งการแปลงความเร็วนี้เป็นเพียงหน้าที่หนึ่งของฮับที่จะต้องทำในการจัดการกับการสื่อสารที่ความเร็วหลายๆ แบบ เนื่องจากบัสไฮสปีดจะมีความเร็วมากกว่าฟูลสปีดอยู่ประมาณ 40 เท่า และเร็วกว่าโลว์สปีดอยู่ 320 เท่า ด้วยความเร็วที่แตกต่างกันอย่างมากนี้จึงอาจทำให้เกิดปัญหาขึ้นในระบบ โดยระบบจะต้องรอการส่งข้อมูลในขณะที่ฮับมีการแลกเปลี่ยนข้อมูลระหว่างข้อมูลโลว์สปีดหรือฟูลสปีดกับอุปกรณ์รอบข้าง

ปัญหาดังกล่าวแก้ไขได้โดยการแบ่งทรานแซคชัน หรือสปลิตทรานแซคชัน (รูปที่ 2.4) โฮส 2.0 จะใช้สปลิตทรานแซคชันเมื่อมันทำการติดต่อสื่อสารกับอุปกรณ์โลว์สปีดหรือฟูลสปีดบนบัสแบบไฮสปีด ตามปกติทรานแซคชันหนึ่งๆ ที่สื่อสารระหว่างบัสไฮสปีดกับบัสโลว์สปีดหรือฟูลสปีดจะต้องการสปลิตทรานแซคชัน 2 ชนิด นั่นคือจะมี 1 ทรานแซคชันหรือมากกว่านั้นที่เป็นสตาร์ทสปลิตทรานแซคชัน (Start Split Transaction) เพื่อใช้ในการส่งข้อมูลไปยังอุปกรณ์ และจะมีอีก 1 ทรานแซคชันหรือมากกว่านั้นที่เป็นคอมพลีตสปลิตทรานแซคชัน (Complete Split Transaction) เพื่อใช้ในการรับข้อมูลจากอุปกรณ์ ยกเว้นทรานแซคชันแบบไอโซโครนัส OUT ซึ่งไม่จำเป็นต้องใช้คอมพลีตสปลิตทรานแซคชันเนื่องจากในการส่งข้อมูลแบบนี้จะไม่มีการส่งค่ากลับมา

ถึงแม้ว่าการทำสปลิตทรานแซคชันจะทำให้เกิดทรานแซคชันขึ้นหลายชุดในระบบบัสก็ตาม แต่สปลิตทรานแซคชันจะทำให้การใช้เวลาของบัสดีขึ้น เนื่องจากมันช่วยลดปริมาณการใช้เวลาของบัสในการรอให้อุปกรณ์แบบโลว์สปีดหรือฟูลสปีดตอบสนองกลับมา ตารางที่ 2.6 เป็นการเปรียบเทียบโครงสร้างและเนื้อหาภายในของทรานแซคชันกับอุปกรณ์แบบโลว์สปีดและฟูลสปีดที่ความเร็วที่แตกต่างกัน

1. โฮสทำสตาร์ทสปลิตทรานแซคชันกับฮับ

2. ฮับทำทรานแซคชันกับอุปกรณ์

3. โฮสทำคอมพลีตสปลิตทรานแซคชันกับฮับ

รูปที่ 2.4 สปลิตทรานแซคชัน

ตารางที่ 2.6 ทรานแซคชันเฟสเมื่อมีการสื่อสารบนบัสที่มีความเร็วต่างกัน

ความเร็วบัส	ชนิดของทรานแซคชัน	ทรานแซคชันเฟส		
		โทเค็น	ดาต้า	แฮนด์เช็ก
การสื่อสารโลว์/ฟูลสปีดกับอุปกรณ์	SETUP, OUT	PRE ถ้าเป็นโลว์สปีด, โทเค็น LF/FS	PRE ถ้าเป็นโลว์สปีด, ดาต้า	สเตตัส (ยกเว้นสำหรับไอโซโครนัส)
	IN	PRE ถ้าเป็นโลว์สปีด, โทเค็น LF/FS	ดาต้าหรือสเตตัส	PRE ถ้าเป็นโลว์สปีด, สเตตัส (ยกเว้นสำหรับไอโซโครนัส)
การสื่อสารไฮสปีดระหว่างฮับ 2.0 และโฮสในการทรานแซคชันกับอุปกรณ์โลว์สปีดหรือฟูลสปีด	SETUP, OUT (ไอโซโครนัส OUT จะไม่มีทรานแซคชัน CSPLIT)	SSPLIT, โทเค็น LF/FS	ดาต้า	สเตตัส
		CSPLIT, โทเค็น LF/FS	-	สเตตัส (เฉพาะบัลก์และคอนโทรล)
	IN	SSPLIT, โทเค็น LF/FS	-	-
		CSPLIT, โทเค็น LF/FS	ดาต้าหรือสเตตัส	

สำหรับการทำงานของสปลิตทรานแซคชันในการส่งถ่ายข้อมูลแบบบัลก์และคอนโทรลซึ่งไม่มีการบังคับทางด้านเวลาอย่างเข้มงวดเหมือนกับการส่งถ่ายข้อมูลแบบอินเตอร์รัพท์และไอโซโครนัสสามารถอธิบายได้ดังนี้

ในส่วนของสตาร์ทสปลิตทรานแซคชันจะเริ่มจากโฮส 2.0 ทำการส่งสตาร์ทสปลิตโทเค็นแพ็กเก็ต (Start Split Token Packet : SSPLIT) ออกมาแล้วตามด้วยโทเค็นแพ็กเก็ตตามปกติของโลว์สปีดหรือฟูลสปีด และดาต้าแพ็กเก็ตใดๆ ที่มีจุดหมายไปที่อุปกรณ์ ฮับ 2.0 ที่อุปกรณ์ต่ออยู่จะส่งค่ากลับมาเป็น ACK หรืออาจเป็น NAK ในกรณีที่อุปกรณ์อยู่ในภาวะไม่ว่างหรือไม่มีข้อมูลที่ต้องการส่ง จากนั้นโฮสจะอยู่ในภาวะว่างเพื่อให้สามารถใช้บัสกับทรานแซคชันอื่นๆ ได้ แต่ในขณะนี้อุปกรณ์จะยังไม่รับรู้เกี่ยวกับทรานแซคชันเลย

เมื่อมีการส่ง ACK เพื่อเป็นการตอบสตาร์ทสปลิตทรานแซคชันกลับมา ฮับจะทำหน้าที่ 2 อย่างด้วยกัน อย่างแรกคือมันต้องทำทรานแซคชันกับอุปกรณ์ให้เสร็จสมบูรณ์ และอย่างที่สองคือมันยังต้องจัดการกรับการขนส่งข้อมูลอื่นๆ ที่อยู่บนบัสซึ่งรับมาจากโฮสหรือจากอุปกรณ์ตัวอื่นๆ ที่ต่ออยู่ในระบบ

ในการทำทรานแซคชันให้เสร็จสมบูรณ์ ฮับจะแปลงแพ็กเก็ตที่รับเข้ามาจากโฮสให้อยู่ในความเร็วที่เหมาะสมแล้วส่งไปยังอุปกรณ์ และบางครั้งอาจต้องเก็บการตอบสนองของอุปกรณ์เอาไว้ด้วย อุปกรณ์อาจส่งข้อมูลหรือแฮนด์เช็กกลับออกมา หรือไม่ส่งอะไรออกมาก็ได้ทั้งนี้ขึ้นอยู่กับชนิดของทรานแซคชัน ทรานแซคชันที่มายังอุปกรณ์จะถูกส่งต่อไปที่โลว์สปีดหรือฟูลสปีดตามแต่ชนิดของอุปกรณ์ซึ่งในขณะนี้ถือว่าทรานแซคชันเสร็จสมบูรณ์แล้ว แต่อุปกรณ์จะไม่ทราบว่าสิ่งที่รับเข้ามาเป็นสปลิตทรานแซคชัน และในขณะนี้โฮสยังไม่ได้รับการตอบสนองจากอุปกรณ์

ในขณะที่ฮับกำลังทำทรานแซคชันกับอุปกรณ์ให้เสร็จสมบูรณ์ โฮสอาจต้องเริ่มทำการขนส่งข้อมูลอื่นๆ บนบัส ซึ่งฮับจะต้องเป็นตัวที่ช่วยจัดการด้วยเช่นกัน หน้าที่การทำงานทั้งสองอย่างของฮับจะถูกจัดการโดยโมดูลของฮาร์ดแวร์ที่แยกจากกันภายในตัวฮับ

สำหรับทรานแซคชันทั้งหมดยกเว้นทรานแซคชันแบบไอโซโครนัส OUT เมื่อโฮสเห็นว่าฮับมีเวลาที่เพียงพอในการทำทรานแซคชันกับอุปกรณ์ให้สมบูรณ์ มันก็จะเริ่มทำคอมพลีตสปลิตทรานแซคชันกับฮับ

ในคอมพลีตสปลิตทรานแซคชันนั้น โฮสจะส่งคอมพลีตสปลิตโทเค็นแพ็กเก็ต (Complete Split Token Packet : CSPLIT) ออกมา แล้วตามด้วยโทเค็นแพ็กเก็ตของโลว์สปีดหรือฟูลสปีดตามปกติเพื่อร้องขอข้อมูลหรือสถานะ ซึ่งฮับจะได้รับจากอุปกรณ์อีกต่อหนึ่ง ฮับจะส่งข้อมูลหรือโค้ดสถานะที่โฮสร้องขอมากลับออกไป ซึ่งในขณะนี้ถือว่าทรานแซคชันเสร็จสมบูรณ์แล้ว และทางโฮสไม่จำเป็นต้องส่ง ACK กลับออกมา แต่ถ้าฮับไม่มีแพ็กเก็ตซึ่งพร้อมจะส่งออกมา ฮับจะส่งโค้ดสถานะ NYET กลับออกมาและโฮสจะพยายามใหม่อีกครั้งในภายหลัง ทางด้าน

อุปกรณ์จะไม่รู้เลยว่ามันเป็นคอมพลีตสปลิตทรานแซคชัน เนื่องจากอุปกรณ์ได้ทำทรานแซคชันกับฮับเสร็จไปเสร็จสมบูรณ์ไปก่อนหน้านี้แล้ว

สปลิตทรานแซคชันในการส่งถ่ายข้อมูลอินเตอร์รัพท์และไอโซโครนัสจะไม่มีเฟสของแฮนด์เช็ก ไม่เหมือนกับที่ใช้ในการส่งแบบบัลก์และคอนโทรล ซึ่งในกรณีนี้จะมีเพียงสตาร์ทสปลิตโทเค็นแล้วตามด้วยโทเค็น IN, OUT หรือ SETUP และข้อมูลในกรณีที่มันเป็นทรานแซคชัน OUT หรือเซ็ตอัพทรานแซคชัน

ในส่วนของอินเตอร์รัพท์ทรานแซคชันนั้น ฮับจะทำการจัดตารางเวลาสตาร์ทสปลิตในไมโครเฟรมไว้ก่อนเวลาซึ่งฮับคาดว่าจะเริ่มทรานแซคชันกับอุปกรณ์ ตัวอย่างเช่น สมมติว่าไมโครเฟรมที่อยู่ในเฟรมถูกกำหนดให้ชื่อ Y0 ถึง Y7 ถ้าสตาร์ทสปลิตอยู่ใน Y0 ทรานแซคชันกับอุปกรณ์อาจเกิดขึ้นก่อน Y1 อุปกรณ์อาจมีข้อมูลหรือแฮนด์เช็กที่ใช้ในการตอบสนองกลับไปยังโฮสก่อน Y2 ผลของทรานแซคชันที่ผ่านมาและการทำบิตสตัฟ (Bit Stuff) จะสามารถมีผลต่อมาได้เมื่อทรานแซคชันกับอุปกรณ์เกิดขึ้นจริง ดังนั้นโฮสจะจัดตารางเวลาคอมพลีตทรานแซคชันใน Y2, Y3 และ Y4 ถ้าฮับยังไม่มีข้อมูลที่จะส่งกลับในคอมพลีตมันก็จะส่ง NYET กลับไป และโฮสจะพยายามส่งใหม่อีกครั้ง

ไอโซโครนัสทรานแซคชันแบบฟูลสปีดจะสามารถส่งถ่ายข้อมูลได้ถึง 1023 ไบต์ การทำทรานแซคชันด้วยแพ็กเก็ตขนาดใหญ่จะใช้สตาร์ทสปลิตหรือคอมพลีตสปลิตหลายตัว โดยแต่ละตัวจะมีถึง 188 ไบต์ เพื่อให้มั่นใจได้ว่าการส่งถ่ายข้อมูลสามารถทำได้ทันเวลาหรือทันทีทันใดที่อุปกรณ์มีการส่งหรือพร้อมที่จะรับข้อมูล ซึ่งค่านี้คือปริมาณสูงสุดของข้อมูลฟูลสปีดซึ่งสามารถส่งถ่ายได้ภายในไมโครเฟรม ข้อมูลของทรานแซคชันหนึ่งๆ อาจต้องการสตาร์ทสปลิตหรือคอมพลีตสปลิตทรานแซคชันได้มากถึง 8 ทรานแซคชัน

ในไอโซโครนัสทรานแซคชัน IN โฮสจะจัดเวลาสำหรับคอมพลีตสปลิตทรานแซคชันในทุกๆ ไมโครเฟรมซึ่งมันคาดหมายว่าอุปกรณ์จะมีส่วนของข้อมูลอย่างน้อยส่วนหนึ่งถูกส่งกลับออกมา การร้องขอข้อมูลในส่วนที่เล็กกว่าจะทำให้มั่นใจได้ว่าโฮสรับข้อมูลได้อย่างเร็วที่สุดเท่าที่จะเป็นไปได้ โดยโฮสไม่ต้องรอข้อมูลทั้งหมดที่จะส่งถ่ายจากอุปกรณ์ฟูลสปีดก่อนที่จะมีการเริ่มรับมันเข้ามา

ในไอโซโครนัสทรานแซคชัน OUT โฮสจะส่งข้อมูลในสตาร์ทสปลิตทรานแซคชัน 1 ครั้งหรือมากกว่านั้น โดยโฮสจะจัดเวลาของทรานแซคชันเพื่อไม่ให้บัฟเฟอร์ของฮับว่างเปล่าลง แต่จะบรรจุด้วยไบต์เล็กน้อยเท่าที่จะเป็นไปได้ลงไปแทน ในแต่ละแพ็กเก็ต SPLIT จะมีบิตซึ่งใช้บอกตำแหน่งของข้อมูลในดาต้าแพ็กเก็ตของโลว์สปีดหรือฟูลสปีด (ที่จุดเริ่มต้น, กึ่งกลาง, ตอนท้าน หรือ ทั้งหมด) แต่ในที่นี้จะไม่มีคอมพลีตสปลิตทรานแซคชัน

2.5.5 การตรวจสอบความผิดพลาดในการส่งถ่ายข้อมูล

ระบบบัส USB เราจะใช้การทำแฮนด์เช็กและการตรวจสอบความผิดพลาดของข้อมูลเพื่อช่วยให้มั่นใจได้ว่าการส่งถ่ายข้อมูลทุกครั้งประสบความสำเร็จและไม่มีความผิดพลาดเกิดขึ้น

2.5.5.1 การทำแฮนด์เช็ก (Handshaking)

การทำแฮนด์เช็กแบ่งเป็นการทำแฮนด์เช็กด้วยฮาร์ดแวร์และการทำแฮนด์เช็กด้วยซอฟต์แวร์ สำหรับการทำแฮนด์เช็กด้วยฮาร์ดแวร์จะมีสายสัญญาณเส้นหนึ่งสำหรับเป็นตัวส่งสัญญาณแฮนด์เช็ก เช่น สาย RTS และ CTS ในการเชื่อมต่อแบบ RS-232 สำหรับการทำแฮนด์เช็กด้วยซอฟต์แวร์นั้นจะมีสายสัญญาณเส้นหนึ่งสำหรับส่งโค้ดของแฮนด์เช็กเช่นกัน เช่น โค้ด XON และ XOFF ที่ถูกส่งไปบนสายดาต้าในการเชื่อมแบบ RS-232

ระบบบัส USB จะใช้ซอฟต์แวร์ทำแฮนด์เช็ก โดยใช้โค้ดแฮนด์เช็กเป็นตัวบอกถึงความสำเร็จหรือความผิดพลาดของทรานแซคชันยกเว้นการส่งถ่ายข้อมูลแบบไอโซโครนัส นอกจากนี้ในการส่งข้อมูลแบบคอนโทรลจะมีสเตตัสสเตจเพื่อให้อุปกรณ์สามารถรายงานความสำเร็จหรือความผิดพลาดของการส่งถ่ายข้อมูลทั้งหมดออกมาได้

ตารางที่ 2.7 แหล่งที่มา-สิ่งที่อยู่ภายในดาต้าแพ็กเก็ตและแฮนด์เช็กแพ็กเก็ต

ชนิดของทรานแซคชัน	แหล่งที่มาของดาต้าแพ็กเก็ต	สิ่งที่อยู่ภายในดาต้าแพ็กเก็ต	แหล่งที่มาของแฮนด์เช็กแพ็กเก็ต	สิ่งที่อยู่ภายในแฮนด์เช็กแพ็กเก็ต
Setup	โฮส	ข้อมูล	อุปกรณ์	ACK
OUT	โฮส	ข้อมูล	อุปกรณ์	ACK, NAK, STALL, NYET (เฉพาะในกรณีไฮสปีด), ERR (จากฮับในคอมพลีตสปลิต)
IN	อุปกรณ์	ข้อมูล, NAK, STALL, ERR (จากฮับในคอมพลีตสปลิต)	โฮส	ACK
PING (มีเฉพาะในไฮสปีด)	ไม่มี	ไม่มี	อุปกรณ์	ACK, NAK, STALL

สัญญาณแฮนด์เช็กส่วนใหญ่จะถูกส่งไปในแฮนด์เช็กแพ็กเก็ต แต่จะมีบางครั้งที่ส่งไปในดาต้าแพ็กเก็ต ข้อกำหนด USB ได้กำหนดโค้ดของสเตตัสไว้ 6 ชนิด ได้แก่ ACK, NAK, STALL, NYET และ ERR ส่วนสเตตัสตัวที่ 6 เป็นการไม่ส่งแฮนด์เช็กกลุ่มดังกล่างกลับไป ซึ่งเป็นการบอกว่าเกิดความผิดพลาดรุนแรงที่ระบบบัส ส่วนรายละเอียดและความหมายมีดังนี้

ACK (Acknowledge)

ACK เป็นสเตตัสโค้ดที่ใช้บอกว่าโฮสหรืออุปกรณ์ได้รับข้อมูลอย่างถูกต้องโดยปราศจากความผิดพลาด โดยอุปกรณ์จะส่ง ACK กลับออกมาในแฮนด์เช็กแพ็กเก็ตของเซ็ตอัพทรานแซคชัน นอกจากนี้อุปกรณ์อาจส่ง ACK กลับออกมาในแฮนด์เช็กแพ็กเก็ตของทรานแซคชัน OUT สำหรับทางด้านโฮสจะส่ง ACK กลับออกมาในแฮนด์เช็กแพ็กเก็ตของทรานแซคชัน IN

NAK (Negative Acknowledge)

NAK หมายถึง อุปกรณ์อยู่ในสภาวะไม่ว่าง (Busy) หรือไม่มีข้อมูลที่จะส่งกลับ ถ้าโฮสส่งข้อมูลออกมาในเวลาที่อุปกรณ์อยู่ในสภาวะไม่ว่างมันก็จะไม่สามารถรับข้อมูลนั้นได้ อุปกรณ์จะส่ง NAK กลับออกมาในแฮนด์เช็กแพ็กเก็ต แต่ถ้าโฮสร้องขอข้อมูลจากอุปกรณ์ในขณะที่อุปกรณ์ไม่มีข้อมูลที่จะส่งไป อุปกรณ์จะส่ง NAK ลงมาในดาต้าแพ็กเก็ต ในแต่ละกรณี NAK จะเป็นตัวที่บอกสภาวะชั่วขณะและโฮสพยายามใหม่อีกครั้งได้ในภายหลังและสำหรับด้านโฮสจะไม่มีการส่ง NAK ออกมาแต่อย่างใด ส่วนไอโซโครนัสเอนด์พอยน์จะไม่สนับสนุน NAK เนื่องจากมันจะไม่มีแฮนด์เช็กแพ็กเก็ตสำหรับที่จะส่ง NAK กลับออกมา ถ้าอุปกรณ์หรือโฮสไม่ได้รับข้อมูลไอโซโครนัส มันจะผ่านเลยโดยไม่ตรวจสอบใดๆ ทั้งสิ้น

STALL

แฮนด์เช็ก STALL หมายความได้ 3 อย่าง ได้แก่ การไม่สนับสนุนต่อคำร้องขอคอนโทรล, การร้องขอคอนโทรลไม่สำเร็จ และเกิดความผิดพลาดขึ้นที่เอนด์พอยน์

เมื่ออุปกรณ์ได้รับคำร้องขอจากการส่งถ่ายข้อมูลแบบคอนโทรลซึ่งเอนด์พอยน์นั้นไม่สนับสนุนอุปกรณ์นั้นจะส่ง STALL กลับไปยังโฮส นอกจากนี้บางกรณีอุปกรณ์อาจส่ง STALL ออกไปถ้าอุปกรณ์สนับสนุนคำร้องนั้นๆ แต่ไม่สามารถทำงานตามคำร้องนั้นได้ เช่น ถ้าโฮสส่งคำร้องขอ Set_Configuration ซึ่งร้องขอไปยังอุปกรณ์เพื่อเซ็ตคอนฟิกกุเรชันของมันให้เป็นคอนฟิกกุเรชันหมายเลข 2 แต่อุปกรณ์สนับสนุนคอนฟิกกุเรชันหมายเลข 1 ดังนั้นอุปกรณ์จะส่ง STALL กลับออกมา และเพื่อเป็นการเคลียร์ STALL ที่เกิดขึ้นนี้ ทางโฮสจะทำเพียงแค่ส่งเซ็ตอัพแพ็กเก็ตอีกตัวหนึ่งออกมาเพื่อเป็นการเริ่มการส่งถ่ายข้อมูลคอนโทรลชุดใหม่ ในข้อกำหนด USB จะเรียก STALL ชนิดนี้ว่าเป็น โปรโตคอลสตอล์ (Protocol Stall)

การใช้ STALL ในอีกกรณีหนึ่งจะใช้เพื่อตอบสนองต่อการร้องขอการส่งถ่ายข้อมูลเมื่อแฟล็ก HALT ที่ทำงานร่วมกับเอนด์พอยน์ถูกเซ็ตขึ้นมา ซึ่งเป็นการบอกว่าเอนด์พอยน์ไม่สามารถส่งหรือรับข้อมูลใดๆ ได้ ในข้อกำหนด USB จะเรียก STALL ชนิดนี้ว่า ฟังก์ชันนอลสตอล์ (Functional Stall)

เมื่อโฮสได้รับฟังก์ชันนอลสตอล์จากอุปกรณ์ โฮสจะทำการทิ้งคำร้องขอที่ค้างอยู่ทั้งหมดที่ส่งไปอุปกรณ์และไม่มีการคืนการติดต่ออีกจนกว่ามันจะส่งคำร้องขอในการเคลียร์แฟล็ก HALT บนตัวอุปกรณ์สำเร็จ นอกจากนี้ทางด้านโฮสจะไม่มีการส่ง STALL ออกมา

NYET

NYET จะมีใช้ในอุปกรณ์ไฮสปีดเท่านั้น ซึ่งมันมีความหมายว่าไม่สามารถทำตามคำร้องขอได้ในขณะนั้น การส่งถ่ายข้อมูลบัลก์และคอนโทรลในไฮสปีดจะมีการปรับปรุงโปรโตคอลซึ่งทำให้โฮสสามารถตรวจสอบว่าอุปกรณ์พร้อมที่จะทำการรับข้อมูลหรือไม่ก่อนที่จะทำการส่งออกไป ในระบบบัสแบบโลว์และฟูลสปีดเมื่อโฮสต้องการจะส่งข้อมูลแบบคอนโทรล, บัลก์ หรือ อินเตอร์รัพท์ มันจะส่งโทเค็นแพ็กเก็ตและดาต้าแพ็กเก็ตออกมา และรอรับการตอบกลับจากอุปกรณ์ที่จะส่งมาในแฮนด์เช็กแพ็กเก็ตของทรานแซคชัน ถ้าอุปกรณ์ไม่พร้อมที่จะรับข้อมูลมันจะส่ง NAK กลับออกมาและโฮสจะพยายามใหม่อีกครั้งในภายหลัง การกระทำเช่นนี้อาจทำให้เวลาในบัสสูญเปล่าได้ถ้าดาต้าแพ็กเก็ตมีขนาดใหญ่และอุปกรณ์อยู่ในสภาวะไม่พร้อมบ่อยๆ

สำหรับบัลก์และคอนโทรลทรานแซคชันบนบัสไฮสปีดนั้นการใช้ดาต้าแพ็กเก็ตหลายๆ ชุดจะเป็นวิธีที่ดีกว่า โดยหลังจากที่มีการรับดาต้าแพ็กเก็ตแล้วดีไวซ์เอนด์พอยน์อาจจะส่งแฮนด์เช็ก NYET กลับออกมา ซึ่งหมายความว่าได้มีการรับข้อมูลเข้าไปแล้ว แต่ในขณะนั้นเอนด์พอยน์ยังไม่พร้อมที่รับดาต้าแพ็กเก็ตอีกชุด เมื่อโฮสคิดว่าอุปกรณ์น่าจะอยู่ในสภาวะที่พร้อมแล้ว มันจะส่งโทเค็นแพ็กเก็ต PING ออกไป และเอนด์พอยน์จะทำการส่ง ACK กลับออกมาเพื่อเป็นการตอบตกลงให้มีการส่งดาต้าแพ็กเก็ตถัดไป หรืออาจส่ง NAK หรือ STALL กลับมาถ้ามันไม่ตกลง การส่ง PING จะมีประสิทธิภาพมากกว่าการส่งดาต้าแพ็กเก็ตทั้งหมดเพียงอย่างเดียว และเพื่อเป็นการหาว่าอุปกรณ์พร้อมหรือไม่ จากนั้นโฮสะมีการส่งซ้ำอีกครั้งในภายหลัง

แม้ว่าหลังการตอบสนองต่อ PING หรือ OUT ด้วย ACK แล้วก็ตาม แต่เอนด์พอยน์ยังอนุญาตให้มีการส่ง NAK กลับไปในการรับดาต้าแพ็กเก็ตที่ตามมาได้ ซึ่งโฮสต้องพยายามส่ง PING ไปใหม่อีกครั้ง เหตุการณ์นี้จะสามารถเกิดขึ้นได้แต่ไม่พบบ่อยมากนัก

ในฮับ 2.0 อาจมีการใช้ NYET ในการทำสปลิตทรานแซคชันในส่วนของฟูลสปีดหรือโลว์สปีดยังไม่เสร็จสมบูรณ์ หรือในกรณีที่ฮับไม่สามารถจัดการกับสปลิตทรานแซคชันได้

ERR

แฮนด์เช็ก ERR จะถูกใช้ในไฮสปีดในคอมพลีตสปลิตทรานแซคชันเท่านั้น ซึ่ง ERR หมายถึงอุปกรณ์ไม่ได้ส่งค่าแฮนด์เช็กที่ต้องการกลับมาในทรานแซคชันของฮับการโฮสที่กำลังจะเสร็จสมบูรณ์

การไม่ตอบสนอง (No Response)

โค้ดบอกสถานะตัวสุดท้ายจะเกิดขึ้นเมื่อโฮสหรืออุปกรณ์คาดหมายว่าจะได้รับแฮนด์เช็กแต่มันกลับไม่ได้รับอะไรเลย ซึ่งเหตุการณ์นี้เป็นการบอกถึงการคำนวณผิดพลาดของตัวรับได้ตรวจพบข้อผิดพลาดและแจ้งแก่ผู้ส่งว่าให้มันพยายามส่งกลับมาให้ใหม่อีกครั้ง หรือถ้ามันมีการพยายามส่งหลายครั้งแล้วแต่ยังผิดพลาดให้ไปทำงานอย่างอื่นแทน

2.5.5.2 การรายงานสถานะของการส่งถ่ายข้อมูลคอนโทรล

นอกเหนือจากการรายงานสถานะของทรานแซคชันแล้ว โค้ด ACK, NAK และ STALL ยังถูกใช้ในการรายงานถึงความสำเร็จหรือล้มเหลวของการส่งถ่ายข้อมูลคอนโทรลอีกด้วย นอกจากนี้ยังมีโค้ดสถานะเพิ่มอีกตัวหนึ่งคือ ดาต้าแพ็กเก็ตที่มีความยาวศูนย์ (Zero-Length Data Packet) ซึ่งใช้รายงานความสมบูรณ์ของดาต้าเสตจในการส่งถ่ายข้อมูลคอนโทรลจากโฮสไปอุปกรณ์ ตารางที่ 2.8 แสดงตำแหน่งของตัวบอกสถานะต่างๆ ในการส่งถ่ายข้อมูลแบบคอนโทรล

ตารางที่ 2.8 การรายงานสถานะของการส่งถ่ายข้อมูลคอนโทรล

ชนิดการส่งถ่ายข้อมูลและทิศทาง	ทิศทางของสเตตัสสเตจ	ดาต้าแพ็กเก็ตของ สเตตัสสเตจ	แฮนด์เช็กแพ็กเก็ตของสเตตัสสเตจ
คอนโทรล Write (โฮสส่งข้อมูลไปยังอุปกรณ์)	IN	อุปกรณ์ส่งสถานะ : ดาต้าแพ็กเก็ตความยาวศูนย์ (สำเร็จ), NAK (ไม่ว่าง) หรือ STALL (ผิดพลาด)	โฮสส่ง ACK กลับออกมา
คอนโทรล Read (อุปกรณ์ส่งข้อมูลไปยังโฮส)	OUT	โฮสส่งดาต้าแพ็กเก็ตความยามเป็นศูนย์ (Zero-Length Data Packet)	อุปกรณ์ส่งสถานะ : ACK (สำเร็จ), NAK (ไม่ว่าง) หรือ STALL (ผิดพลาด)

สำหรับการส่งถ่ายข้อมูลคอนโทรล Write นั้นทางด้านอุปกรณ์จะมีการรับข้อมูลในดาต้าสเตจและสถานะของการส่งถ่ายข้อมูลจะถูกส่งกลับมาในดาต้าแพ็กเก็ตของสเตตัสสเตจ ซึ่งถ้ามีการส่งดาต้าแพ็กเก็ตที่มีความยาวเป็นศูนย์กลับไปจะหมายถึงการส่งถ่ายข้อมูลสำเร็จ หรืออุปกรณ์อาจจะส่ง NAK หรือ STALL กลับออกมา จากนั้นโฮสจะส่ง ACK ในแฮนด์เช็กแพ็กเก็ตของสเตตัสสเตจกลับออกไป เพื่อเป็นการบอกว่ามันได้รับการตอบแล้ว

สำหรับการส่งถ่ายข้อมูลคอนโทรล Read นั้นโฮสจะมีการรับข้อมูลในดาต้าสเตจ โดยอุปกรณ์จะส่งค่าสถานะของการส่งถ่ายข้อมูลกลับมาในแฮนด์เช็กแพ็กเก็ตของสเตตัสสเตจ ตามปกติโฮสจะรอรับแพ็กเก็ตทั้งหมดในดาต้าสเตจ จากนั้นจะส่งดาต้าแพ็กเก็ตที่มีความยาวศูนย์มาในสเตตัสสเตจ อุปกรณ์จะตอบกลับมาด้วย ACK, NAK หรือ STALL อย่างไรก็ตาม ถ้าโฮสเริ่มเข้าสู่สเตตัสสเตจก่อนที่ดาต้าแพ็กเก็ตทั้งหมดของอุปกรณ์จะถูกส่งออกไป อุปกรณ์จะต้องออกจากดาต้าสเตจและส่งโค้ดสเตตัสออกมา

2.5.5.3 การตรวจสอบความผิดพลาด

ข้อกำหนดทางด้านฮาร์ดแวร์ของระบบบัส USB ประกอบด้วยส่วนต่างๆ ได้แก่ ตัวขับสัญญาณ, ตัวรับสัญญาณ และสายเคเบิล ในข้อกำหนดนี้จะอธิบายถึงความต้องการทางด้านการออกแบบและคุณสมบัติซึ่งทำให้มั่นใจได้ว่าความผิดพลาดที่เกิดขึ้นจากสัญญาณรบกวนภายในสายจะเกิดขึ้นน้อยที่สุดเท่าที่เป็นไปได้ อย่างไรก็ตามเนื่องจากระบบบัส USB ใช้สายเคเบิลจากภายนอกเชื่อมต่อกับระบบ ดังนั้นการเกิดสัญญาณรบกวนชั่วขณะหรือการทำสายเคเบิลหลุดโดยไม่ตั้งใจจึงอาจเกิดขึ้นได้ ซึ่งเป็นเหตุให้สูญเสียการส่งสัญญาณไปได้ ด้วยเหตุผลนี้เอง แพ็กเก็ตของ USB จึงมีการใส่บิตตรวจสอบความผิดพลาดเอาไว้เพื่อให้ตัวรับสามารถตรวจสอบได้ว่าข้อมูลที่มันรับเข้ามานั้นตรงกับข้อมูลที่ส่งเข้ามาหรือไม่ นอกเหนือจากนี้สำหรับการส่งถ่ายข้อมูลซึ่งต้องทำการทรานแซคชันแต่ละชุด จะมีการใช้ดาต้าท็อกเกิล (Data Toggle) เพื่อทำให้ตัวรับและตัวส่งทำงานได้เข้าจังหวะกัน และทำให้มั่นใจได้ว่าไม่มีทรานแซคชันใดขาดหายไป

2.5.5.3.1 บิตตรวจสอบความผิดพลาด

สำหรับโทเค็นแพ็กเก็ต, ดาต้าแพ็กเก็ต และแพ็กเก็ตเริ่มเฟรม (Start of Frame) จะมีการใส่บิตที่ใช้สำหรับตรวจสอบความผิดพลาดเอาไว้ ค่าของบิตนี้ถูกคำนวณโดยการใช้อัลกอลิธึมทางคณิตศาสตร์ซึ่งเรียกว่า CRC (Cyclic Redundancy Check)

CRC จะถูกใช้กับข้อมูลที่ต้องการตรวจสอบ อุปกรณ์ตัวที่ทำหน้าที่ส่งข้อมูลจะทำการคำนวณ CRC และส่งผลลัพธ์ที่ได้ไปกับข้อมูลด้วย ทางด้านอุปกรณ์ที่ทำหน้าที่รับข้อมูลจะนำข้อมูลที่ได้มาคำนวณในลักษณะเดียวกัน ถ้าผลลัพธ์ที่ได้ตรงกันแสดงว่าข้อมูลที่ได้รับมานี้ไม่มีความผิดพลาด และอุปกรณ์จะทำการส่ง ACK กลับออกไป แต่ถ้าผลลัพธ์ไม่ตรงกันอุปกรณ์ที่เป็นตัวรับจะไม่ส่งแฮนด์เช็กกลับไปซึ่งหมายความว่าให้ด้านส่งทำการส่งข้อมูลใหม่อีกครั้ง

แม้ว่าในข้อกำหนดจะให้ความยืดหยุ่นในการส่งข้อมูลใหม่ แต่โดยทั่วไปแล้วโฮสจะใช้ความพยายามในการส่งข้อมูลใหม่เป็นจำนวน 3 ครั้ง ถ้ายังคงไม่มีสัญญาณแฮนด์เช็กตอบกลับมา *โฮสจะยกเลิกการส่งและแจ้งปัญหาที่เกิดขึ้นกับดีไวซ์ไดร์ฟเวอร์*

ฟิลด์ PID ที่อยู่ในโทเค็นแพ็กเก็ตจะใช้รูปแบบในการตรวจสอบความผิดพลาดที่ง่ายกว่า โดยที่กำหนดให้สี่บิตล่างที่อยู่ในฟิลด์ *PID* และสี่บิตบนเป็นอินเวอร์สหรือคอมพลิเมนต์ของสี่บิตล่างทางด้านผู้รับจะสามารถตรวจสอบความผิดพลาดของข้อมูลได้โดยการคอมพลิเมนต์สี่บิตบนแล้วเทียบกับสี่บิตล่างซึ่งเป็น PID ว่าตรงกันหรือไม่ ถ้าไม่ตรงกันแสดงว่ามีความผิดพลาดเกิดขึ้นและจะถูกละทิ้งไป

2.5.5.3.2 บิตดาต้าท็อกเกิล (Data Toggle Bit)

ในการส่งถ่ายข้อมูลซึ่งต้องการทรานแซคชันหลายๆ ชุดนั้นจะมีการใช้บิตดาต้าท็อกเกิลเพื่อช่วยให้มั่นใจว่าไม่มีทรานแซคชันใดๆ ขาดหายไป ซึ่งมันเป็นตัวรักษาการเข้าจังหวะกันของอุปกรณ์ตัวส่งและตัวรับ บิตดาต้าท็อกเกิลอยู่ในฟิลด์ PID ของดาต้าแพ็กเก็ตที่ใช้สำหรับทรานแซคชัน IN และ OUT นั่นคือ DATA0 และ DATA1 ซึ่ง DATA0 มีค่า PID เท่ากับ 0011 และ DATA1 มีค่า PID เท่ากับ 1011 นั่นคือบิต 3 (บิตซ้ายมือสุด) ของ DATA0 และ DATA1 จะเป็นตัวที่ใช้บอกสภาวะกลับบิต ในบางครั้งบิตนี้อาจถูกเรียกว่า DATA0/1 หรือในบางครั้งอาจเป็น DATA1/0(!)

ทั้งตัวส่งและตัวรับจะคอยติดตามสถานะของบิตดาต้าท็อกเกิล เมื่อมีการตั้งค่าเริ่มต้นใช้งานของอุปกรณ์ ที่บิตนี้ของทั้งสองฝั่งจะถูกเซ็ตให้เป็น DATA0 เมื่อตัวรับตรวจจับได้ว่ามีดาต้าทรานแซคชันเข้ามามันจะทำการเปรียบเทียบบิตดาต้าท็อกเกิลที่ได้รับเข้ามากับสภาวะของดาต้าท็อกเกิลของมันเอง ถ้าบิตนี้มีค่าตรงกัน ตัวรับจะท็อกเกิลให้เป็น DATA1 แล้วส่งแฮนด์เช็กแพ็กเก็ต ACK กลับไปยังตัวส่ง หลังจากที่ตัวส่งได้รับสัญญาณ ACK แล้ว มันจะทำการท็อกเกิลบิตของตัวมันเองให้เป็น DATA1 สำหรับในการส่งแพ็กเก็ตถัดไป (DATA1)

แพ็กเก็ตถัดไปที่ถูกส่งยังตัวรับจะประกอบด้วย DATA1 ซึ่งขณะนี้ทางตัวรับจะมีสถานะของท็อกเกิลบิตเป็น DATA1 เหมือนกัน ดังนั้นตัวรับจะทำการท็อกเกิลบิตของมันแล้วส่ง ACK กลับออกไปอีกครั้ง บิตดาต้าทอกเกิลจะสลับกันไปมาอย่างนี้เรื่อยๆ จนข้อมูลเสร็จสมบูรณ์

ถ้าในขณะใดขณะหนึ่งตัวรับอยู่ในสภาวะไม่ว่าง (Busy) มันจะส่ง NAK กลับออกมา แต่ถ้าตัวรับตรวจพบว่ามีข้อมูลเสียหายมันจะไม่ตอบอะไรกลับไป (No Response) ทางด้านผู้ส่งนั้นถ้าไม่ได้รับ ACK ตอบกลับมา มันจะไม่ท็อกเกิลบิตของมันและทำการส่งข้อมูลพร้อมกับดาต้าท็อกเกิลตัวเดิมอีกครั้ง

ถ้าตัวรับทำการส่ง ACK กลับมาแต่ด้วยเหตุผลบางประการทำให้ตัวส่งมองไม่เห็นสัญญาณ ACK ทางตัวส่งจะคิดว่าตัวรับไม่สามารถรับข้อมูลได้ ตัวส่งจะทำการส่งข้อมูลและบิตดาต้าท็อกเกิลตัวเดิมอีกครั้ง ในกรณีนี้ตัวรับได้รับข้อมูลของดาต้าท็อกเกิลซ้ำกัน มันจะไม่ทำการท็อกเกิลบิตของมันเองและทำการละทิ้งข้อมูลนั้นไป แต่จะส่ง ACK ตอบกลับไป ซึ่งการทำลักษณะนี้มีจุดประสงค์เพื่อให้ดาต้าท็อกเกิลเข้าจังหวะกันใหม่อีกครั้ง

ระบบปฏิบัติการวินโดว์บนโฮสจะจัดการดาต้าท็อกเกิลได้โดยไม่ต้องการโปรแกรมใดๆ เพิ่มเติมจากผู้ใช้เลย นอกจากนี้คอนโทรลเลอร์ชิปของอุปกรณ์รอบข้างบางตัวยังจัดการดาต้าท็อกเกิลโดยอัตโนมัติได้อย่างสมบูรณ์ด้วย ในขณะที่บางตัวอาจต้องการการควบคุมจากเฟิร์มแวร์อยู่บ้าง

ในการส่งข้อมูลไอโซโครนัสแบบฟูลสปีด โฮสจะใช้ดาต้าท็อกเกิล DATA0 เสมอ ซึ่งเหตุผลการส่งถ่ายข้อมูลไอโซโครนัสฟูลสปีดไม่สามารถใช้ดาต้าท็อกเกิลได้เนื่องจากมันไม่มีแฮนด์เช็กแพ็กเก็ตสำหรับส่ง ACK หรือ NAK กลับออกมา และไม่มีเวลาส่งข้อมูลที่หายไปด้วย

ในบางครั้งการส่งถ่ายข้อมูลไอโซโครนัสแบบไฮสปีดจะใช้ DATA0, DATA1 และ PID อื่นๆ ได้แก่ DATA2 และ MDATA การส่งถ่ายข้อมูลไอโซโครนัส IN บนบัสไฮสปีดที่มีทรานแซคชันสองถึงสามชุดต่อไมโครเฟรมจะใช้รูปแบบการวาง DATA0, DATA1 และ DATA2 เพื่อเป็นการบอกถึงตำแหน่งของทรานแซคชันในไมโครเฟรมดังตารางที่ 2.9

ตารางที่ 2.9 รูปแบบการวางดาต้าท็อกเกิลสำหรับการส่งถ่ายข้อมูลไอโซโครนัส IN บนบัสไฮสปีดที่มีทรานแซคชันสองถึงสามชุดต่อไมโครเฟรม

จำนวนของทรานแซคชัน IN ภายในไมโครเฟรม	ดาต้า PID		
	ทรานแซคชันที่ 1	ทรานแซคชันที่ 2	ทรานแซคชันที่ 3
1	DATA0	-	-
2	DATA1	DATA0	-
3	DATA2	DATA1	DATA0

ส่วนการส่งถ่ายข้อมูลไอโซโครนัส OUT บนบัสไฮสปีดที่มีทรานแซคชันสองถึงสามชุดต่อไมโครเฟรมจะใช้รูปแบบการวาง DATA0, DATA1 และ MDATA เพื่อเป็นการบอกว่าภายในไมโครเฟรมนั้นยังมีข้อมูลที่จะตามอีกกี่ชุดดังตาราง

ตารางที่ 2.10 รูปแบบการวางดาต้าท็อกเกิลสำหรับการส่งถ่ายข้อมูลไอโซโครนัส OUT บนบัสไฮสปีดที่มีทรานแซคชันสองถึงสามชุดต่อไมโครเฟรม

จำนวนของทรานแซคชัน OUT ภายในไมโครเฟรม	ดาต้า PID		
	ทรานแซคชันที่ 1	ทรานแซคชันที่ 2	ทรานแซคชันที่ 3
1	DATA0	-	-
2	MDATA	DATA1	-
3	MDATA	MDATA	DATA2

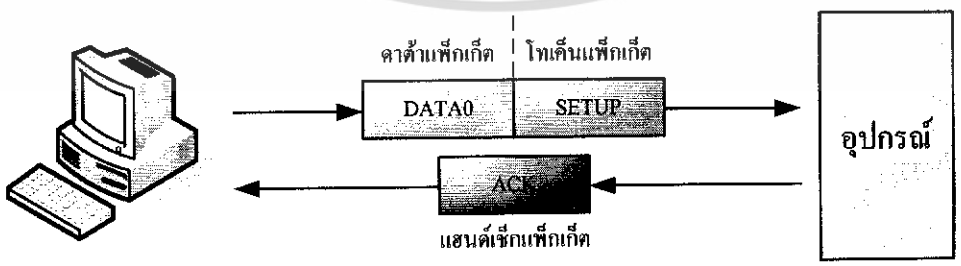
2.6 ชนิดของการส่งถ่ายข้อมูล

ข้อกำหนดของ USB ได้กำหนดรูปแบบการส่งไว้ 4 ชนิด ได้แก่ การส่งถ่ายข้อมูลแบบคอนโทรล, บัลก์, อินเตอร์รัพท์ และไอโซโครนัส ซึ่งแต่ละชนิดมีจุดประสงค์ใช้งานแตกต่างกัน

2.6.1 การส่งถ่ายข้อมูลชนิดคอนโทรล (Control Transfer)

- เป็นการสื่อสารข้อมูลแบบ 2 ทิศทาง
- ใช้สำหรับตั้งค่าการทำงาน, การส่งคำสั่งหรือข้อมูลสถานะ
- การส่งข้อมูลชนิดนี้เริ่มต้นจากโฮสและใช้ความพยายามในการส่งมากที่สุด
- ตรวจสอบความผิดพลาดด้วย CRC (Cyclic Redundancy Check) เพื่อไม่ให้เกิดการผิดพลาดเนื่องจากการส่งข้อมูลชนิดนี้มีความสำคัญมากที่สุด
- ขนาดของข้อมูลที่ใส่ลงในฟิลด์ข้อมูลดาต้าแพ็กเก็ตหรือเรียกว่าดาต้าเพย์โหลด (Data Payload) สำหรับการส่งถ่ายในโลว์สปีดจะมีขนาด 8 ไบท์ ฟูลสปีดมีขนาด 64 ไบท์ และไฮสปีดมีขนาด 8, 16, 32 หรือ 64 ไบท์
- ขั้นตอนการทำงานประกอบไปด้วย 2 ถึง 3 สเตจ ได้แก่ เซ็ตอัพสเตจ (Setup State), ดาต้าสเตจ (Data State : อาจมีหรือไม่มีก็ได้), สเตตัสสเตจ (Status State)

- **เซ็ตอัพสเตจ (Setup State)**
  - เป็นขั้นตอนที่คำร้องขอถูกส่งออกไป ประกอบด้วยแพ็กเก็ต 3 ชุด
  - โทเค็นแพ็กเก็ตจะถูกส่งออกมาเป็นอันดับแรกซึ่งภายในประกอบด้วย PID ที่มีค่าเป็น Setup
  - จากนั้นดาต้าแพ็กเก็ตจะถูกส่งถัดไปซึ่ง PID ของแพ็กเก็ตนี้จะเป็น DATA0 เสมอ ภายในแพ็กเก็ตนี้ประกอบด้วยเซ็ตอัพแพ็กเก็ตซึ่งมีข้อมูลเกี่ยวกับคำร้องขอ รวมถึงหมายเลขของคำร้องขอด้วย
  - ส่วนแพ็กเก็ตสุดท้ายคือ แฮนด์เช็กแพ็กเก็ตซึ่งใช้สำหรับการบอกสถานะของการรับส่งข้อมูลว่าสำเร็จหรือไม่ ถ้าอุปกรณ์รับส่งข้อมูลสำเร็จ (CRC และ PID ถูกต้อง) มันจะตอบกลับมาเป็น ACK อย่างเดียว มิฉะนั้นมันจะละทิ้งข้อมูลนั้นไปและไม่ส่งกลับมายังโฮส ทั้งนี้อุปกรณ์จะไม่มีการส่ง STALL หรือ NAK ในการตอบสนองต่อเซ็ตอัพแพ็กเก็ต



รูปที่ 2.5 เซ็ตอัพสเตจของการส่งถ่ายข้อมูลแบบคอนโทรล

- **ดาต้าสเตจ (Data State)**
  - สเตจนี้อาจมีหรือไม่มีก็ได้
  - ประกอบด้วยการส่งถ่ายข้อมูลแบบ IN หรือ OUT อย่างใดอย่างหนึ่งเป็นจำนวน 1 ครั้งหรือมากกว่านั้น
  - จำนวนของข้อมูลที่ถูกส่งไปในสเตจนี้จะถูกกำหนดไว้ใน Setup Packet
  - ถ้าหากข้อมูลมีมากกว่าขนาดสูงสุดของดาต้าเพย์โหลด ข้อมูลนั้นจะถูกส่งหลายครั้ง โดยแต่ครั้งจะมีขนาดเท่ากับขนาดสูงสุดของดาต้าเพย์โหลด ยกเว้นในการส่งครั้งสุดท้าย
  - ดาต้าสเตจมีบทบาทต่างกันอยู่ 2 อย่างโดยขึ้นอยู่กับทิศทางการส่งถ่ายข้อมูลได้แก่
    1. **IN :** เกิดขึ้นในกรณีการส่งถ่ายข้อมูลแบบคอนโทรล Read โดยเมื่อโฮสพร้อมรับข้อมูลคอนโทรล โฮสจะส่งโทเค็น IN ออกมา ถ้าอุปกรณ์ได้รับโทเค็น IN ที่มีความผิดพลาด เช่น PID กับส่วนอินเวิร์สของมันไม่สอดคล้องกัน อุปกรณ์จะละทิ้งแพ็กเก็ตนี้ไป แต่ถ้าโทเค็นที่ได้รับมาถูกต้อง อุปกรณ์อาจตอบกลับมาด้วยดาต้าแพ็กเก็ตซึ่งบรรจุข้อมูลคอนโทรลที่จะส่ง หรืออาจเป็นแพ็กเก็ต STALL เพื่อเป็นการบอกว่าเกิดการผิดพลาดขึ้นที่เอนด์พอยน์ หรืออาจเป็น NAK เพื่อเป็นการบอกโฮสว่าเอนด์พอยน์สามารถทำงานได้ตามปกติแต่ในขณะนั้นไม่มีข้อมูลที่จะส่ง
    2. **OUT :** เกิดในการส่งถ่ายข้อมูลแบบคอนโทรล Write เมื่อโฮสต้องการส่งคอนโทรลไปยังอุปกรณ์ มันจะส่งโทเค็น OUT ออกมาแล้วตามด้วยดาต้าแพ็กเก็ตซึ่งบรรจุดาต้าเพย์โหลดของข้อมูลคอนโทรลเอาไว้ ถ้าบอกส่วนของ OUT หรือดาต้าแพ็กเก็ตไม่ถูกต้อง อุปกรณ์จะละทิ้งแพ็กเก็ตนี้ไป ถ้าเอนด์พอยน์บัฟเฟอร์ของอุปกรณ์ว่างและสามารถป้อนข้อมูลเข้าไปในบัฟเฟอร์ได้ อุปกรณ์จะส่ง ACK ออกมาเพื่อบอกให้โฮสทราบว่ามันรับข้อมูลสำเร็จแล้ว แต่ถ้าเอนด์พอยน์บัฟเฟอร์ไม่ว่างเนื่องจากการประมวลผลของแพ็กเก็จก่อนหน้านี้ยังไม่เสร็จสมบูรณ์มันก็จะทำการส่ง NAK กลับไป แต่อย่างไรก็ตามมันอาจส่งค่า STALL กลับออกมาถ้าเอนด์พอยน์มีข้อผิดพลาดและบิต HALT ถูกเซ็ตอยู่

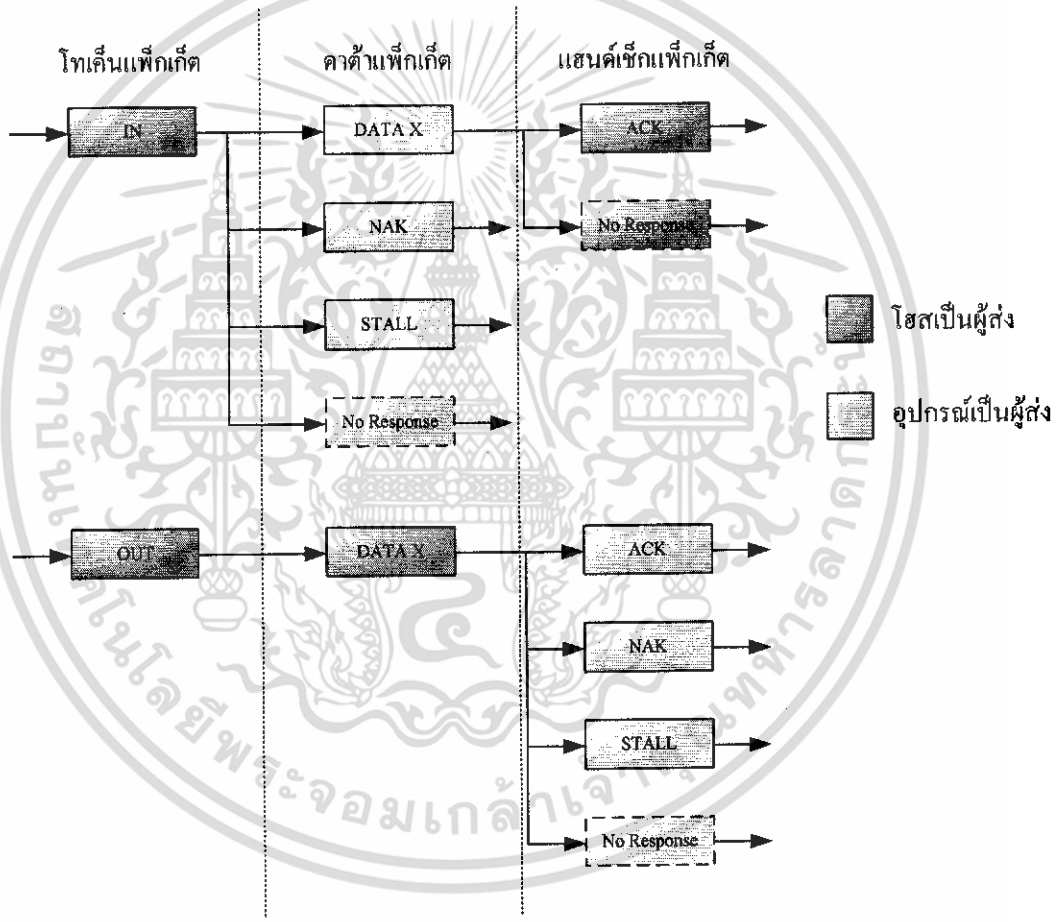
- **สเตตัสสเตจ (Status State)**

เป็นสเตจที่ใช้รายงานสถานะของการร้องขอทั้งหมด และมีทิศทางแปรเปลี่ยนไปตามการเปลี่ยนแปลงทิศทางการส่งถ่ายข้อมูล ซึ่งการรายงานสถานะจะกระทำจากอุปกรณ์เสมอ

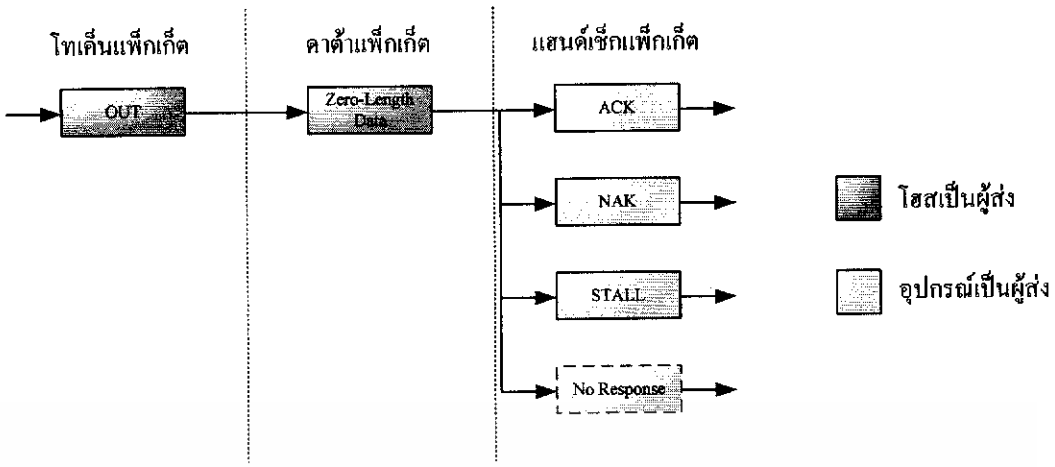
- ถ้าการส่งถ่ายข้อมูลคอนโทรลครั้งนั้นๆ ไม่มีดาต้าสเตจ อุปกรณ์จะตอบกลับด้วย ACK แต่ถ้าข้อมูลมีความผิดพลาด มันจะละทิ้งข้อมูลนั้นไปและไม่ส่งแฮนด์เช็กแพ็กเก็ตกลับไปยังโฮส
- กรณีที่เป็นคอนโทรล Read โฮสจะต้องตอบรับ (Acknowledge) ว่าการรับข้อมูลนี้ทำได้สำเร็จ โดยโฮสต้องทำการส่งโทเค็น OUT แล้วตามด้วยดาต้าแพ็กเก็ตที่มีความยาวเป็นศูนย์ ในขณะที่อุปกรณ์จะสามารถรายงานสถานะของมันลงมาในแฮนด์เช็กสเตจด้วย ACK ซึ่ง

เป็นการบอกว่าอุปกรณ์ได้ทำตามคำสั่งโดยสมบูรณ์แล้วและพร้อมที่จะรับคำสั่งถัดไป ถ้าเกิดการผิดพลาดขึ้นในขณะที่ทำการประมวลผลคำสั่งนี้อุปกรณ์จะส่ง STALL ออกมา อย่างไรก็ตามถ้าอุปกรณ์ยังคงประมวลผลต่อไปมันก็จะส่ง NAK กลับออกมาเพื่อแจ้งให้โฮสทำสเตตัสสเตจอีกครั้งในภายหลัง

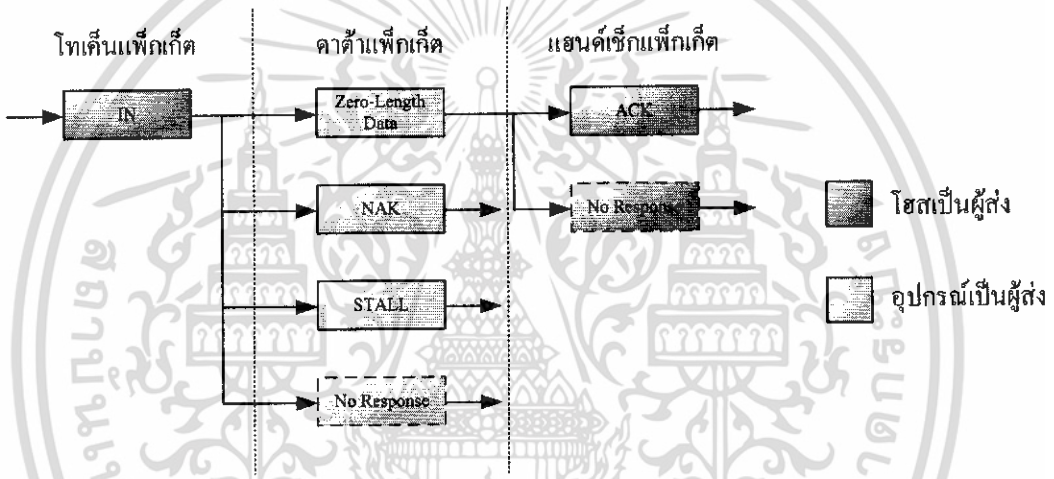
- กรณีที่เป็นคอนโทรล Write อุปกรณ์จะทำการตอบรับว่าการรับข้อมูลทำได้สำเร็จโดยการส่งแพ็กเก็ตที่มีความยาวเป็นศูนย์ในการตอบสนองโทเค็น IN อย่างไรก็ตามถ้ามีข้อผิดพลาดเกิดขึ้นมันจะส่ง STALL ออกมา หรือถ้าอุปกรณ์ยังอยู่ในระหว่างการประมวลผลข้อมูลอยู่มันก็จะส่ง NAK กลับออกมาเพื่อบอกโฮสให้เข้าสเตตัสสเตจอีกครั้งในภายหลัง



รูปที่ 2.6 ดาต้าสเตจของการส่งถ่ายข้อมูลคอนโทรล



รูปที่ 2.7 สเตตัสสเตจของการส่งถ่ายข้อมูลคอนโทรล Read



รูปที่ 2.8 สเตตัสสเตจของการส่งถ่ายข้อมูลคอนโทรล Write

2.6.2 การส่งถ่ายข้อมูลอินเตอร์รัพท์ (Interrupt Transfer)

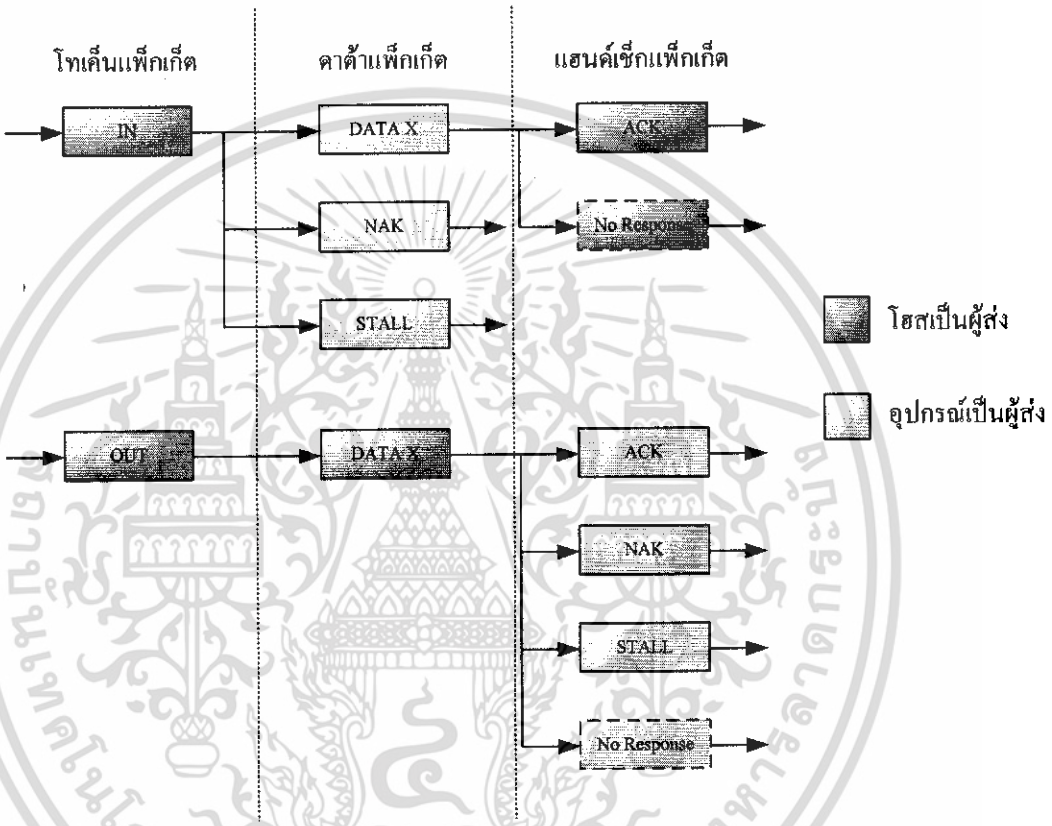
การส่งถ่ายข้อมูลอินเตอร์รัพท์ในระบบบัส USB มีความหมายไม่เหมือนกับการอินเตอร์รัพท์ที่ใช้กันในระบบไมโครคอนโทรลเลอร์ซึ่งเป็นสิ่งที่อุปกรณ์รอบข้างสร้างขึ้น แต่ในระบบบัส USB จะใช้สำหรับส่งสัญญาณสอบถามหรือโพลอุปกรณ์ต่างๆ เพื่อทำการหาว่าอุปกรณ์ตัวไหนต้องการรับส่งข้อมูลกับโฮส ซึ่งถ้าอุปกรณ์ต้องการติดต่อกับโฮส มันจะต้องรอจนกระทั่งโฮสส่งสัญญาณโพลไปที่มันก่อน จากนั้นจึงบอกความต้องการในการรับส่งข้อมูลกับโฮส

คุณสมบัติของการส่งถ่ายข้อมูลอินเตอร์รัพท์

- มีการรับประกันความถูกต้องของข้อมูล
- ใช้สตรีมไปป์ (ทิศทางเดียว)
- มีการตรวจสอบความผิดพลาดของข้อมูล

ตามปกติการส่งถ่ายข้อมูลอินเตอร์รัพท์จะไม่เป็นช่วงเวลา คำร้องอินเตอร์รัพท์จะถูกจัดไว้ในตัวอุปกรณ์จนกระทั่งโฮสทำการโพลไปยังอุปกรณ์ USB ตัวนั้นเพื่อสอบถามข้อมูล สำหรับขนาดสูงสุดของดาต้าเพย์โหลดของโหมดการสื่อสารต่างๆ ดังนี้

- ขนาสูงสุดของข้อมูลภายในดาต้าเพย์โหลดสำหรับอุปกรณ์โลว์สปีดมีค่า 8 ไบต์
- ขนาสูงสุดของข้อมูลภายในดาต้าเพย์โหลดสำหรับอุปกรณ์ฟูลสปีดมีค่า 64 ไบต์
- ขนาสูงสุดของข้อมูลภายในดาต้าเพย์โหลดสำหรับอุปกรณ์ไฮสปีดมีค่า 1024 ไบต์



รูปที่ 2.9 เฟสของการส่งถ่ายข้อมูลอินเตอร์รัพท์

ทรานแซคชันของอินเตอร์รัพท์อาจประกอบไปด้วยการส่งถ่ายข้อมูล IN และ OUT ดังนี้

- **IN :**
  - โฮสจะส่งโพลไปที่อินเตอร์รัพท์เอนด์พอยน์อย่างสม่ำเสมอ ซึ่งอัตราการโพลนี้จะระบุไว้ในเอนด์พอยน์ดีคริปเตอร์
  - การโพลแต่ละครั้งจะทำให้โฮสส่งโทเค็น IN ออกไป ถ้าโทเค็น IN เกิดความผิดพลาดอุปกรณ์จะละทิ้งแพ็กเก็ตนั้น และคอยตรวจสอบบัสเพื่อหาโทเค็นตัวใหม่
  - ถ้าคำร้องขออินเตอร์รัพท์ถูกจัดเตรียมไว้แล้วในอุปกรณ์ เมื่อมันได้รับโทเค็น IN มันจะส่งดาต้าแพ็กเก็ตซึ่งบรรจุข้อมูลที่เกี่ยวข้องกับการอินเตอร์รัพท์ออกไป
  - หลังจากที่โฮสรับข้อมูลได้สำเร็จมันจะส่ง ACK กลับมา แต่ถ้าข้อมูลเกิดการผิดพลาดมันจะไม่ส่งสถานะใดๆ กลับออกมา

- อีกกรณีหนึ่งคือถ้าอุปกรณ์ไม่ต้องการอินเตอร์รัพท์ เมื่อโฮสโพลไปที่อินเตอร์รัพท์เอนด์พอยน์ด้วยโทเค็น IN อุปกรณ์จะให้สัญญาณ NAK ออกมา แต่ถ้าเกิดข้อผิดพลาดขึ้นที่เอนด์พอยน์ อุปกรณ์จะส่ง STALL ตอบโทเค็น IN กลับมา

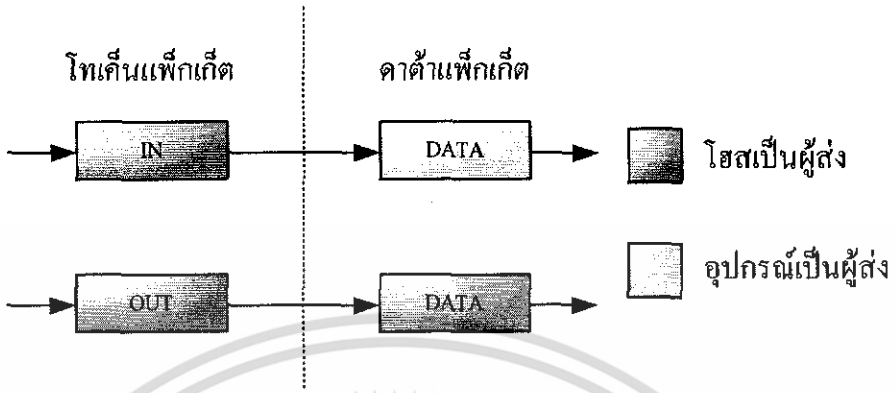
- **OUT :**
  - เมื่อโฮสต้องการส่งข้อมูลอินเตอร์รัพท์อุปกรณ์ มันจะส่งโทเค็น OUT ออกมา แล้วตามด้วยดาต้าแพ็กเก็ตซึ่งบรรจุข้อมูลการอินเตอร์รัพท์ไว้
  - ถ้าบางส่วนของโทเค็น OUT หรือดาต้าแพ็กเก็ตเกิดความผิดพลาดขึ้น อุปกรณ์จะละทิ้งดาต้าแพ็กเก็ตนี้ไป
  - ถ้าเอนด์พอยน์บัฟเฟอร์ของอุปกรณ์มีพื้นที่ว่างและมันมีการป้อนข้อมูลเข้าไปในเอนด์พอยน์บัฟเฟอร์ อุปกรณ์จะส่ง ACK ออกไปเพื่อแจ้งให้โฮสทราบว่าได้ทำการรับข้อมูลสำเร็จแล้ว
  - ถ้าเอนด์พอยน์บัฟเฟอร์ของอุปกรณ์ไม่มีพื้นที่ว่างเนื่องจากการประมวลผลแพ็กเก็ตก่อนหน้านี้ อุปกรณ์จะส่ง NAK กลับออกมา
  - แต่ถ้าเกิดความผิดพลาดขึ้นที่เอนด์พอยน์และบิต Halt ถูกเซ็ตให้เป็น 1 อุปกรณ์จะส่ง STALL กลับออกมา

2.6.3 การส่งถ่ายข้อมูลไอโซโครนัส (Isochronous Transfer)

การส่งถ่ายข้อมูลแบบไอโซโครนัสเป็นการส่งข้อมูลที่เกิดขึ้นอย่างต่อเนื่องและเป็นช่วงเวลาที่แน่นอน ตามปกติใช้กับข้อมูลที่มีการเปลี่ยนแปลงต่อเนื่องตามเวลา เช่น สตรีมของข้อมูลเสียงและภาพ สำหรับคุณสมบัติการส่งถ่ายข้อมูลแบบไอโซโครนัสมีดังนี้

- มีการรับประกันการเข้าถึงแบนวิดธ์ของบัส
- ใช้สตรีมไปป์ (มีทิศทางเดียว)
- มีการตรวจจับข้อผิดพลาดโดยใช้ CRC แต่ไม่มีการส่งซ้ำหรือรับประกันการส่ง
- ใช้กับโหมดฟูลสปีดและไฮสปีดเท่านั้น
- ไม่มีการทำดาต้าท็อกเกิล
- ขนาดสูงสุดของข้อมูลในดาต้าเพย์โหลดจะถูกระบุไว้ในเอนด์พอยน์ดีสคริปเตอร์ของไอโซโครนัสเอนด์พอยน์ ซึ่งสามารถมีค่าได้สูงสุด 1023 ไบต์ในกรณีฟูลสปีดและ 1024 ไบต์สำหรับอุปกรณ์ไฮสปีด
- เนื่องจากขนาดสูงสุดของดาต้าเพย์โหลดอาจส่งผลกระทบต่อความต้องการทางแบนด์วิดธ์ของระบบบัสเป็นอย่างมาก ดังนั้นการจำกัดขนาดของดาต้าเพย์โหลดจึงเป็นสิ่งที่ควรกระทำ นั่นคือถ้าหากต้องการใช้ดาต้าเพย์โหลดขนาดใหญ่ในการสื่อสารข้อมูล เราควรสร้างทางเลือกของอินเตอร์เฟสที่มีขนาดของดาต้าเพย์โหลดที่มีขนาดแตกต่างกันไป ซึ่งจะเป็นผลดีถ้าในขณะที่ทำการอีนิวเมอร์เรทนั้น โฮสไม่

สามารถจัดหาแบนด์วิดธ์ให้แก่ไอโซโครนัสที่ใช้ดาต้าเพย์โหลดขนาดใหญ่ได้ โฮสจะมีทางเลือกในการใช้อินเตอร์เฟสทางอื่นซึ่งใช้แบนด์วิดธ์ของไอโซโครนัสที่น้อยกว่าได้



รูปที่ 2.10 เซ็ตอัพสเตจของการส่งถ่ายข้อมูลไอโซโครนัส

รูปแบบของทรานแซคชันแบบไอโซโครนัส IN และ OUT แสดงดังรูป 2.10 ซึ่งทรานแซคชันแบบนี้ไม่มีสเตจสำหรับการทำแฮนด์เช็กและไม่สามารถรายงานความผิดพลาดได้ (STALL และ HALT)

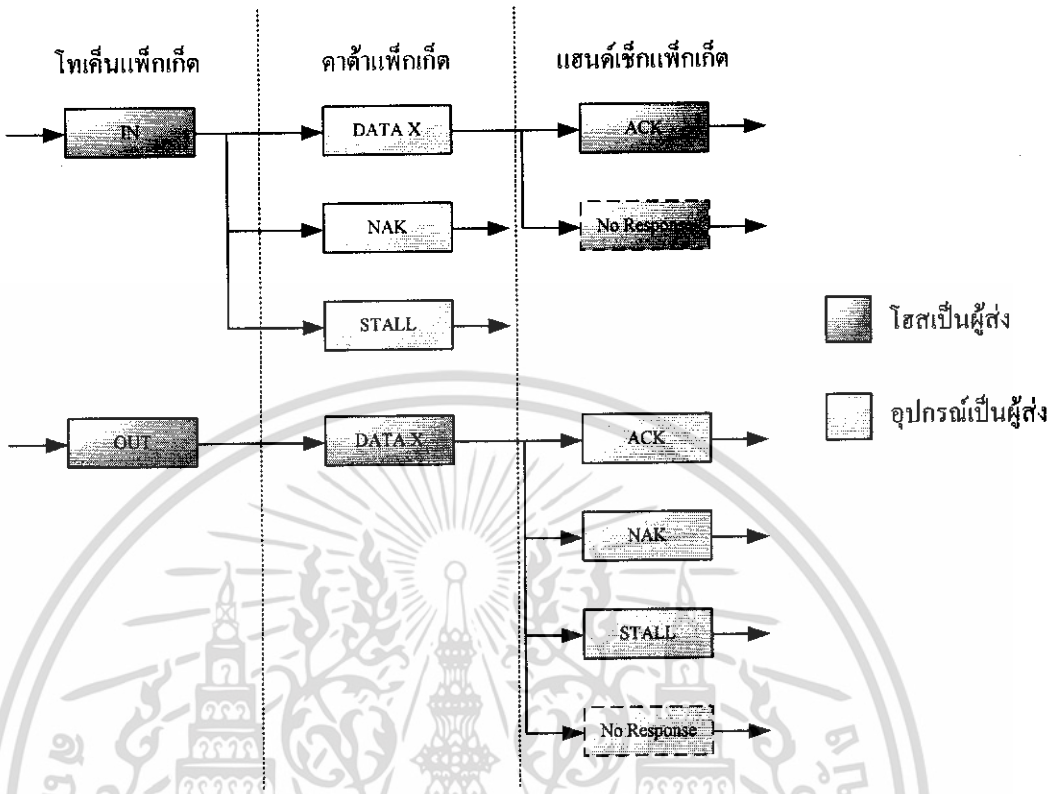
2.6.4 การส่งถ่ายข้อมูลบัลก์ (Bulk Transfer)

การส่งถ่ายข้อมูลแบบบัลก์ถูกออกแบบมาให้ใช้สำหรับการรับส่งข้อมูลที่มีปริมาณมากๆ ซึ่งต้องมีการประกันความถูกต้องของข้อมูล แต่เวลาที่ใช้ในการส่งนั้นไม่ใช่สิ่งสำคัญ ตัวอย่างการรับส่งข้อมูลแบบบัลก์ได้แก่ ข้อมูลจากปริ้นเตอร์ หรือข้อมูลจากเครื่องสแกนเนอร์ การส่งถ่ายข้อมูลแบบบัลก์มีการแก้ไขความผิดพลาดด้วยการใช้ CRC16 และด้วยกลไกการตรวจจับความผิดพลาดรวมถึงการส่งข้อมูลซ้ำ ทำให้มั่นใจได้ว่าข้อมูลที่ได้รับและส่งนั้นจะไม่มีความผิดพลาดเกิดขึ้นเลย

การส่งถ่ายข้อมูลแบบบัลก์จะใช้แบนด์วิดธ์ของบัสที่เหลือภายหลังจากที่ทรานแซคชันอื่นได้ทำการจองเสร็จเรียบร้อยแล้ว ถ้าระบบบัสอยู่ในภาวะที่ไม่ว่างเนื่องจากมีการส่งถ่ายข้อมูลแบบไอโซโครนัสหรืออินเตอร์รัพท์อยู่อาจส่งผลให้การส่งถ่ายข้อมูลแบบบัลก์มีความล่าช้าลงไป ดังนั้นการส่งถ่ายข้อมูลแบบนี้จึงเหมาะกับข้อมูลที่ไม่ขึ้นเปลี่ยนแปลงตามเวลา สำหรับคุณสมบัติของการส่งถ่ายข้อมูลแบบบัลก์มีดังนี้

- สามารถใช้กับข้อมูลที่มีขนาดใหญ่มากๆ ได้
- มีการตรวจจับความผิดพลาดโดยใช้ CRC และมีการรับประกันการส่งได้ว่าสามารถรับส่งข้อมูลได้อย่างครบถ้วนและถูกต้อง
- ไม่มีการรับประกันด้านแบนด์วิดธ์
- ใช้สตรีมไปป์ (มีทิศทางเดียว)
- ใช้เฉพาะในโหมดฟูลสปีดและไฮสปีด

- ขนาดสูงสุดของแพ็กเก็ตในอุปกรณ์ฟูลสปีดมีขนาดเท่ากับ 8, 16, 32 หรือ 64 ไบต์ และในอุปกรณ์ไฮสปีดสามารถมีขนาดเท่ากับ 512 ไบต์



รูปที่ 2.11 เฟสการส่งถ่ายข้อมูลบัลก์

ทรานแซคชันของบัลก์อาจประกอบไปด้วยการส่งถ่ายข้อมูล IN และ OUT ดังนี้

- IN : เมื่อโฮสพร้อมรับส่งข้อมูลบัลก์มันจะส่งโทเค็น IN ออกมา ถ้าอุปกรณ์ได้รับโทเค็น IN แล้วเกิดความผิดพลาดมันจะละทิ้งแพ็กเก็ตนี้ไป แต่ถ้าอุปกรณ์ได้รับโทเค็นอย่างถูกต้องมันอาจจะทำการตอบกลับด้วยแพ็กเก็ต DATA ซึ่งบรรจุข้อมูลบัลก์ที่ต้องการส่ง หรืออาจส่งแพ็กเก็ต STALL เพื่อเป็นการบอกว่ามีการผิดพลาดขึ้นที่เอนด์พอยน์ หรืออีกกรณีหนึ่งอุปกรณ์ส่ง NAK ออกมาเพื่อเป็นการบอกแก่โฮสว่าเอนด์พอยน์นั้นๆ สามารถทำงานได้แต่ไม่มีข้อมูลที่จะทำการส่งในขณะนั้น
- OUT : เมื่อโฮสต้องการส่งข้อมูลบัลก์ มันจะส่งโทเค็น OUT ออกมาแล้วตามด้วยดาต้าแพ็กเก็ตซึ่งบรรจุข้อมูลบัลก์เอาไว้ ถ้าบางส่วนของโทเค็น OUT หรือดาต้าแพ็กเก็ตเกิดการผิดพลาดขึ้นมาอุปกรณ์จะละทิ้งแพ็กเก็ตนี้ไป ถ้าเอนด์พอยน์บัฟเฟอร์ของอุปกรณ์มีพื้นที่ว่างและมีการป้อนข้อมูลเข้าไปในเอนด์พอยน์บัฟเฟอร์ อุปกรณ์ก็จะส่ง ACK กลับออกมาเพื่อแจ้งแก่โฮสว่าได้ทำการรับข้อมูลสำเร็จแล้ว ถ้าเอนด์พอยน์บัฟเฟอร์ไม่มีพื้นที่ว่างเนื่องจากมีการประมวลผลแพ็กเก็ตก่อนหน้านี้ อุปกรณ์ก็จะส่ง NAK กลับออกมา แต่ถ้าเอนด์พอยน์เกิดความผิดพลาดขึ้นและบิต HALT ของมันถูกเซ็ต อุปกรณ์จะส่ง STALL กลับออกมา

ตารางที่ 2.11 การส่งถ่ายข้อมูลของ USB แต่ละชนิด

ชนิดของข้อมูล	คอนโทรล	บัลก์	อินเตอร์รัพท์	ไอโซโครนัส
การใช้งาน	การตั้งค่า	พริ้นเตอร์, สแกนเนอร์	เมาส์, คีย์บอร์ด	เสียง
ความจำเป็น	จำเป็น	ไม่จำเป็น	ไม่จำเป็น	ไม่จำเป็น
การสนับสนุนของอุปกรณ์โลว์สปีด	สนับสนุน	ไม่สนับสนุน	สนับสนุน	ไม่สนับสนุน
ทิศทางการไหลของข้อมูล	IN และ OUT	IN หรือ OUT	IN หรือ OUT (เวอร์ชัน 1.0 จะสนับสนุนเฉพาะ IN)	IN หรือ OUT
ปริมาณการจองแบนด์วิดธ์สูงสุด	10% ของเฟรมสำหรับโลว์สปีดและฟูลสปีด, 20% ของไมโครเฟรมสำหรับไฮสปีด	ไม่มี	90% ของเฟรมสำหรับโลว์สปีด/ฟูลสปีด, 80% ของไมโครเฟรมสำหรับไฮสปีด (ไอโซโครนัสและอินเตอร์รัพท์รวมกัน)	
การแก้ไขความผิดพลาดของข้อมูล	มี	มี	มี	ไม่มี
ชนิดของข้อมูล (เมสเสจ / สตรีม)	เมสเสจ	สตรีม	สตรีม	สตรีม
การรับประกันอัตราการส่ง	ไม่มี	ไม่มี	ไม่มี	มี
การรับประกันค่าเวลาสูงสุดระหว่างการส่งถ่ายข้อมูล	ไม่มี	ไม่มี	มี	มี

2.7 การส่งผ่านข้อมูลแบบ RS-232

2.7.1 มาตรฐานพอร์ตอนุกรมแบบ RS-232

มาตรฐานการเชื่อมต่อแบบอนุกรม RS-232 เป็นมาตรฐานอุตสาหกรรมที่ออกแบบมาเพื่อใช้ในการส่งข้อมูลอนุกรมแบบอะซิงโครนัส 2 ทิศทาง โดยมาตรฐาน RS-232 ในอดีตนั้นถูกออกแบบมาเพื่อการส่งผ่านข้อมูลจากคอมพิวเตอร์ไปยังโมเด็มเพียงอย่างเดียว เพื่อที่จะนำข้อมูลจากโมเด็มนี้สื่อสารผ่านสายโทรศัพท์ไปยังคอมพิวเตอร์อีกชุดซึ่งอยู่ห่างไกลกัน โดยคณะกรรมการที่เรียกว่า สมาคมอุตสาหกรรมอิเล็กทรอนิกส์ (Electronic Industries Association: EIA) ได้วางมาตรฐานที่มีชื่อเรียกกันว่า EIA RS-232 มาตรฐานนี้ในช่วงแรกจะใช้คอนเน็กเตอร์เป็นแบบ DB25 โดยกำหนดความยาวสูงสุดของสายสัญญาณไว้ที่ 50 ฟุต มีระดับสัญญาณตั้งแต่ -3 ถึง -12 Volt แสดงว่ามีข้อมูล (Mark) และ +3 ถึง +12 Volt แสดงว่าเป็นช่องว่าง (Space)

มาตรฐาน RS-232 ได้กำหนดรูปแบบของอุปกรณ์เชื่อมต่อข้อมูล (Data Terminal Equipment: DTE) กับวงจรข้อมูลปลายทาง (Data Circuit Terminating: DCE) ไว้ว่า อุปกรณ์

DTE จะเป็นอุปกรณ์ที่มีการประมวลผลในตัวเช่น Microcomputer หรือ Microcontroller ซึ่งมีความสามารถในการสร้างข้อมูลแบบอนุกรมได้ ส่วนอุปกรณ์ DCE จะทำหน้าที่เป็นเพียงตัวรับข้อมูลที่ส่งมาจาก DTE เท่านั้น โดยการรับส่งข้อมูลระหว่างอุปกรณ์ทั้งสองจะกระทำผ่านมาตรฐาน RS-232

คำจำกัดความของ DTE และ DCE ซึ่งแสดงในบรรทัดด้านล่างนี้ ได้คัดมาจากคำแปลศัพท์ในหนังสือ "Technical Aspect of Data Communication" ซึ่งเขียนโดย John McNamara (Digital Press, 1977) ดังนี้

DCE : อุปกรณ์ที่มีฟังก์ชั่นการทำงานต่างๆ ที่ทำให้เกิดการเชื่อมต่อยังดำรงต่อไป และยุติการเชื่อมต่อ นอกจากนี้ยังใช้เปลี่ยนลักษณะของสัญญาณและสร้างรหัสสัญญาณต่างๆ ที่จำเป็นต้องใช้ในการสื่อสารข้อมูลระหว่าง DTE และ Data Circuit โดย DCE อาจเป็นส่วนใดส่วนหนึ่งของ Computer หรือไม่ก็ได้

DTE : 1. เป็นอุปกรณ์ที่ประกอบไปด้วยตัวส่งข้อมูล (Data Source) หรือตัวรับข้อมูล (Data Sink) หรือเป็นทั้งตัวส่งและตัวรับข้อมูลก็ได้

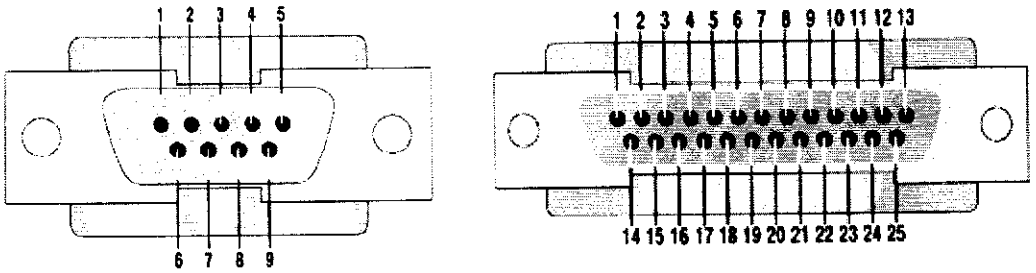
2. เป็นอุปกรณ์ที่ประกอบด้วย Function Unit ต่อไปนี้ Control Logic, Buffer Store และอุปกรณ์อินพุตหรือเอาท์พุตจำนวนหนึ่งตัวหรือมากกว่าก็ได้ หรือรวมเครื่องคอมพิวเตอร์เข้าไปด้วยก็ได้ DTE อาจจะรวมส่วน Error Control, Synchronization และความสามารถในการบ่งหรือระบุว่าต้องการเกี่ยวข้องกับอุปกรณ์ตัวใด (Station Identification Capability) เข้าไปด้วยก็ได้

ในความเป็นจริงแล้ว DTE มักจะแทนแหล่งกำเนิดข้อมูลแหล่งแรก และ/หรืออุปกรณ์ที่เป็นแหล่งข้อมูลแหล่งสุดท้าย เช่นเครื่องพิมพ์จะเป็น DTE เพราะเป็นอุปกรณ์ที่รับข้อมูลเป็นตัวสุดท้าย หรือ CRT, Keyboard เป็นทั้งตัวรับข้อมูลและตัวกำเนิดข้อมูล ส่วน DCE เป็นอุปกรณ์ที่ทำให้การสื่อสารข้อมูลระหว่างแหล่งกำเนิดกับตัวรับข้อมูลที่ปลายทางทำได้สะดวกขึ้น ตัวอย่างของ DCE ก็คือโมเด็ม

ข้อแตกต่างของอุปกรณ์ DTE และอุปกรณ์ DCE อย่างหนึ่งที่เห็นได้ชัดคือ คอนเน็กเตอร์ของ DTE จะเป็นตัวผู้ ส่วนคอนเน็กเตอร์ของ DCE จะเป็นตัวเมีย ซึ่งพอร์ตอนุกรมของคอมพิวเตอร์ที่ใช้กันอยู่ทั่วไปจะเป็นแบบ DTE ส่วนคอนเน็กเตอร์ที่อยู่ที่โมเด็มจะเป็นแบบ DCE

2.7.2 คอนเน็กเตอร์สำหรับพอร์ต RS-232 และการเชื่อมต่อ

มาตรฐานการเชื่อมต่อแบบ RS-232 จะใช้คอนเน็กเตอร์แบบ DB25 ตัวผู้หรือ DB9 ตัวผู้ ซึ่งคอนเน็กเตอร์แบบ DB25 จะมีขาต่อใช้งานเพียง 9 เส้นเช่นเดียวกับคอนเน็กเตอร์แบบ DB9 เนื่องจากขาอื่นๆ ที่เคยใช้งานในอดีต ปัจจุบันมีการใช้งานไม่มากนัก จึงถูกยกเลิกไป โดยแสดงรูปร่างและตำแหน่งขาในรูป 2.12



รูปที่ 2.12 รูปร่างและตำแหน่งขาของพอร์ต RS-232

ตารางที่ 2.12 ตำแหน่งของขาต่างๆ

คอนเน็กเตอร์ DB9	คอนเน็กเตอร์ DB25	ชื่อของสายสัญญาณ	ชนิดของสายสัญญาณ
1	8	Data Carrier Detect	Input
2	3	Received Data	Input
3	2	Transmitted Data	Output
4	20	Data Terminal Ready	Output
5	7	Signal Ground	-
6	6	Data Set Ready	Input
7	4	Request To Send	Output
8	5	Clear To Send	Input
9	22	Ring Indicator	Input

รายละเอียดหน้าที่การทำงานในแต่ละขาของพอร์ตอนุกรม RS-232 มีดังนี้

Data Carrier Detect : DCD หรืออาจเรียกว่า Carrier Detect: CD ขานี้จะแอกตีฟเมื่อมีการส่งสัญญาณพาห์จากอุปกรณ์สื่อสารข้อมูลเช่น โมเด็ม สำหรับการใช้งานปกติขานี้ไม่ได้ถูกใช้งานมากนัก

Receive Data : RD หรือ RxD ขานี้ใช้เพื่อรับสัญญาณอนุกรมเข้ามายังคอมพิวเตอร์ โดยนำข้อมูลที่อ่านได้เก็บไว้ในรีจิสเตอร์บัฟเฟอร์

Transmitted Data : TD หรือ TxD ใช้ส่งข้อมูลออกจากคอมพิวเตอร์ โดยนำข้อมูลที่เก็บอยู่ในบัฟเฟอร์สำหรับส่งข้อมูลส่งออกไป

Data Terminal Ready : DTR เป็นขาสัญญาณที่ส่งออกจากคอมพิวเตอร์เพื่อให้อุปกรณ์ปลายทางรับรู้ว่า ต้องการติดต่อด้วย โดยขา DTR นี้ต้องเชื่อมต่อกับ DSR ของอุปกรณ์ปลายทาง และขา DTR นี้ต้องเชื่อมต่อกับขา DSR ของคอมพิวเตอร์ ถ้าใช้การเชื่อมต่อเป็นแบบ Null Modem ซึ่งใช้สายในการเชื่อมต่อเพียง 3 เส้น จะต้องต่อขา DTR และ DSR ของตัวมันเองเข้าด้วยกันและต้องต่อกับขา DCD ด้วยในกรณีที่โปรแกรมสื่อสารที่ใช้มีการตรวจจับสัญญาณพาห์

Signal Ground : GND กราวด์ระบบ

Data Set Ready : DSR ขานี้จะใช้คู่กับขา DTR เพื่อตรวจสอบการเชื่อมต่อกันระหว่างคอมพิวเตอร์กับอุปกรณ์ปลายทาง ซึ่งขา DSR นี้จะเป็นขาสำหรับรับข้อมูลจากภายนอกซึ่งถูกส่งมาจากขา DTR

Request To Send : RTS เป็นขาสำหรับส่งสัญญาณร้องขอให้ทางอุปกรณ์ปลายทางส่งข้อมูลกลับมายังคอมพิวเตอร์ โดยขาที่รับสัญญาณ RTS ก็คือขา CTS ในกรณีที่ใช้การเชื่อมต่อแบบ Null Modem 3 สาย จะต้องเชื่อมต่อขา RTS และ CTS ของตัวมันเองเข้าด้วยกัน เพื่อจะให้การรับและส่งข้อมูลสามารถเกิดขึ้นได้ตลอดเวลา

Clear To Send : CTS ขานี้จะคอยรับสัญญาณจากขา RTS เมื่อรับสัญญาณได้ ข้อมูลที่ขา TxD จะถูกส่งออกไป ดังนั้นขานี้จึงถูกใช้เพื่อตรวจสอบอุปกรณ์ต่อพ่วงว่าพร้อมที่จะรับข้อมูลหรือไม่

Ring Indicator : RI ใช้แสดงสถานะสัญญาณเรียกจากสายโทรศัพท์ ปกติในการสื่อสารโดยทั่วไปสายนี้จะไม่ถูกใช้งาน จะใช้งานก็ต่อเมื่อมีการเชื่อมต่อกับโมเด็มและโปรแกรมมีการตรวจสอบสองสัญญาณนี้เท่านั้น

2.7.3 UART

UART ย่อมาจากคำว่า Universal Asynchronous Receiver Transmitter ซึ่งหมายถึงอุปกรณ์ที่ทำหน้าที่รับและส่งข้อมูลแบบ Asynchronous นั่นเอง สำหรับการสื่อสารอนุกรมบนคอมพิวเตอร์แล้ว UART ถือว่าเป็นหัวใจสำคัญของการสื่อสารอนุกรม

หน้าที่หลักของ UART คือทำหน้าที่แปลงข้อมูลที่อยู่ในรูปแบบขนานจากคอมพิวเตอร์ให้อยู่ในรูปแบบอนุกรมแบบ Asynchronous แล้วส่งออกไป และทำหน้าที่แปลงสัญญาณอนุกรมแบบ Asynchronous ที่ป้องเข้ามายัง UART ให้เป็นแบบขนานก่อนที่จะส่งเข้าสู่คอมพิวเตอร์ ซึ่งนอกจาก UART จะส่งข้อมูลไปยังคอมพิวเตอร์แล้ว ยังแจ้งข้อมูลอื่นๆ ให้คอมพิวเตอร์รับทราบด้วย เช่น อัตราเร็วในการรับส่งข้อมูล (บอดเรต), รูปแบบการส่งข้อมูล, ความผิดพลาดที่เกิดขึ้นระหว่างการถ่ายทอดข้อมูล (ผิดพลาดจากพาริตี้, เฟรมข้อมูล โอเวอร์รัน) เป็นต้น

ภายใน UART จะมีส่วนของวงจรสร้างบอดเรตแบบโปรแกรมได้ (Programmable Baudrate Generator) โดยการกำหนดค่าตัวหารให้กับสัญญาณนาฬิกาของ UART และ สามารถรับส่งข้อมูลได้ทั้งแบบ Half Duplex และ Full Duplex

2.7.3.1 การรับส่งข้อมูลแบบ Asynchronous

คือระบบการรับส่งข้อมูลที่แต่ละคำถูกส่งไปอย่างไม่มีกำหนดเวลาแน่นอน นั่นคือระยะเวลาระหว่างข้อมูลแต่ละคำที่ถูกส่งออกไปมีค่าไม่แน่นอน ดังนั้นสิ่งที่กำหนดเวลาในการส่งข้อมูลคือ ความพร้อมเพรียงของเครื่องส่งและเครื่องรับ

ในการส่งข้อมูลอนุกรมแบบอะซิงโครนัสนั้น โครงสร้างของข้อมูลที่จะส่งจะมีลักษณะเป็นบล็อกๆ ซึ่งแต่ละบล็อกประกอบด้วยบิทเริ่มต้น (Start Bit) ส่วนของข้อมูลและบิทสุดท้ายคือบิทสิ้นสุด (Stop Bit) โดยบิทเริ่มต้อนจะแสดงถึงการเริ่มต้อนของกลุ่มข้อมูล แล้วตามด้วยส่วนของกลุ่มข้อมูล และบางกรณีอาจมีการเพิ่มบิทพาริตี้เพื่อใช้ตรวจสอบความถูกต้องของข้อมูล และบิทสุดท้ายจะเป็นการบอกว่าข้อมูลในบล็อกๆ นี้หมดลงเพียงแค่นี้ ข้อมูลในบล็อกๆ หนึ่งจะต้องมีบิทที่แสดงให้รู้ว่าเป็นการเริ่มต้นของข้อมูลและสิ้นสุดข้อมูล

บิทเริ่มต้น :

ในโปรโตคอลของการส่งข้อมูลอนุกรมแบบอะซิงโครนัส กำหนดให้สถานะมาร์ค (Marking State) เป็นสัญญาณลอจิก 1 เมื่อทางด้านส่งจะทำการส่งข้อมูลก็จะต้องส่งบิทเริ่มต้นโดยแทนด้วยสถานะสเปส (Space State) หรือสัญญาณลอจิก 0 จำนวน 1 บิทไปก่อน ซึ่งจะทำให้ทางด้านรับดีเท็กซ์สถานะของสายส่งได้ว่าขณะนั้นสายส่งกำลังมีข้อมูลส่งมา

บิทข้อมูล :

หลังจากที่ด้านรับสามารถดีเท็กซ์สัญญาณบิทเริ่มต้นได้แล้ว ก็จะทำการเซ็ทสถานะของชิฟรีจิสเตอร์ ให้พร้อมที่จะรับบิทข้อมูลได้ โดยบิทข้อมูลจะมีจำนวนบิทเป็น 5, 6, 7 หรือ 8 บิท ขึ้นกับจำนวนคาร์แรกเตอร์ที่ใช้ดังแสดงตามตารางต่อไปนี้

ตารางที่ 3.13

จำนวนบิทข้อมูลใน 1 คาร์แรกเตอร์	จำนวนคาร์แรกเตอร์
5 บิต	32
6 บิต	64
7 บิต	128
8 บิต	256

นอกจากนี้รหัสต่างๆ ที่ใช้อาจจะแทนด้วยรหัสต่างๆ ที่ใช้อาจจะแทนด้วย 5 บิท ซึ่งเป็นมาตรฐานของรหัส Baudot โดยประกอบด้วยกลุ่มของคาร์แรกเตอร์ต่างๆ จำนวน 32 คาร์แรกเตอร์ และถ้าเป็นรหัสขนาน 7 บิท จะประกอบด้วยกลุ่มของคาร์แรกเตอร์ต่างๆ จำนวน 128 ตัว ซึ่งเป็นมารตรฐานของรหัส ASCII และใช้กันแพร่หลายมาก นอกจากนี้ก็ยังมีรหัสขนาด 8 บิท หรือมาตรฐานของรหัส EBCDIC โดยมาตรฐานนี้ประกอบด้วยกลุ่มของคาร์แรกเตอร์ 256 คาร์แรกเตอร์ เป็นต้น

บิทพาริตี้ :

บิทนี้จะทำหน้าที่ในการบอกให้ด้านรับทราบว่าข้อมูลที่ส่งมาแต่ละไบท์นั้น มีจำนวนบิทที่เป็น 1 อยู่เป็นจำนวนคี่หรือจำนวนคู่ บิทพาริตี้นี้จะถูกส่งออกมาพร้อมกับบิทข้อมูล ซึ่งบิทนี้จะเป็น

1 หรือ 0 นั้นขึ้นกับข้อมูลที่ส่งมาว่ามีจำนวนบิทที่เป็น 1 เป็นจำนวนคู่หรือคี่ และยังขึ้นกับอุปกรณ์รับส่งข้อมูลด้วยว่าถูกออกแบบไว้ให้รับส่งบิทพาริตี้ในลักษณะพาริตี้คู่หรือคี่อีกด้วย

สิ่งสำคัญอีกสิ่งหนึ่งก็คือ ถ้าอุปกรณ์ส่งข้อมูลทำการส่งในลักษณะพาริตี้คู่ หรือคี่ก็ตามส่วนรับข้อมูลก็จะต้องทำการรับในลักษณะพาริตี้เดียวกับอุปกรณ์ส่งข้อมูลด้วย เช่น ในกรณีที่อุปกร์ส่งข้อมูลทำการส่งข้อมูลในลักษณะพาริตี้เดียวกับอุปกรณ์ส่งข้อมูลด้วย เช่น ในกรณีที่อุปกรณ์ส่งข้อมูลทำการส่งข้อมูลในลักษณะพาริตี้คู่ อุปกรณ์รับข้อมูลก็จะต้องทากรรับข้อมูลในลักษณะของพาริตี้คู่ด้วย เป็นต้น

บิทสิ้นสุดข้อมูล (Stop Bit) :

บิทสุดท้ายที่เพิ่มเข้าไปนี้ จะใช้ในการจรวจสอบจุดสิ้นสุดของข้อมูล บิทนี้จะถูกเพิ่มเข้าไปหลักพาริตี้บิท ถ้าอุปกรณ์รับข้อมูลตรวจไม่พบบิทนี้ ก็แสดงว่าข้อมูลที่รับเข้านั้นเกิดข้อผิดพลาดขึ้น สำหรับบิทสิ้นสุดข้อมูลนี้อาจจะมีจำนวนบิทเป็น 1, 1.5 หรือ 2 บิทก็ได้ ซึ่งสรุปได้ว่าข้อมูลที่ส่งออกมาในแต่ละไบท์นั้นไม่ใช่มีแต่ข้อมูล 8 บิทเท่านั้น แต่อาจมีได้ถึง 12 บิท หรือมากกว่าก็ได้

2.7.3.2 การส่งข้อมูลแบบ Full Duplex

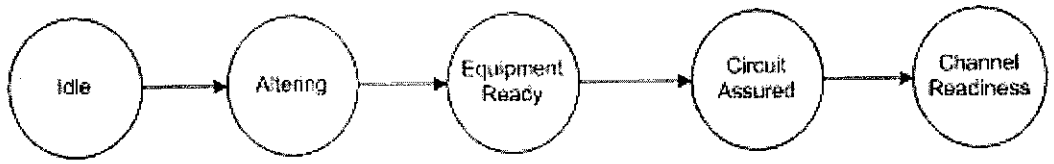
สัญญาณต่างๆ ที่ใช้ในการส่งข้อมูลแบบ Full Duplex มีดังนี้

Signal Ground
Transmitted Data
Received Data
Request to Send
Clear to Send
Data Set Ready
Received Line Signal Detector

ถ้า DCE ที่ใช้ในการสื่อสารข้อมูลผ่านทางเครือข่ายโทรศัพท์ คือ MODEM เราจะต้องใช้สัญญาณต่อไปนี้เพิ่มจากสัญญาณต่างๆ ที่กล่าวไว้ข้างต้น

Data Terminal Ready
Ring Indicator

การเปลี่ยนแปลงจากสภาวะหนึ่งไปเป็นอีกสภาวะหนึ่งนั้นต้องอาศัย เหตุการณ์ (Event) และ การจัดลำดับ (Sequence) เพื่อให้การเปลี่ยนสภาวะเป็นไปอย่างถูกต้อง EIA ได้รวบรวมเหตุการณ์ (Event) และการเปลี่ยนแปลงต่างๆ (Transition) เข้าเป็น กลุ่ม (Phase) ต่างๆ ระหว่างสภาวะนิ่ง (Idle) และสภาวะที่สามารถแลกเปลี่ยนข้อมูลได้ (Data Exchange) ซึ่งอุปกรณ์เช่น DTE และ DCE ต้องผ่านเหตุการณ์และการเปลี่ยนแปลงเหล่านี้จึงสามารถแลกเปลี่ยนข้อมูลได้ กลุ่มของเหตุการณ์และการเปลี่ยนแปลงได้แสดงไว้อย่างย่อๆ ดังรูปที่ 2.13



รูปที่ 2.13 การเปลี่ยนแปลงจากสภาวะหนึ่งไปเป็นอีกสภาวะหนึ่งโดยอาศัย เหตุการณ์ (Event) และการจัดลำดับ (Sequence)

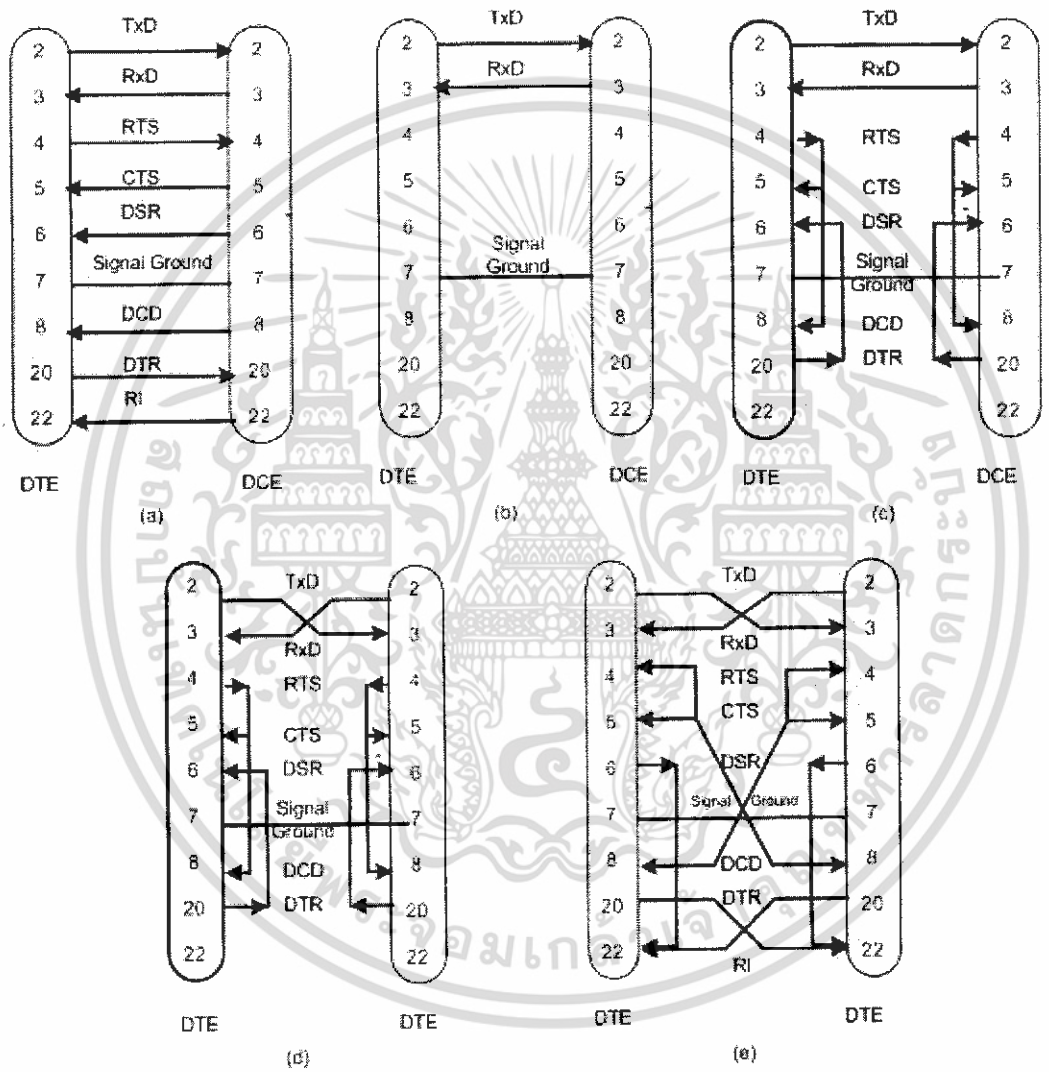
กลุ่ม (Phase) หลักที่เกิดขึ้นมีดังนี้

- การเตรียมพร้อม (Altering)
- อุปกรณ์พร้อม (Equipment Readiness)
- การรับรองความพร้อมของเซอร์กิต (Circuit Assurance)
- ช่องทางการสื่อสารที่ใช้ในการส่งข้อมูลทั้งหมดพร้อม (Channel Readiness)

ใน Altering Phase สถานีเรียก (Call-Originating Station) จะต่อโทรศัพท์ตามหมายเลขของสถานีปลายทาง (Remote Call-Answering Station) ซึ่งจะตอบรับต่อการเรียกที่มีเข้ามา เมื่อกริ่งโทรศัพท์ที่ด้านปลายทาง (Remote) ดังขึ้น สัญญาณ Ring Indicator ที่ DCE ปลายทาง (Answering DCE) จะเปลี่ยนสภาวะจาก OFF เป็น ON เมื่อ DCE ตอบรับต่อการเรียกจะเป็นการเสร็จสิ้นการทำงานของ Altering Phase ในเวลาเดียวกันเหตุการณ์ที่จะนำระบบเข้าสู่ Equipment Readiness Phase ก็จะเกิดขึ้น สัญญาณตามมาตรฐาน RS-232C ที่เกี่ยวกับการทำงานของ Phase นี้คือ Data Set Ready และ Data Terminal Ready เมื่อเหตุการณ์ต่างๆ ที่ใช้ในการ ON สัญญาณทั้งสองนี้เกิดขึ้นเรียบร้อยแล้วจะเป็นการสิ้นสุดการทำงานของ Equipment Readiness Phase ซึ่งจะเข้าสู่การทำงานของ Circuit Assurance Phase ต่อไป การรับรองความพร้อมของ Phase นี้ประกอบด้วยเหตุการณ์ต่างๆ ที่ใช้ในการ ON สัญญาณ Receive Line Signal Detector ที่สถานีทั้งสอง เมื่อสัญญาณนี้มีสภาวะเป็น ON ทั้งสองด้าน การทำงานของ Circuit Assurance Phase ก็จะสิ้นสุดลง จะเข้าสู่การทำงานของ Channel Readiness Phase ต่อไป ใน Channel Readiness Phase จะมีการทำ Handshaking ของสัญญาณ Request to Send และ Clear to Send เพื่อนำไปสู่ Target State ในที่นี้สภาวะเป้าหมายที่ต้องการคือ สามารถแลกเปลี่ยนข้อมูลระหว่างเครื่องไมโครคอมพิวเตอร์กับเครื่องปลายทางได้ ในระหว่างการเกิดเหตุการณ์ต่างๆ ในแต่ละ Phase อาจจะมีบางเหตุการณ์ที่เกิดผิดพลาดขึ้นหรืออุปกรณ์บางตัวเกิดไม่พร้อมที่จะทำการสื่อสารข้อมูลขึ้นมาทำให้การสื่อสารข้อมูลไม่อาจเกิดขึ้นได้ ซึ่งจะทำให้ระบบกลับไปเริ่มต้นที่สภาวะ Idle ใหม่

2.8 ลักษณะการต่อ RS-232 ที่ไม่ใช่แบบมาตรฐาน

ในรูปที่ 2.14 แสดงไดอะแกรมรูปภาพของการต่อ RS-232 แบบต่างๆ ที่ไม่ใช่แบบมาตรฐาน ในระบบไมโครคอมพิวเตอร์มักจะใช้การต่อเคเบิล RS-232 ตามรูป 2.14 รูปที่ 2.14a แสดงการต่อสายเคเบิลแบบ Full-Duplex ซึ่งถูกต้องตามมาตรฐานทุกอย่าง สำหรับรูปที่ 2.14b ถึง 2.14e แสดงการต่อที่ต่างออกไปจากมาตรฐาน โดยไดอะแกรมแต่ละรูปแทนคอนเน็คเตอร์ด้วยกล่องสี่เหลี่ยมผืนผ้าขอบมนและได้กำหนดไว้ด้วยว่าคอนเน็คเตอร์ตัวใดเป็น DTE หรือ DCE ลูกศรที่ปลายเส้นตรงแต่ละเส้นแสดงทิศทางการส่งสัญญาณ



รูปที่ 2.14 ลักษณะการต่อ RS-232 ที่ไม่ใช่แบบมาตรฐาน

Three-Wire Economy Model

จากไดอะแกรมรูปภาพในรูปที่ 2.14b สายเคเบิลที่ใช้นั้นใช้เพียงสามเส้น คือใช้ที่ขา 2, 3 และ 7 การต่อสายเคเบิลแบบนี้ส่วนมากมักใช้ในการอินเทอร์เฟสกับระบบไมโครคอมพิวเตอร์ การต่อสายเคเบิลแบบนี้ใช้กับการส่งข้อมูลแบบ Full Duplex ซึ่งเป็นแบบที่ใช้ขาน้อยที่สุด คือใช้เพียงขา Transmitted Data, Received Data และ Signal Ground ข้อผิดพลาดที่เกิดขึ้นบ่อยที่สุดในการใช้งานสายเคเบิลแบบนี้คืออุปกรณ์บางตัวของระบบไมโครคอมพิวเตอร์ใช้ Request to Send และ Clear to Send ด้วย ดังนั้นอุปกรณ์เหล่านี้จะส่งข้อมูลได้ก็ต่อเมื่อได้รับสัญญาณ Clear to Send ก่อน อุปกรณ์เหล่านี้ได้แก่อุปกรณ์จำพวก USART (Universal Synchronous /Asynchronous Receiver/Transmitter)

Three-Wire with Luxury Loop-Back

จากไดอะแกรมรูปภาพในรูปที่ 2.14c จะเห็นว่าการต่อสายแบบนี้เป็นการแก้ปัญหาต่างๆ ที่กล่าวไว้ในหัวข้อก่อน สำหรับสาเหตุของการต่อ Data Terminal Ready เข้ากับ Data Set Ready นั้นเป็นการทำให้ระบบเข้าสู่ Equipment Readiness Phase ทันทีที่ DTE ป้อนสัญญาณ Data Terminal Ready ปกติสัญญาณนี้จะ ON เมื่อเราเปิดสวิตช์จ่ายไฟให้กับ DTE เมื่อ DTE ป้อนสัญญาณ Request to Send ระบบจะเข้าสู่ Circuit Assurance Phase ในทันที เนื่องจาก Request to Send ถูกต่อกับเข้ากับ Received Line Signal Detector (ปกติ Request to Send จะ ON เมื่อเราจ่ายไฟให้แก่ DTE) หรือจะกล่าวได้อีกทางหนึ่งก็คือเหตุการณ์ Request to Send ที่เกิดขึ้นจะเป็นตัว Trigger ให้เกิดเหตุการณ์ Data Carrier Detect นอกจากนี้ขา Request to Send ยังถูกต่อกับขา Clear to Send ดังนั้นเมื่อ Request to Send มีสภาวะ ON จะทำให้เกิด Channel Readiness Phase ขึ้นทันที ดังนั้นถ้า Data Terminal Ready และ Request to Send ยังอยู่ในสภาวะ ON ก็จะเข้าสู่สภาวะ Data-Exchange State ได้ทันที

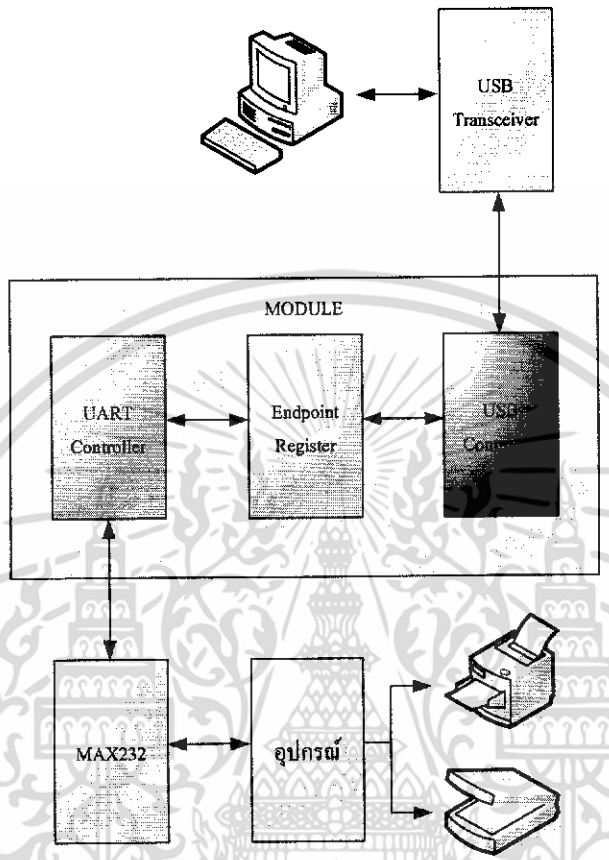
การต่อสายแบบนี้ได้ตัดการตรวจสอบเหตุการณ์บางอย่างทิ้งไปเพื่อลดความซับซ้อนของระบบ เช่น การต่อขา Data Terminal Ready เข้ากับ Data Set Ready โดยปกติ Data Set Ready จะ ON เมื่อเกิด Connect Sequence ขึ้นแล้วและ Data Terminal Ready มีสภาวะ ON แต่การต่อสายสองเส้นนี้เข้าด้วยกัน Data Set Ready จะ ON ตามสัญญาณ Data Terminal Ready เพียงอย่างเดียว จึงเป็นการตัดการตรวจสอบ Connect Sequence ทิ้งไป สำหรับการต่อขา Request to Send เข้ากับขา Data Carrier Detect จะตัดการตรวจสอบสัญญาณ Carrier ที่ต้องส่งมาจาก DCE ทางด้านที่เราจะติดต่อด้วยออกไป การต่อขา Request to Send กับ Clear to Send จะตัดการทำ Hand Shake ระหว่าง Request to Send กับ Clear to Send ซึ่งอาจก่อให้เกิด Overrun Error ขึ้น

การใช้ Null Modem ร่วมกับการต่อสายแบบ Luxury Loop-Back หรือแบบ Double Cross

จากไดอะแกรมรูปภาพในรูปที่ 2.14d และ 2.14e มีการไขว้สายขา Transmitted Data และ Received Data ในระบบไมโครคอมพิวเตอร์ส่วนใหญ่จะต้องไขว้สายสองเส้นนี้ เนื่องจากว่าอุปกรณ์ส่วนใหญ่ที่ใช้เป็นพอร์ทของไมโครคอมพิวเตอร์มักจะเป็น DTE มีเพียงส่วนน้อยมากที่เป็น DCE โดยอาศัยการไว้สายสองเส้นนี้ทำให้สามารถส่งข้อมูลระหว่าง DTE 2 ตัวได้อย่างถูกต้อง โดยไม่ต้องเพิ่มโมเด็มหรือ DCE เข้าไปอีก เทคนิคของการใช้ Null Modem แบบนี้เรียกว่า Crossover Technique ซึ่งแบ่งออกเป็น 2 ประเภทคือ Luxury Loop-Band และ Double-Cross จากรูปที่ 2.14d เป็นเป็นการใช้ Null Modem พร้อมทั้งทำ Loop-Back สัญญาณควบคุม จากรูปที่ 2.14e แสดงเทคนิคการไขว้สายแบบ Double-Cross

บทที่ 3
การออกแบบโมดูล

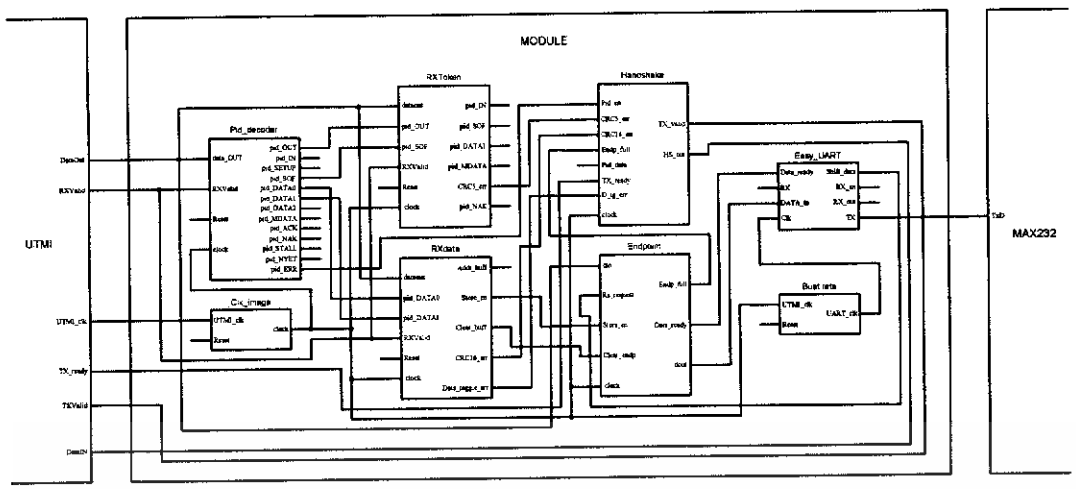
3.1 โมดูลที่ออกแบบ



รูปที่ 3.1 Block Diagram การทำงานของโครงงาน

จากรูปที่ 3.1 อธิบายได้ดังนี้ เริ่มจาก PC ซึ่งเป็น Port USB จะรับส่งข้อมูลกับ USB Transceiver โดยที่โมดูลจะเป็นตัวกลางในการจัดการกับโปรโตคอลทั้งโปรโตคอลที่เป็นแบบ USB ซึ่งออกมาจากตัว USB Transceiver และโปรโตคอลที่เป็นแบบ RS-232 ซึ่ง FPGA จะรับส่งข้อมูลที่เป็นโปรโตคอลในแบบ RS-232 กับ IC MAX232 และ IC MAX232 จะเป็นตัวจัดการแรงดันที่จะส่งไปให้กับอุปกรณ์ที่มีค่าแรงดันเป็นค่ามาตรฐาน

โดยภายในโมดูลจะทำการออกแบบในส่วนของ UART Controller, USB Controller และ Endpoint Register ซึ่งแสดงวงจรรวมไว้ดังรูปที่ 3.2(สามารถดูรูปขนาดขยายได้ที่บทที่ 4 รูปที่ 4.2) Endpoint Register จะเป็นจุดพักข้อมูลจริงซึ่งถูกแยก ออกจากโปรโตคอลของทั้งสองแบบแล้ว

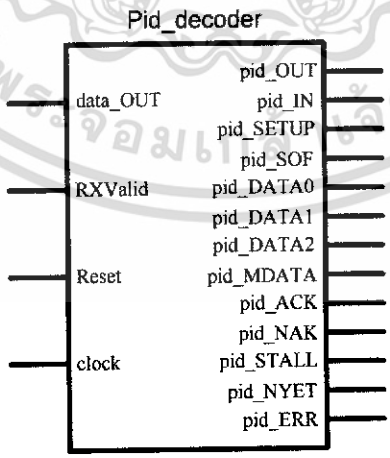


รูปที่ 3.2 บล็อกการออกแบบรวมทั้งหมดของโมดูล

สำหรับการทำงานของวงจรทั้งหมดนั้นจะเริ่มจากโฮสจะส่งข้อมูลโทเค็นแพ็กเก็ตมาก่อน ในส่วนของบล็อก pid_decoder ที่ทำหน้าที่ในการวิเคราะห์ว่าค่า PID ที่ทางโฮสส่งมานั้นมีค่าเป็นเท่าไร หลังจากที่วิเคราะห์ค่า PID เสร็จแล้ว บล็อก Rx_token จะทำหน้าที่ในการวิเคราะห์ค่าของข้อมูลที่อยู่ในโทเค็นแพ็กเก็ตและทำการตรวจสอบค่าความผิดพลาดในโทเค็นแพ็กเก็ตด้วย หลังจากนั้นโฮสจะทำการส่งดาต้าแพ็กเก็ต บล็อก pid_decoder จะทำหน้าที่ในการวิเคราะห์ค่า PID เหมือนเดิมหลังจากนั้นบล็อก rxdata จะทำหน้าที่ในการแยกส่วนของดาต้าและทำการตรวจสอบค่าความผิดพลาดของข้อมูลที่ได้รับมา แล้วส่งต่อไปยัง Endpoint เพื่อทำการเก็บข้อมูลจริงที่ได้เพื่อรอให้ส่วนของบล็อก easy_UART มาเรียกข้อมูลไปใช้ต่อ

3.2 การออกแบบในส่วนของ USB Controller

3.2.1 pid_decoder

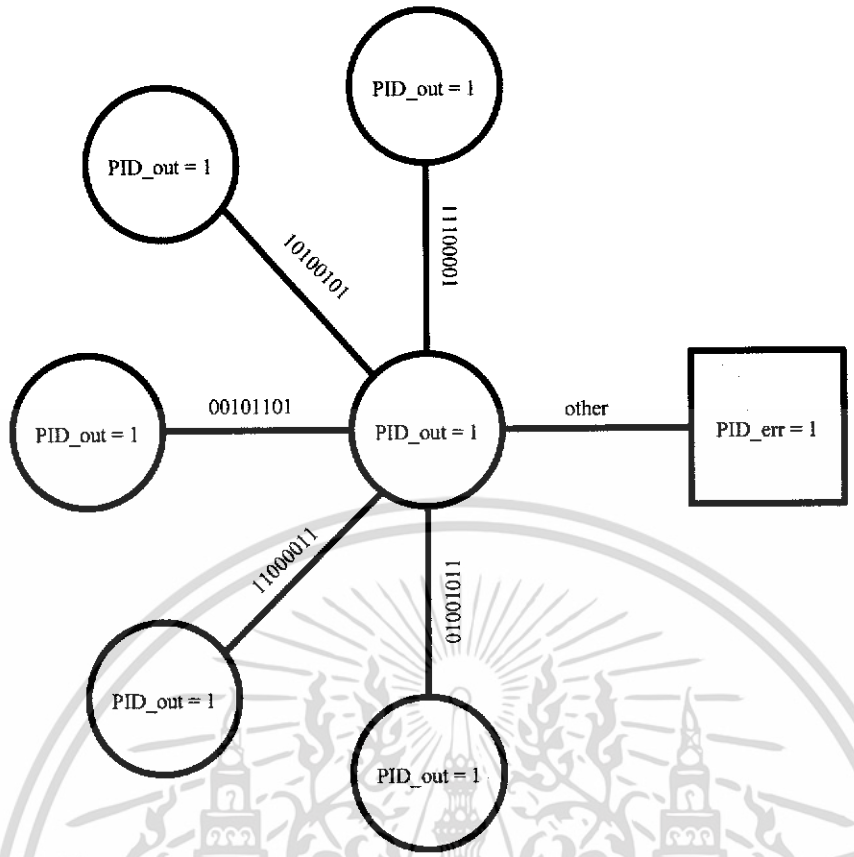


รูปที่ 3.3 บล็อกการออกแบบในส่วนของ pid_decoder

โดยปกติแล้ว token packet และ data packet จะต้องมี PID นำหน้ามาก่อนเสมอ ดังนั้น จึงต้องแยกให้ออกว่า PID ที่นำหน้าข้อมูลนั้นเป็น PID ชนิดใด ดังนั้น pid_decoder จึงมีหน้าที่ในการพิจารณาว่าค่าของ PID ที่ได้รับ ซึ่งแต่ละขามีหน้าที่ดังตารางที่ 3.1

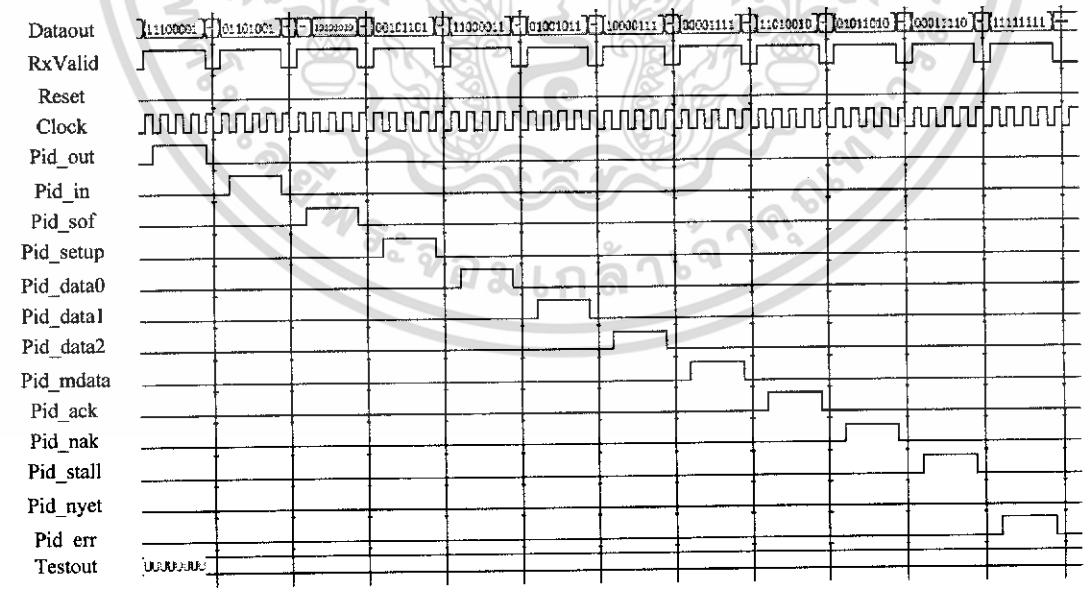
ตารางที่ 3.1รายละเอียดการทำงานของขาต่างๆ ภายในบล็อก pid_decoder

Pin	**Input/Output**	**Description**
DataOut	Input	รับข้อมูลจาก UTMI
RXValid	Input	แสดงสถานะว่าขณะนั้นมีข้อมูลเข้ามาภายในบล็อกหรือไม่ ถ้ามี RXValid = '1' ถ้าไม่มี RXValid = '0'
Reset	Input	ใช้ในการ reset โมดูลเมื่อมีคำสั่ง reset จากภายนอก
Clock	Input	สัญญาณนาฬิกาที่ได้รับจาก clk_image
pid_OUT	Output	ใช้ในการบอกสถานะว่า PID ที่ได้รับมานั้นเป็น PID อะไร ยกตัวอย่างเช่นถ้าได้รับ Token out ขาของ pid_OUT จะเท่ากับ 1
pid_IN	Output	
pid_SOF	Output	
pid_SETUP	Output	
pid_DATA0	Output	
pid_DATA1	Output	
pid_DATA2	Output	
pid_MDATA	Output	
pid_ACK	Output	
pid_NAK	Output	
pid_STALL	Output	
pid_NYET	Output	
pid_Err	Output	



รูปที่ 3.4 Flow chart แสดงการทำงานของ PID_decoder

การจำลองการทำงาน



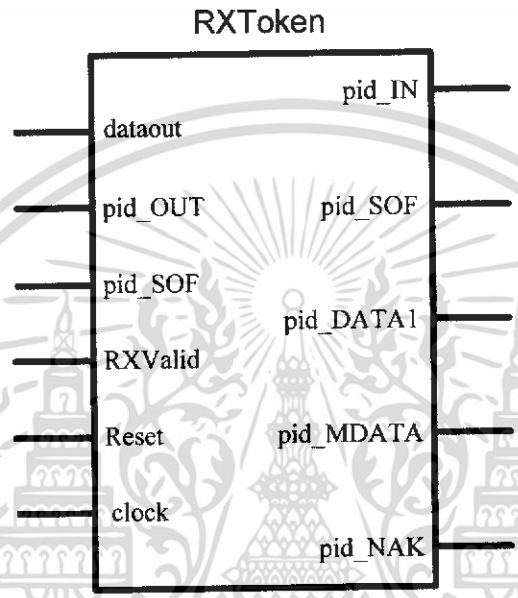
รูปที่ 3.5 รูปคลื่นที่ได้จากการจำลองการทำงานของบล็อก pid_decoder

คำอธิบายการทำงาน

จากกราฟจะเห็นว่าเมื่อข้อมูลที่เข้ามาเป็นค่าของ PID ขาสถานะต่างๆ ของ PID ก็จะขึ้นตามค่าของ PID ที่ได้รับเข้ามาทำให้เราสามารถทราบได้ว่า PID ที่เข้ามานั้นเป็นค่าของ PID อะไร

3.2.2 rxtoken

มีหน้าที่ในการแยกข้อมูลและตรวจสอบความผิดพลาดของข้อมูลที่ส่งมาในรูปของ Token Packet โดยใช้ CRC5 ในการตรวจสอบความถูกต้องของข้อมูล

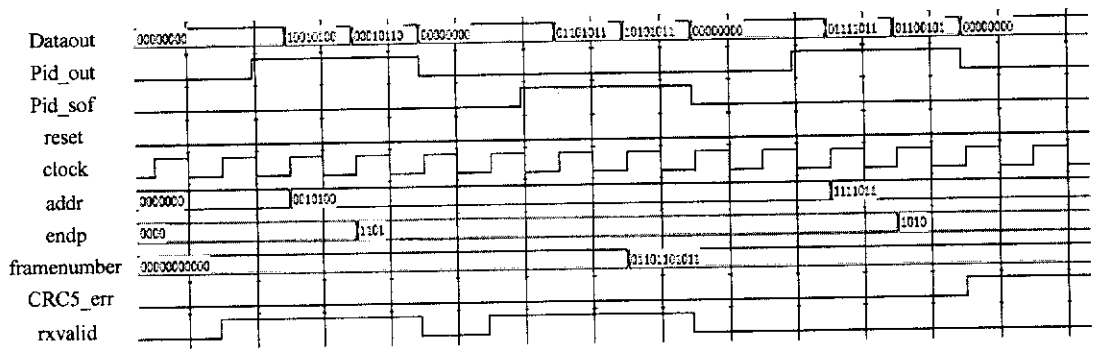


รูปที่ 3.6 บล็อกการออกแบบในส่วนของ rxtoken

ตารางที่ 3.2 รายละเอียดการทำงานของขาต่างๆ ภายในบล็อก rxtoken

Pin	Input/Output	Description
dataout	Input	ต่อกับ Pin DataOUT ของ IC UTMI
pid_OUT	Input	แจ้งสถานะว่ามี PID OUT เข้ามา
pid_SOF	Input	แจ้งสถานะว่ามี PID SOF เข้ามา
Clock	Input	Input สัญญาณนาฬิกาเข้า Module
reset	Input	ใช้ Reset Module ให้อยู่ในสภาวะเริ่มต้น
rxvalid	Input	รับ Input RXvalid จาก UTMI
Addr	Output	แจ้งตำแหน่ง Address ที่มากับ Token Packet
Endp	Output	แจ้งตำแหน่ง Endpoint ที่มากับ Token Packet
FrameNumber	Output	แจ้งตำแหน่ง Frame Number ที่มากับ Token Packet
CRC5_err	Output	แจ้งสถานะ Error เมื่อ CRC5 ของข้อมูลไม่ตรงกับที่รับมา

การจำลองการทำงาน

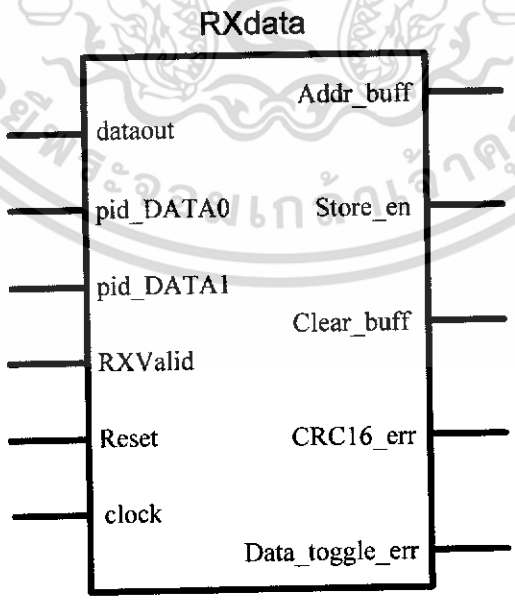


รูปที่ 3.7 รูปคลื่นที่ได้จากการจำลองการทำงานของบล็อก rxtoken

คำอธิบายการทำงาน

เริ่มจากรับข้อมูล Input จากขา DataOUT ของ UTMI ซึ่ง Module จะตรวจสอบก่อนว่า PID ที่เข้ามานั้นเป็น PID ชนิดใด หากค่าของ PID ไม่ตรงกับข้อกำหนดของ USB จะถือว่า PID ชุดนั้นเกิดความผิดพลาด และจะส่งสัญญาณ PID_err ไปยังส่วน HS(Handshake) เพื่อให้ตอบสนองกลับไปยัง Host ใน Packet Handshake ถ้าหาก PID ที่รับเข้ามานั้นถูกต้อง ก็จะทำการรับข้อมูลที่มาถัดจาก PID แต่ในระหว่างที่รับข้อมูลนั้น จะทำการตรวจ CRC5 จากข้อมูลที่รับเข้ามาด้วยโดยตรวจตามสัญญาณนาฬิกา ถ้าหาก CRC5 ที่รับเข้ามาจาก Token Packet ไม่ตรงกับ CRC5 ที่คำนวณได้ ก็จะตอบสถานะ CRC5_err ไปยังส่วน HS เพื่อให้ HS ตอบกลับไปยัง Host ใน Packet Handshake

3.2.3 rxdata



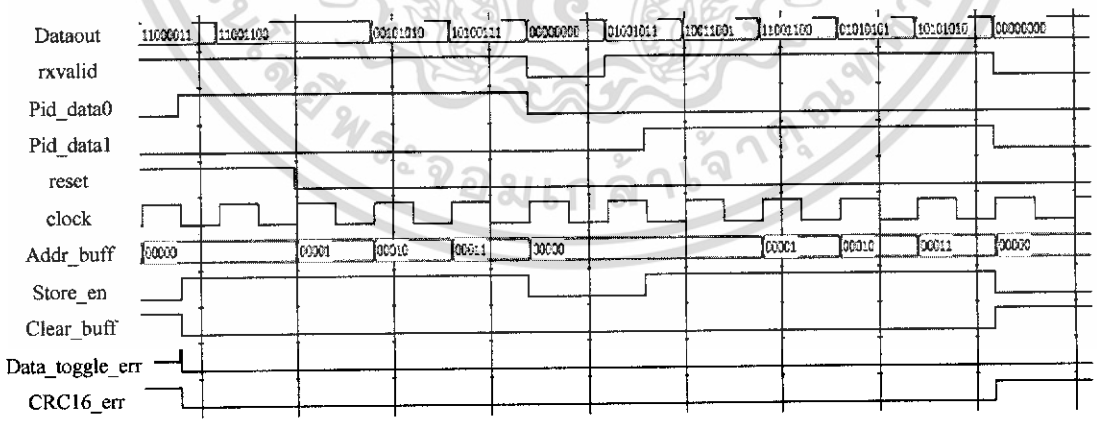
รูปที่ 3.8 บล็อกการออกแบบในส่วนของ rxdata

ในส่วนของบล็อกนี้มีหน้าที่ในการสร้างสัญญาณ Enable ให้บล็อกส่วนของการเก็บข้อมูล (Endpoint) ทำการตรวจสอบความถูกต้องของข้อมูล (ใช้ Data toggle และ CRC16)

ตารางที่ 3.3 รายละเอียดการทำงานของขาต่างๆ ภายในบล็อก rxdata

Pin	Input/Output	Description
Dataout	Input	ต่อกับ Pin DataOUT ของ IC UTMI
Rxvalid	Input	รับ Input RXvalid จาก UTMI
pid_data0	Input	Input PID Data0 จาก PID Decoder
Pid_data1	Input	Input PID Data1 จาก PID Decoder
Reset	Input	ใช้ Reset Module ให้อยู่ในสภาวะเริ่มต้น
Clock	Input	Input สัญญาณนาฬิกาเข้า Module
Addr_buff	Output	แจ้งตำแหน่งที่จะทำการเก็บข้อมูลใน Endpoint
Store_en	Output	ใช้สั่งให้ Endpoint ทำการเก็บข้อมูล
Clear_buff	Output	สั่งให้ทำการ Clear Endpoint เมื่อพบว่าข้อมูลที่รับเข้ามานั้นผิด
CRC16_err	Output	แจ้งสถานะ Error เมื่อ CRC16 ของข้อมูลไม่ตรงกับที่รับมา
Data_toggle_err	Output	แจ้ง Data Toggle ผิดพลาด

การจำลองการทำงาน



รูปที่ 3.9 รูปคลื่นที่ได้จากการจำลองการทำงานของบล็อก rxdata

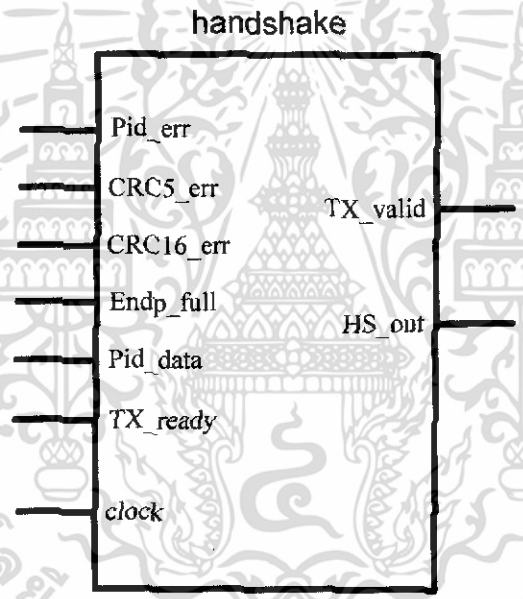
คำอธิบายการทำงาน

RXData เป็นทั้งตัวควบคุมการเก็บข้อมูลไปยัง Endpoint ตรวจสอบความถูกต้องของข้อมูล โดยสัญญาณที่ควบคุมการเก็บข้อมูลคือสัญญาณ Store_En ซึ่ง Store_EN จะเป็น 1 เมื่อ

หมดข้อมูลที่เป็น PID และจะเป็น 0 เมื่อหมดข้อมูลที่เป็น CRC16 ชุดสุดท้าย และยังทำการตรวจสอบข้อมูลที่รับเข้ามาแต่ละครั้งด้วยโดยการคำนวณ CRC16 ภายใน Module ไปเรื่อยๆ จนกระทั่งถึงข้อมูลตัวสุดท้ายที่เป็น CRC16 ถ้าหากว่า CRC16 ที่คำนวณได้นั้นไม่เป็น 0 ทั้งหมด แสดงว่าข้อมูลที่รับเข้ามานั้นผิดพลาด ก็จะทำการ Enable สัญญาณ Clear_Buff เพื่อให้ Endpoint การลบข้อมูลที่รับเข้ามา และ Enable สัญญาณ CRC16_err เพื่อให้ HS ทำการตอบสถานการณ์รับข้อมูลผิดพลาด (NAK) กลับไปยัง Host

3.2.4 hs (Handshake)

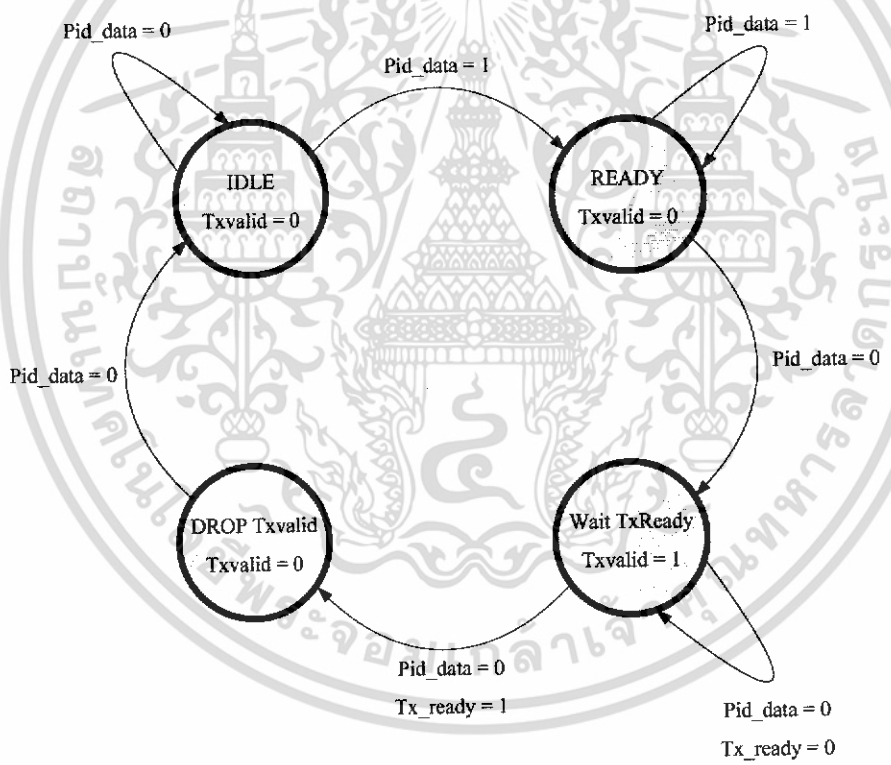
มีหน้าที่ตอบสถานการณ์รับข้อมูลไปยัง Host ว่ามีความผิดพลาดในการรับข้อมูลหรือไม่ โดยการนำสัญญาณตอบสนองจาก Module ต่างๆ มาทำการวิเคราะห์โดยมีรูปแบบการตอบรับ (สำหรับ Full Speed) อยู่ 3 แบบ คือ ACK, NAK, STALL ถ้าเป็น ACK แสดงว่าข้อมูลที่รับเข้ามานั้นสมบูรณ์ ไม่มีข้อผิดพลาดใด ๆ เกิดขึ้น



รูปที่ 3.10 บล็อกการออกแบบในส่วนของ hs

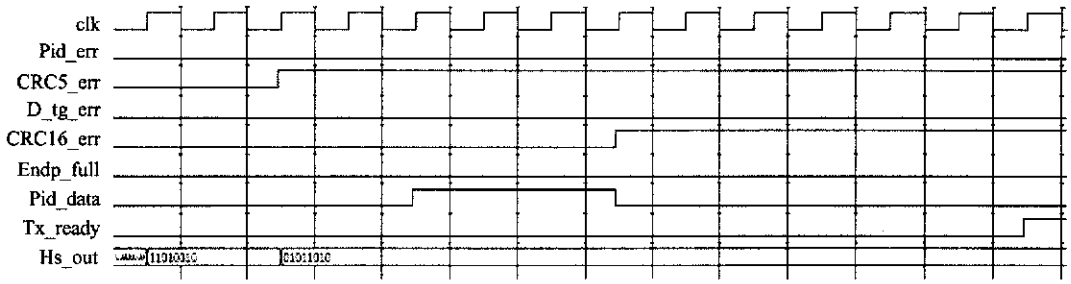
ตารางที่ 3.4 รายละเอียดการทำงานของขาต่างๆ ภายในบล็อก hs

Pin	Input/Output	Description
clk	Input	Input สัญญาณนาฬิกาเข้า Module
PID_err	Input	Input รับค่า PID Error จาก PID Decode Module
Crc5_err	Input	Input รับค่า CRC5 Error จาก RXToken Module
Crc16_err	Input	Input รับค่า CRC16 Error จาก RXToken Module
Endp_full	Input	Input รับค่า Endpoint Full เมื่อ
PID_data	Input	รับผลการ OR PID Data0 กับ PID Data1 เมื่อ
Tx_ready	Input	รับ Input TX Ready จาก UTMI
Tx_valid	Output	รับ Input TX valid จาก UTMI
Hs_out	Output	สัญญาณทดสอบ รวม



รูปที่ 3.11 flow chart แสดงการทำงานของบล็อก Handshake

การจำลองการทำงาน

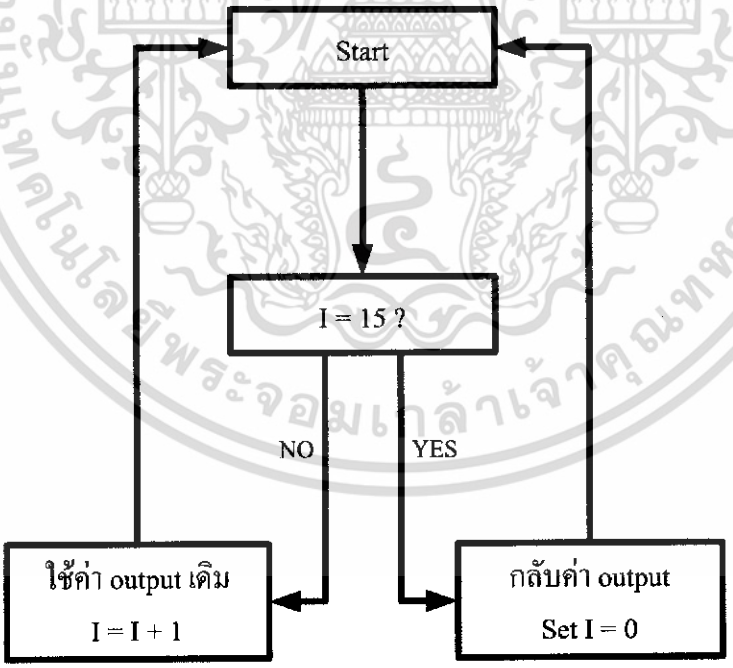


รูปที่ 3.12 รูปคลื่นที่ได้จากการจำลองการทำงานของบล็อก hs

3.2.5 clk_image

ใช้ในการสร้างสัญญาณนาฬิกาให้กับ Module ที่เกี่ยวข้องกับ USB ทั้งหมด

รูปที่ 3.13 บล็อกการออกแบบในส่วนของ clk_image



รูปที่ 3.14 flow chart แสดงการทำงานของบล็อก Clk_image

ตารางที่ 3.5 รายละเอียดการทำงานของขาต่างๆ ภายในบล็อก clk_image

Pin	Input/Output	Description
UTMI_clk	Input	สัญญาณ Input ความถี่จาก UTMI
Reset	Input	ใช้ Reset Module ให้อยู่ในสภาวะเริ่มต้น
clock	Output	Input สัญญาณนาฬิกาเข้า Module

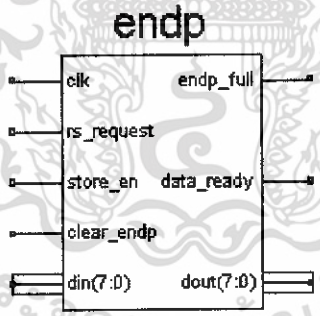
การจำลองการทำงาน

รูปที่ 3.15 รูปคลื่นที่ได้จากการจำลองการทำงานของบล็อก clk_image

คำอธิบายการทำงาน

จากกราฟ ความถี่ Clk_Out เกิดจากสัญญาณ Clk_In ซึ่งรับเข้ามาจาก IC UTMI โดย Module ทำงานในลักษณะ Counter คือจะนับขอบขาขึ้นของ Clk_In จนถึงค่าหนึ่งแล้วจึงเปลี่ยนจากสภาวะ High เป็น Low

3.3 การออกแบบในส่วนของ Endpoint (endp)



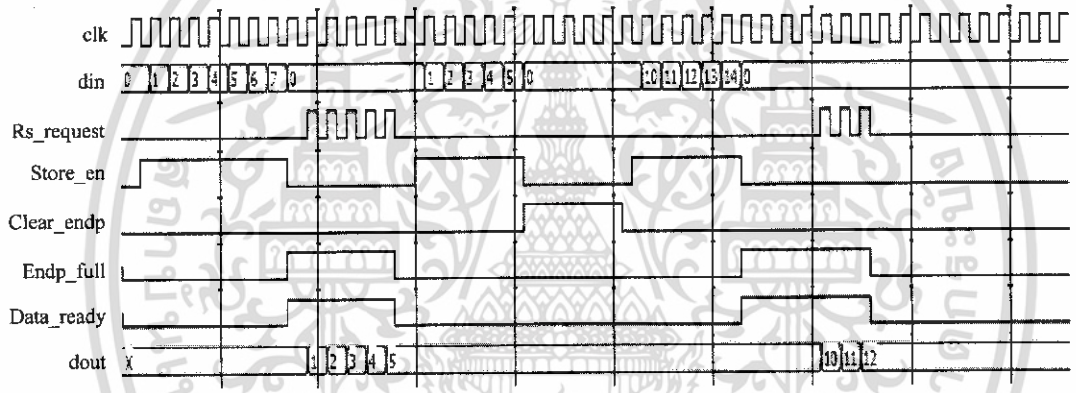
รูปที่ 3.16 บล็อกการออกแบบในส่วนของ Endpoint

ทำหน้าที่ในการพักเก็บข้อมูลที่ได้รับมาจาก rxdata เพื่อรอให้ทางด้าน UART Controller ดึงข้อมูลไปใช้

ตารางที่ 3.6 รายละเอียดการทำงานของขาต่างๆ ภายในบล็อก Endpoint

Pin	Input/Output	Description
clk	Input	สัญญาณนาฬิกาจาก clk_image
din	Input	ข้อมูลที่ได้รับเข้ามาจากทางด้าน UTMI
Rs_request	Input	รับสัญญาณการขอข้อมูลจาก RX232
Store_en	Input	รับคำสั่งให้ทำการเก็บข้อมูล จาก RXData Module
Clear_endp	input	รับคำสั่ง Clear Endpoint จาก RXData Module
Endp_full	Output	แสดงข้อมูลที่รับเข้ามานั้นเต็ม Endpoint แล้ว
Data_ready	Output	แสดงสถานะ UART ทำการดึงเอาข้อมูลไป
dout	Output	แสดงค่าของข้อมูลที่จะถูกส่งออกไป

การจำลองการทำงาน



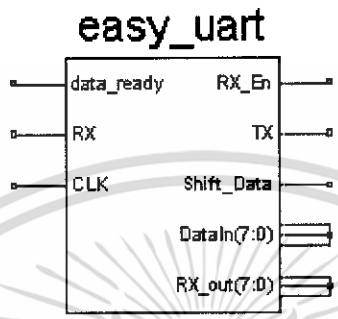
รูปที่ 3.17 รูปคลื่นที่ได้จากการจำลองการทำงานของบล็อก Endpoint

คำอธิบายการทำงาน

เมื่อมีสัญญาณ Store_en เข้ามาที่ Module จะเริ่มทำการรับค่าเข้ามาเก็บจนกระทั่ง Store_en มีค่าเป็น 0 ถึงจะหยุดทำการเก็บข้อมูล และถ้ามีสัญญาณ Clear_buff ตอบสนองมาจาก RXData ซึ่งจะทำให้สัญญาณ Data_Ready เป็น 0 และจะทำให้ข้อมูลไม่ถูกส่งออกไปให้ Easy UART แต่ถ้าข้อมูลที่รับเข้ามานั้นถูกต้อง Data Ready จะเป็น 1 และจะมีสัญญาณ Shift_Data ส่งออกมาจาก Easy UART ไปเรื่อยๆจนกระทั่งหมดส่วนที่เป็นข้อมูลที่แท้จริง (ไม่รวม CRC16 2 byte) Data_Ready จะกลับเป็น 0 อีกครั้ง รอรับข้อมูลชุดใหม่ที่จะรับเข้ามา

3.4 การออกแบบในส่วนของ UART Controller (Easy-UART)

ทำหน้าที่ในการเปลี่ยนข้อมูลแบบขนานไปเป็นแบบอนุกรม ซึ่งเป็นโปรโตคอลในแบบ RS-232 และเปลี่ยนข้อมูลอนุกรมที่มาจากทางด้าน RS-232 ไปเป็นแบบขนาน ซึ่งส่งกลับไปยัง Host ที่มีสัญญาณ 232-request ใช้ในการเลื่อนข้อมูลใหม่ที่ Endpoint เพื่อส่งออกไปยัง RS-232 ซึ่งแต่ละขาทำงานดังนี้

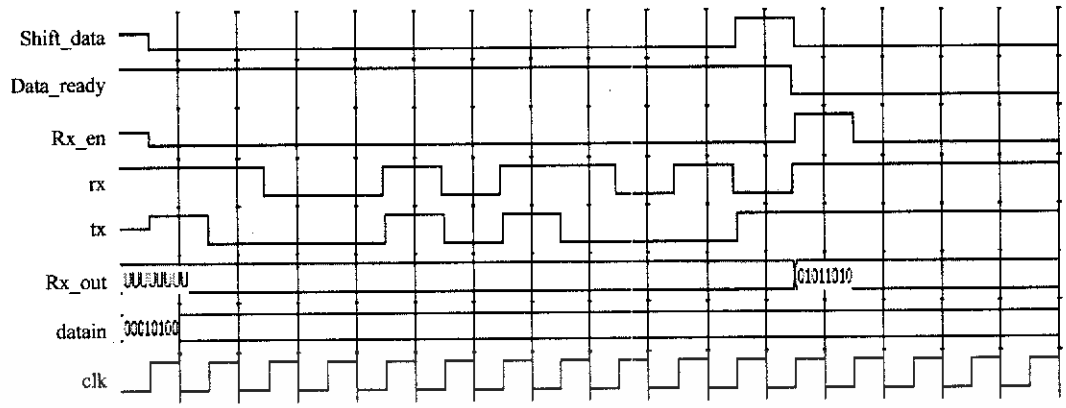


รูปที่ 3.18 บล็อกการออกแบบในส่วนของ Easy-UART

ตารางที่ 3.7 รายละเอียดการทำงานของขาต่างๆ ภายในบล็อก UART Controller (Easy-UART)

Pin	Input/Output	Description
Data_ready	Input	รับสถานะข้อมูลพร้อมส่งออกจาก Endpoint Module
RxD	Input	รับ Input จาก RS232
Datain	Input	รับข้อมูล Data จาก Endpoint Module
clk	Input	รับสัญญาณนาฬิกาจาก Baudrate Module
Shift_Data	Output	ส่งสัญญาณ Shift ข้อมูลไปให้ Endpoint
RX_En	Output	รับการ Enable เพื่อส่งข้อมูลที่รับมาจาก RS232 เข้าไปใน Module
TxD	Output	ใช้ส่งข้อมูลที่แปลงเป็นแบบ RS232 ออกไปยังอุปกรณ์
RX_out	Output	ใช้ส่งข้อมูลเข้าไปใน Module

การจำลองการทำงาน



รูปที่ 3.19 รูปคลื่นที่ได้จากการจำลองการทำงานของบล็อก Easy-UART

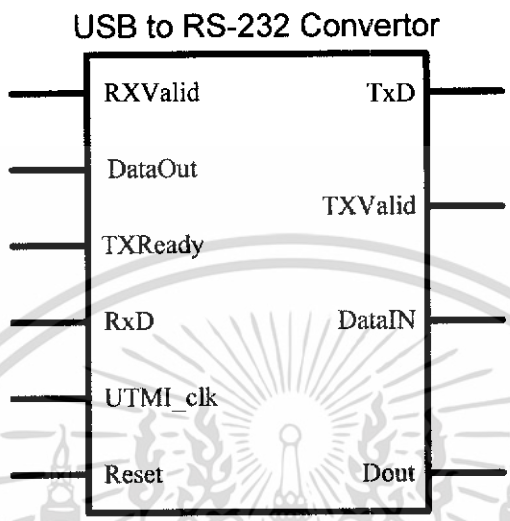
คำอธิบายการทำงาน

ข้อมูลที่รับถูกต้องที่รับเข้ามาที่ Endpoint แล้ว จะถูกส่งมายัง Easy UART เพื่อทำการส่งข้อมูลออกทาง Port TxD โดยข้อมูลที่ส่งออกไปนั้นจะเป็น Serial ซึ่งภายใน Module จะทำการเปลี่ยนข้อมูลที่รับเข้ามาในรูปแบบขนาน ให้เป็นแบบอนุกรมตาม Protocol ของ RS232

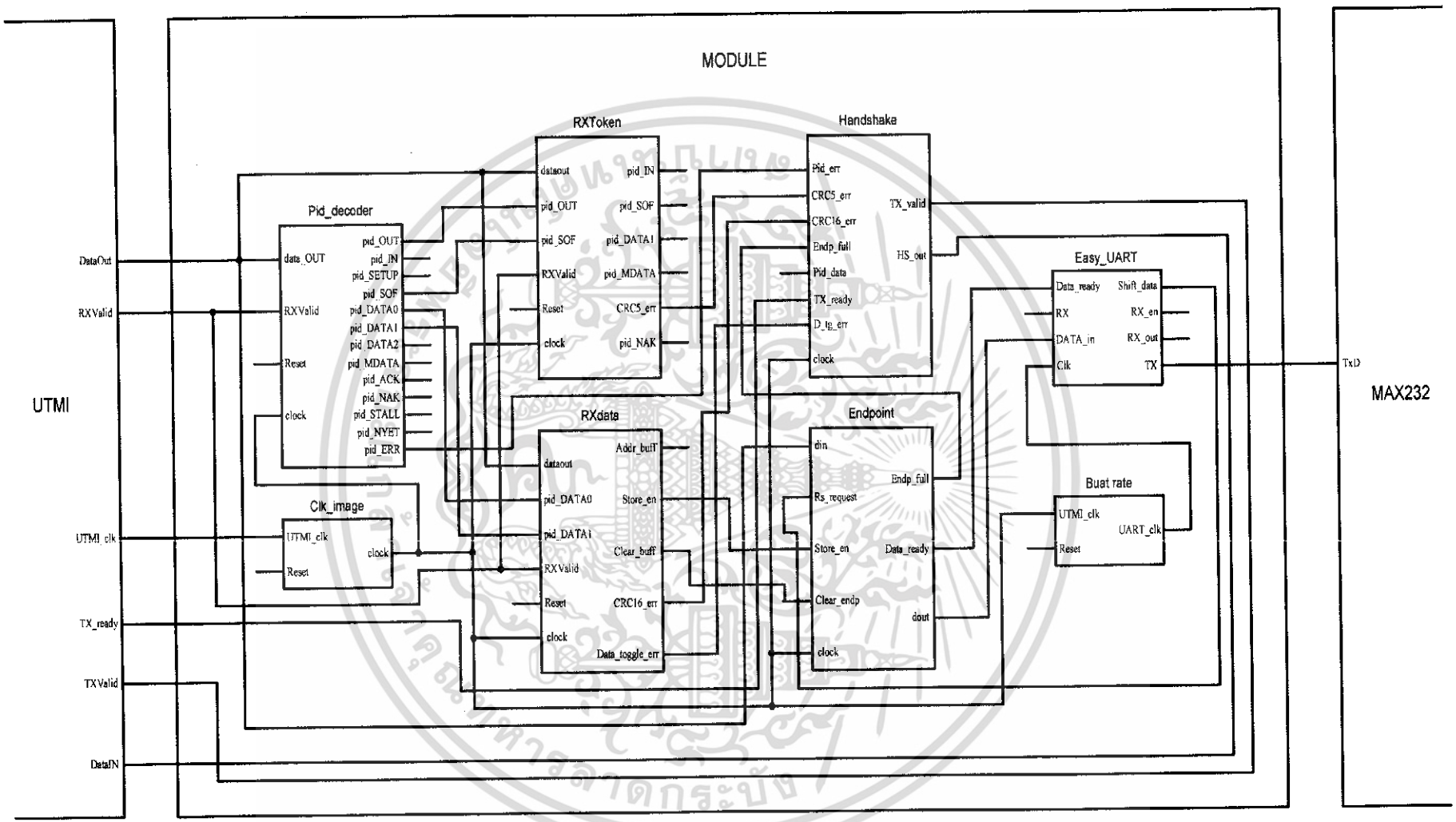
บทที่ 4

ผลการจำลองการทำงาน

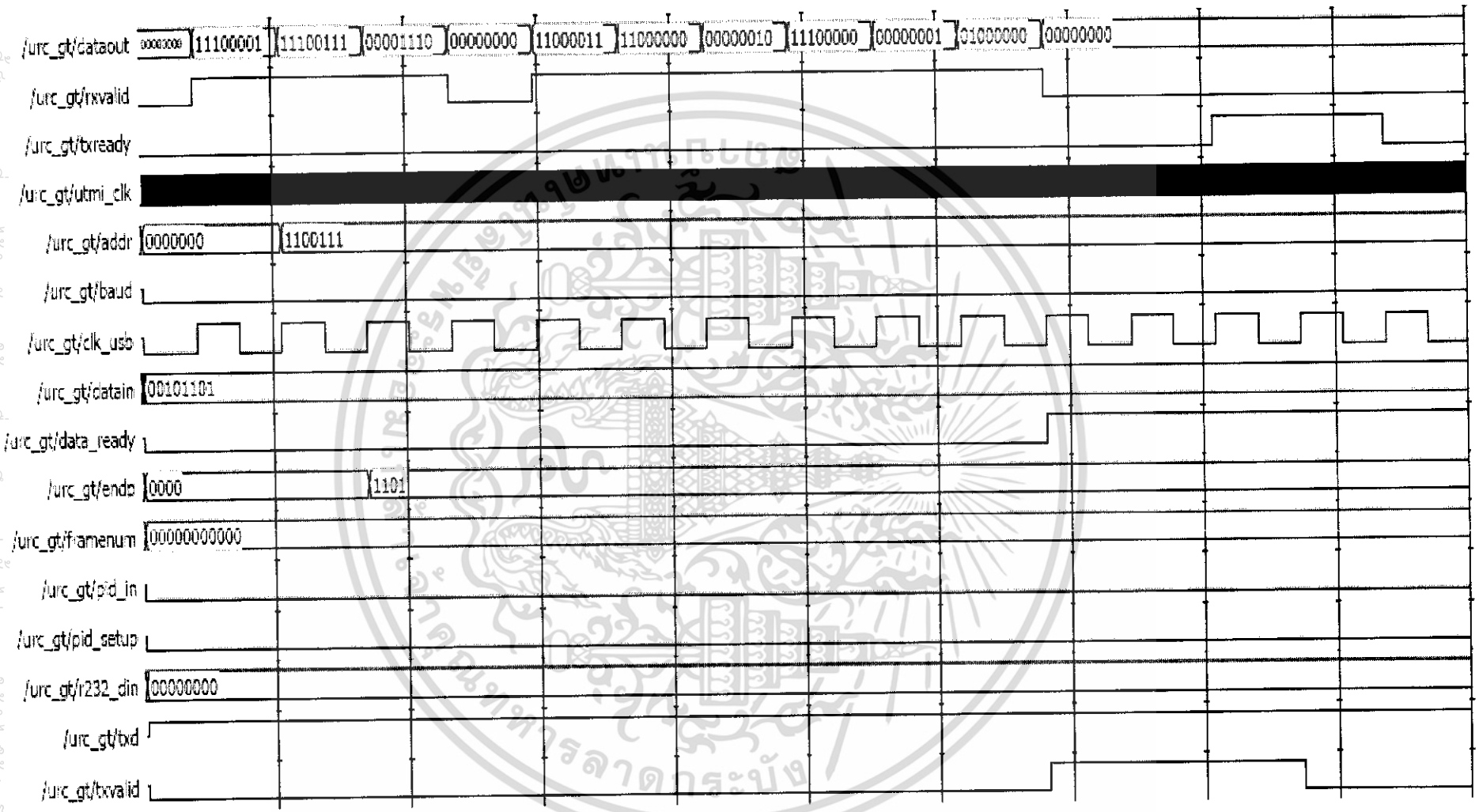
4.1 ผลการจำลองการทำงานของโมดูลที่ออกแบบ



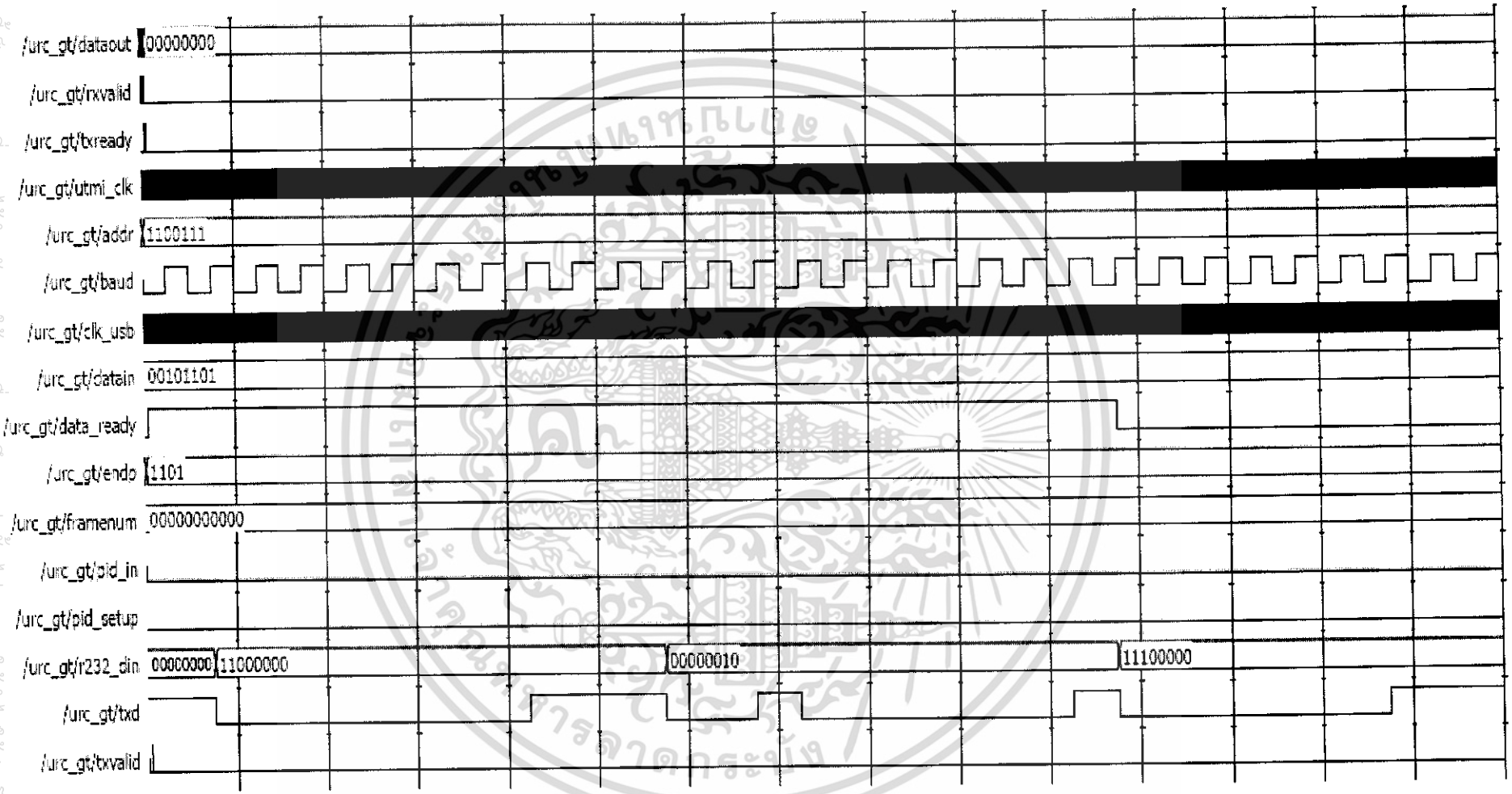
รูปที่ 4.1 โมดูลที่ออกแบบระดับบนสุด



รูปที่ 4.2 แสดงวงจรภายในโมดูลระดับบนสุด



รูปที่ 4.3 แสดงผลการจำลองการทำงานในช่วง 0-10 ไมโครวินาที



รูปที่4.4 แสดงผลการจำลองการทำงานในช่วง 0-3 มิลลิวินาที

อธิบายกราฟ

เริ่มต้นด้วย Host จะส่ง PID ออกมาก่อน โดย PID_out ซึ่งมีค่าเป็น 11100001 ข้อมูลที่มากับ PID_out จะเป็น Address และ Endpoint ในการจำลองการทำงานในครั้งนี้สมมุติให้ Address คือ 11001111 และ Endpoint คือ 1101 ซึ่งจะมีค่าของ CRC5 ที่ได้จากการคำนวณต่อท้ายมาด้วย ดังนั้นจะสามารถจัดเรียงข้อมูลลงใน Token packet ส่งมายังโมดูลที่ออกแบบได้เป็น

PID_out (8 bit)	11100111	00001110

รูปที่ 4.5 การจัดเรียงข้อมูลใน Token packet

โมดูลจะรับรู้ว่ามีข้อมูลเข้ามาหรือไม่โดยตรวจสอบขาของ Rxvalid ซึ่งจะมีสถานะเป็น High เมื่อมีข้อมูลโดยโมดูลที่ออกแบบก็จะตรวจสอบข้อมูลที่เข้ามาว่าถูกต้องหรือไม่โดยตรวจจาก CRC5 ที่โมดูลคำนวณได้ว่าตรงกับ CRC5 ที่รับเข้ามากับข้อมูลหรือไม่ หากไม่ตรงกันโมดูลก็จะไม่สนในข้อมูลที่เข้ามาใน Data packet และจะตอบ Handshake กลับไปเมื่อถึงช่วงของ Handshake packet แต่ในที่นี้ข้อมูลที่เข้ามานั้น CRC5 ถูกต้อง ดังนั้นข้อมูลที่เป็น Address และ Endpoint จะถูกเก็บไว้โดยแสดงผลที่ขาของ Endpoint และ Address ตามรูปที่ 4.3

ในช่วง Data_packet จะเริ่มด้วย Rxvalid เปลี่ยนสถานะเป็น 1 พร้อมๆ กับข้อมูลที่ขา Data_out ซึ่งเป็นค่าของ PID_data0 ใน clock แรกของ usb_clock จากนั้นจะตามด้วยข้อมูลอีก 3 ไบต์ และตามด้วย CRC16 อีก 2 ไบต์ โดยการจัดเรียง Data packet จะเป็นตามรูป

PID_data0	11000000	00000010	11100000	00000001	01000000

รูปที่ 4.6 การจัดเรียงข้อมูลใน Data packet

โดยโมดูลจะทำการคำนวณค่าของ CRC16 ในทุกๆ ไบต์ของข้อมูลที่เข้ามาซึ่งจะคำนวณเลยไปถึง CRC16 ด้วย ผลการคำนวณจะอยู่ภายในโมดูลซึ่งจะอยู่ในรูปของตัวเลข 16 บิต ถ้าตัวเลขทั้ง 16 บิตนั้นไม่เป็น 0 ทั้งหมด CRC16_err จะมีค่าเป็น 1 แต่ในที่นี้ข้อมูลที่ทำการป้อนและค่าของ CRC16 ที่รับเข้ามานั้นถูกต้อง CRC16_err จะมีค่าเป็น 0

ในขณะเดียวกัน ก็จะทำการเก็บข้อมูลเข้าไปยัง Endpoint ด้วยการทำสถานะ Store_en ให้เป็น 1 เพื่อบอกให้ Endpoint ทำการเก็บข้อมูลที่รับเข้ามา ซึ่งถ้าข้อมูลที่รับเข้ามานั้นไม่มีความผิดพลาด Endpoint จะทำการบอกไปยัง UART เพื่อให้ UART มาทำการ Shift ข้อมูลออกไป

โดย Endpoint จะทำการบอกไปยัง UART เพื่อให้ UART มาทำการ Shift ข้อมูลออกไป โดย Endpoint จะทำการเปลี่ยนข้อมูลที่จะส่งไปยัง UART ทุกครั้งที่มีสัญญาณ Shift มาจาก UART

ในช่วงของ Handshake packet นั้นโมดูลต้องทำสถานะไปบอก UTMI ว่าต้องการจะส่งข้อมูลกลับ โดยทำ Txvalid ให้เป็น 1 และรอจนกว่า Txready ซึ่งเป็นสถานะบอกว่าพร้อมส่งข้อมูลออกไปมีค่าเป็น 1 เมื่อ Txready มีค่าเป็น 1 Txvalid จะมีค่าเป็น 0 ใน clock ถัดไปเป็นการบอกให้ UTMI รู้ว่าข้อมูลที่จะทำการส่งออกไปนั้นส่งเข้าไปที่ขา DataIn ของ UTMI โดยข้อมูลที่ส่งออกไปจะอยู่ในรูปของ PID ของ Handshake

ในขณะที่ส่วน USB ทำการตอบสถานะกลับไปยังโฮส ส่วนที่เป็น RS-232 จะเริ่มทำการส่งข้อมูล โดยโปรโตคอลที่ใช้ในโครงงานนี้เป็นลักษณะคือมี start bit 1 bit, ข้อมูลที่จะส่ง 8 บิต, ไม่มี parity bit และตามด้วย stop bit อีก 1 บิต ซึ่งจะได้ผลการทดลองดังรูปที่ 4.4 โดย din จะเป็นตัวแสดงถึงข้อมูลที่ทำการส่งออกไปยัง TxD

บทที่ 5
สรุปและวิจารณ์ผลการทดลอง

ผลการทดลองจากกราฟ ข้อมูลที่เป็น Real Data ที่มาในแบบของ USB สามารถส่งออกทางพอร์ท TxD ได้จริง แต่จะเห็นได้ว่าความเร็วในการส่งแบบ USB นั้นเร็วกว่า RS232 มาก ดังนั้นถ้าหากนำไปประยุกต์ใช้จริง จะต้องเพิ่มขนาดของส่วนเก็บข้อมูล เพื่อให้ USB สามารถส่งถ่ายข้อมูลได้อย่างต่อเนื่อง ในการทดลองนี้ สามารถกำหนดช่วงเวลา End Time ได้สูงสุดเพียงแค่ 3 mS เท่านั้น เนื่องจากข้อจำกัดของ Computer และสัญญาณจะต้องทำการประมวลผลนั้นมีมาก ทำให้เมื่อทำการประมวลผลข้อมูลไปได้ระยะหนึ่ง Windows ก็ทำการปิดโปรแกรมที่ใช้ในการประมวลผลนั้นเพราะโปรแกรมนั้นไม่มีการตอบสนอง (No Response) ซึ่งเป็นข้อจำกัดในการทดสอบการทำงานของโครงงานนี้ อีกทั้งรูปแบบการส่งข้อมูลในแบบ USB นั้นเพิ่งได้รับความนิยมอย่างสูง จึงทำให้มีข้อมูลในการทำการค้นคว้าค่อนข้างน้อย เนื่องจากผู้ผลิตอุปกรณ์ประเภท USB Device นั้นไม่เปิดเผยเพื่อประโยชน์ของผู้ผลิตเอง ดังนั้นโครงงานนี้จึงเป็นเพียงแนวคิดหนึ่งเท่านั้น ซึ่งในอนาคตข้างหน้าแนวคิดนี้อาจเปลี่ยนไป ทั้งนี้ทั้งนั้นขึ้นกับมีข้อมูลที่จำเป็นต้องใช้ในโครงงานมากน้อยเพียงใด

บรรณานุกรม

1. วรเทพ ไพบูลย์รัตนากร, "สัมผัสโลก USB ด้วย EZY USB Module", Astron logic research & development, 2537
2. www.opencores.org
3. www.usb.org
4. www.intel.com/technology/usb/download/2_0_Xcvr_macrocell_1_05.pdf

ภาคผนวก

BAUDGEN

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Baudgen is
        port(   clk      :in std_logic;
                uartclk :out std_logic);
end baudgen;

architecture behavior of baudgen is
begin

process(clk)
variable Stage : std_logic;
variable I       : integer range 0 to 2499;
begin
        if (clk'event and clk='1') then
                if i=2499 then
                 stage := not stage;
                uartclk<=stage;
                I := 0;
                else
                stage := stage;
                uartclk <= stage;
                i:=i+1;
                end if;
        end if;
end process;

end behavior;
```

USB_CLK

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity USB_clk is
   Port ( clk_in  : in  std_logic;
          clk_out : out std_logic);
end USB_clk;

architecture Behavioral of USB_clk is
begin

process(clk_in)
variable Stage : integer range 0 to 1;
variable i1    : integer range 0 to 15;
variable i2    : integer range 0 to 15;
begin
        if (clk_in'event and clk_in='1') then
                if Stage=1 then
                        if i1=15 then
                        clk_out<='0';
                        Stage := 0;
                        i1 := 0;
                        else
                        clk_out <= '1';
                        i1:=i1+1;
                        end if;
                elsif Stage=0 then
                        if i2=15 then
                        clk_out<='1';
                        Stage :=1;
                        i2:= 0;
                        else
                        clk_out <= '0';
                        i2:=i2+1;
                        end if;
                end if;
        end if;
 end process;

 end Behavioral;
```

PID_DECODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pid_decode is
   Port ( DataOut : in std_logic_vector(7 downto 0);

          RXValid : in std_logic;
          Clock : in std_logic;
          pid_OUT : out std_logic;
          pid_IN : out std_logic;
          pid_SOF : out std_logic;
          pid_SETUP : out std_logic;
          pid_DATA0 : out std_logic;
          pid_DATA1 : out std_logic;
          pid_Err : out std_logic);
end pid_decode;

architecture Behavioral of pid_decode is
shared variable store : std_logic_vector(7 downto 0);
begin

Receive_PID: process(clock,dataout,RXValid)
begin
          if RXValid'event and RXValid = '1' then
          store := DataOut;
          end if;
end process;

Decode_PID: process(clock,dataout,RXValid)
begin
          if RXValid = '1' then
                    if clock'event and clock = '0' then
                              if store = "11100001" then
                              pid_OUT  <= '1';
                              elsif    store = "10100101" then
                              pid_SOF  <= '1';

                              elsif    store = "00101101" then
                              pid_SETUP    <= '1';
                              elsif    store = "11000011" then
                              pid_DATA0    <= '1';
                              elsif    store = "01001011" then
                              pid_DATA1 <= '1';
                              else
                              pid_Err <= '1';
                              end if;
```

```
            end if;
        else
                    pid_OUT <= '0';
                    pid_SOF  <= '0';
                    pid_SETUP <= '0';
                    pid_DATA0 <= '0';
                    pid_DATA1 <= '0';
                    pid_Err  <= '0';
        end if;
end process;

end Behavioral;
```

RXTOKEN

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RXToken is
   Port ( dataOut : in std_logic_vector(7 downto 0);
          pid_OUT : in std_logic;
          pid_SOF : in std_logic;
          Clock : in std_logic;
          Addr : out std_logic_vector(6 downto 0);
          Endp : out std_logic_vector(3 downto 0);
          FrameNumber : out std_logic_vector(10 downto 0);
          CRC5_err : out std_logic;
          rxvalid : in std_logic);

end RXToken;

architecture Behavioral of RXToken is

type t_out_stg is (RCV0,RCV1);
signal state_out : t_out_stg;
type t_SOF_stg is (RCV_SOF0,RCV_SOF1);
signal state_SOF : t_SOF_stg;
shared variable ADDR_v : std_logic_vector(6 downto 0);
shared variable ENDP_v : std_logic_vector(3 downto 0);
shared variable CRC5_r : std_logic_vector(4 downto 0);
shared variable CRC5 :  std_logic_vector(4 downto 0);
shared variable CRC5_OUT_en : std_logic;
shared variable CRC5_SOF_en : std_logic;
shared variable data :  std_logic_vector(15 downto 0);

begin

Decode : process(clock,pid_OUT,dataOut)
begin
        if clock'event and clock = '1' then
                if pid_OUT = '1' then
                case state_out is
                when RCV0 => data(7 downto 0):= dataOut;
                                Addr(6 downto 0)  <= data(6 downto 0);
                                Addr_v := data(6 downto 0);
                                state_out <= RCV1;
                                CRC5_OUT_en := '0';
                when RCV1 => data(15 downto 8) := dataOut;
                                Endp(3 downto 0)  <= data(10 downto 7);
                                Endp_v := data(10 downto 7);
                                CRC5_OUT_en := '1';
```

```
                        CRC5_r      := data(15 downto 11);
                        state_out <= RCV0;
            end case;
            else    state_out <= RCV0;
            end if;

            if pid_SOF = '1' then
            case state_SOF is
            when RCV_SOF0 => data(7 downto 0):= dataOut;
                        state_SOF <= RCV_SOF1;
                        CRC5_SOF_en := '0';
            when RCV_SOF1 => data(15 downto 8) := dataOut;
                        FrameNumber <= data (10 downto 0);
                        CRC5_SOF_en := '1';
                        CRC5_r := data(15 downto 11);
                        state_SOF <= RCV_SOF0;
            end case;
            else    state_SOF <= RCV_SOF0;
            end if;
        end if;
end process;

CRC5_check : process (pid_OUT,pid_SOF,clock)
begin
if (CRC5_OUT_en or CRC5_SOF_en) = '1' then
        CRC5(0) := data(10) xor data(9) xor data(6) xor data(5)
                        xor data(3) xor data(0);
        CRC5(1) := data(10) xor data(7) xor data(6) xor data(4)  xor data(1);
        CRC5(2) := data(10) xor data(9) xor data(8) xor data(7)
                        xor data(6) xor data(3) xor data(2) xor data(0);
        CRC5(3) := data(10) xor data(9) xor data(8) xor data(7)
                        xor data(4) xor data(3) xor data(1);
        CRC5(4) := data(10)  xor data(9) xor data(8) xor data(5)
                        xor data(4) xor data(2);
end if;

if clock'event and clock = '1' then
        if (pid_OUT or pid_SOF) = '1' then
                CRC5_err <= '0';
        elsif (CRC5_OUT_en or CRC5_SOF_en)= '1' then
                if CRC5 = CRC5_r then
                CRC5_err <= '0';
                else
                CRC5_err <= '1';
                end if;
        end if;
end if;
end process;

end Behavioral;
```

RXDATA

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rxdata is
   Port ( dataout : in std_logic_vector(7 downto 0);
          rxvalid: in std_logic;
          pid_data0 : in std_logic;
          pid_data1 : in std_logic;
          clock : in std_logic;
          store_en : out std_logic;
          clear_buff : out std_logic;
          data_toggle_err : out std_logic;
          crc16_err: out std_logic );
end rxdata;

architecture Behavioral of rxdata is

signal pid_data : std_logic;
shared variable crc_out : std_logic_vector (15 downto 0);
shared variable toggle_status : std_logic := '0';
shared variable toggle_status_new : std_logic := '0';
shared variable din : std_logic_vector (7 downto 0);
shared variable stg : std_logic;
shared variable stg_new : std_logic;
shared variable crc_in : std_logic_vector (15 downto 0);

begin

PID_DATA_IN : process(pid_DATA0,pid_DATA1)
begin
        if pid_DATA0 = '1' then
        pid_data <= '0';
        elsif pid_DATA1 = '1' then
        pid_data <= '1';
        end if;
end process;

dt2 : process(pid_data,clock,rxvalid)
begin
        if pid_data'event and pid_data= '1' then
        toggle_status := toggle_status_new;
        end if;
end process;


dt3 : process(pid_DATA0,pid_DATA1)
```

```
begin
        if (pid_DATA0 or pid_DATA1) = '1' then
                if pid_data = toggle_status then
                store_en <= '1';
                data_toggle_err <= '0';
                toggle_status_new := not toggle_status;
                else
                store_en <= '0';
                data_toggle_err <= '1';
                toggle_status_new := toggle_status;
                end if;
        else
                store_en <= '0';

        end if;
end process;

crc16_1 : process(pid_data0,pid_data1)
begin
        if (pid_data0 or pid_data1)  = '0' then
                crc_in := "0000000000000000";
        elsif (pid_data0 or pid_data1) = '1' then
                crc_in := crc_out;
        end if;
end process;

crc16_2 : process(pid_data0,pid_data1,dataout,clock)
begin
        if clock'event and clock = '1' then
                if (pid_data0 or pid_data1) = '1' then
                din := dataout;
                crc_out(0) := din(7) xor din(6) xor din(5) xor din(4)
                        xor din(3) xor din(2) xor din(1) xor din(0) xor crc_in(8)
                        xor crc_in(9) xor crc_in(10) xor crc_in(11) xor crc_in(12)
                        xor crc_in(13) xor crc_in(14) xor crc_in(15);
                crc_out(1) := din(7) xor din(6) xor din(5) xor din(4) xor din(3)
                        xor din(2) xor din(1) xor crc_in(9) xor crc_in(10)
                        xor crc_in(11) xor crc_in(12) xor crc_in(13) xor crc_in(14)
                        xor crc_in(15);
                crc_out(2) := din(1) xor din(0) xor crc_in(8) xor crc_in(9);
                crc_out(3) := din(2) xor din(1) xor crc_in(9) xor crc_in(10);
                crc_out(4) := din(3) xor din(2) xor crc_in(10) xor crc_in(11);
                crc_out(5) := din(4) xor din(3) xor crc_in(11) xor crc_in(12);
                crc_out(6) := din(5) xor din(4) xor crc_in(12) xor crc_in(13);
                crc_out(7) := din(6) xor din(5) xor crc_in(13) xor crc_in(14);
                crc_out(8) := din(7) xor din(6) xor crc_in(0) xor crc_in(14)
                        xor crc_in(15);
                crc_out(9) := din(7) xor crc_in(1) xor crc_in(15);
                crc_out(10) := crc_in(2);
                crc_out(11) := crc_in(3);
```

```
                crc_out(12) := crc_in(4);
                crc_out(13) := crc_in(5);
                crc_out(14) := crc_in(6);
                crc_out(15) := din(7) xor din(6) xor din(5) xor din(4) xor din(3)
                        xor din(2) xor din(1) xor din(0) xor crc_in(7) xor crc_in(8)
                        xor crc_in(9) xor crc_in(10) xor crc_in(11) xor crc_in(12)
                        xor crc_in(13) xor crc_in(14) xor crc_in(15);
                else
                din := "00000000";
                crc_out := crc_out;
                end if;
        end if;
end process;

crc16_4 : process(pid_DATA0,pid_DATA1)
begin
        if  (pid_data0 or pid_data1) = '0' then
                if crc_out = "0000000000000000" then
                clear_buff <= '0';
                crc16_err <= '0';
                else
                clear_buff <= '1';
                crc16_err <= '1';
                end if;
        else
                clear_buff <= '0';
                crc16_err <= '0';
        end if;
end process;

end Behavioral;
```

UART

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity uart is
   Port ( Shift_Data : out std_logic;
          data_ready : in std_logic;
          TX : out std_logic;
          DataIn : in std_logic_vector(7 downto 0);
          CLK : in std_logic);
end uart;

architecture Behavioral of uart is

type state_type_tx is (idel_tx,readyTX,send,stop_tx);
signal state_tx :state_type_tx;

begin

transmiter : process(data_ready,DataIn,CLK)
variable TX_data_count : integer range 0 to 7 ;
begin

if CLK'event and CLK='1' then
          case state_tx is
          when idel_tx   => Shift_Data <= '0';
                            tx <= '1';
                            tx_data_count := 0;
                    if data_ready = '1'then
                            state_tx <= readyTX;
                            Shift_Data <= '0';
                    else
                            state_tx <= idel_tx;
                    end if;
          when readyTX=>
                    if  data_ready = '1' then
                            tx <= '0';
                            Shift_Data <= '1';
                            state_tx <= send;
                    else
                            tx <= '1';
                            Shift_Data <='0';
                            state_tx <= idel_tx;
                    end if;
          when send=>
                    if data_ready = '1' then
                            if  TX_data_count = 7 then
```

```
                    TX <= DataIn(TX_data_count);
                    Shift_Data <= '0';
                    state_tx <= stop_tx;
                else
                    TX <= DataIn(TX_data_count);
                    TX_data_count := TX_data_count+1;
                    state_tx <= send;
                    Shift_Data <= '0';
                end if;
            else
                tx <= '1';
                state_tx <= idel_tx;
                Shift_Data <= '0';
            end if;
        when stop_tx  =>
            if data_ready = '1' then
                TX <= '1';
                TX_data_count := 0;
                state_tx <= readyTX;
                Shift_Data <= '0';
            else
                tx <= '1';
                state_tx <= idel_tx;
                Shift_Data <= '0';
            end if;
        end case;
end if;
end process;

end Behavioral;
```

ENDPOINT

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity EP is
        Port ( clk_usb : in std_logic;
        din : in std_logic_vector(7 downto 0);
        rs_req : in std_logic;
        store_en : in std_logic;
        clear_ep : in std_logic;
        ep_full : out std_logic;
        data_ready : out std_logic;
        dout : out std_logic_vector(7 downto 0));
end EP;

architecture Behavioral of EP is

shared variable n,m,n_sub,m_sub : integer range 0 to 65 := 0;
type endp is array(0 to 65) of std_logic_vector(7 downto 0);
signal ep : endp;
signal data_ready_a,data_ready_b : std_logic;

begin
data_ready <= data_ready_a and data_ready_b;

a : process(clk_usb,din,store_en)
        begin
                if rising_edge(clk_usb)         then
                        if store_en = '1' then
                        ep(n) <= din;
                                if n = 65 then
                                ep_full <= '1';
                                n_sub := n;
                                else ep_full <= '0';
                                end if;
                        n := n+1;
                        n_sub := n;
                        else
                        n := 0;
                        end if;
                end if;
 end process a;
```

```
b : process(clk_usb,store_en)
begin
        if store_en = '0' then
                if      rs_req'event and rs_req = '1' then
                        dout <= ep(m);
                        m := m+1;
                        m_sub := m;
                end if;
        else m := 0;
        end if;
end process b;

c : process(store_en,clk_usb)
begin
        if falling_edge(store_en) then
                data_ready_a <= '1';
        end if;
        if m_sub = n_sub-2 then
                data_ready_a <= '0';
        end if;
end process c;

d : process(store_en,clk_usb)
begin
        if rising_edge(clk_usb) then
                if store_en = '1' then
                        data_ready_b <= '0';
                else data_ready_b <= '1';
                end if;
        end if;
end process d;

end Behavioral;
```

HANDSHAKE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity handshake is
Port ( clk : in std_logic;
     pid_err : in std_logic;
     crc5_err : in std_logic;
     data_toggle_err : in std_logic;
     crc16_err : in std_logic;
     endp_full : in std_logic;
                tx_ready : in std_logic;
                pid_data : in std_logic;
                tx_valid : out std_logic := '0';
                hs_out : out std_logic_vector(7 downto 0));
end handshake;

architecture Behavioral of handshake is

type state_txvalid is (idel,ready,wait_txready,drop_txvalid);
signal state    :       state_txvalid;
signal status : std_logic;

begin

status <= pid_err or crc5_err or data_toggle_err or crc16_err or endp_full;

p1: process(status)
begin
        if status = '0' then
                hs_out <= "00101101"; --ack
        elsif status = '1' then
                hs_out <= "10100101"; --nak
        elsif endp_full = '1' then
                hs_out <= "11100001"; --stall
        end if;
end process;

p4: process(clk,pid_data,tx_ready)
begin
if clk'event and clk = '1' then
        case state is
        when idel=>   if pid_data = '0' then
                      state <=idel;
                      tx_valid <= '0';
                      else
                      tx_valid <= '0';
```

```
                    state <= ready;
                    end if;
        when ready=> if pid_data = '1' then
                    state <= ready;
                    tx_valid <= '0';
                    else
                    state <= wait_txready ;
                    tx_valid <= '1';
                    end if;
        when wait_txready=>
                    if pid_data = '0' then
                            if tx_ready      = '0' then
                            tx_valid <= '1';
                            state <=wait_txready;
                            else
                            tx_valid <= '1';
                            state <= drop_txvalid;
                            end if;
                    else
                    tx_valid <= '0';
                    state <= ready ;
                    end if;
        when drop_txvalid =>
                    if pid_data = '0' then
                    tx_valid <= '0';
                    state <= idel;
                    else
                    tx_valid <= '0';
                    state <= ready;
                    end if;
        when others   =>      state <= idel;
                    tx_valid <= '0';
        end case;
end if;
end process;

end Behavioral;
```