บทที่ 4

การพัฒนาและทดสอบระบบเชิงเวลาจริงที่ตระหนักถึงกำลังงานที่ใช้

4.1 การพัฒนาระบบเชิงเวลาจริงที่ตระหนักถึงกำลังงานที่ใช้

การพัฒนาระบบที่เกี่ยวข้องสำหรับงานวิจัยประกอบด้วย 2 ส่วน คือ การปรับปรุงระบบปฏิบัติการเชิงเวลาจริง uC/OS-II ให้รองรับการตระหนักถึงกำลังงานที่ใช้ได้และการพัฒนาแอพลิเคชันสำหรับทดสอบการทำงานของระบบเชิงเวลาจริง

4.1.1 การปรับปรุงระบบปฏิบัติการเชิงเวลาจริง uC/OS-II

เทคนิคการลดกำลังงานในระดับระบบปฏิบัติการของระบบปฏิบัติการเชิงเวลาจริง คือ การจัดตารางงานและการเปลี่ยนขนาด Timer tick ดังได้กล่าวไว้แล้วในบทก่อนหน้านี้ ดังนั้นในหัวข้อนี้จะกล่าวถึงตารางงานของระบบปฏิบัติการ uC/OS-II และ timer tick

4.1.1.1 ตารางงานของระบบปฏิบัติการ uC/OS-II

การทำงานของระบบปฏิบัติการ uC/OS-II ทำตามลำดับความสำคัญที่ได้กำหนดไว้ในแต่ละงานขึ้นอยู่กับผู้พัฒนาระบบแต่สำหรับ idle task เป็นงานที่ระบบปฏิบัติการสร้างขึ้นโดยกำหนดลำดับความสำคัญต่ำที่สุดไว้ในส่วนของไฟล์ UCOS_II.H ดังภาพประกอบ 4-1

```
#define OS_IDLE_PRIO (OS_LOWEST_PRIO)   /* IDLE task priority */
```

ภาพประกอบ 4-1 การกำหนดลำดับความสำคัญของ idle task

ระบบปฏิบัติการ uC/OS-II สามารถกำหนดระดับความสำคัญของงานได้ทั้งหมด 64 ระดับตั้งแต่ 0 – 63 โดยที่ OS_LOWEST_PRIO มีค่าต่ำสุดซึ่งเท่ากับ 63 ดังที่ระบบได้กำหนดไว้ในไฟล์ os_cfg_r.h แสดงดังภาพประกอบ 4-2

```
#define OS_LOWEST_PRIO 63  /*Defines the lowest priority that can be assigned*/
                                                   /* ...  MUST NEVER be higher than 63!         */
```

ภาพประกอบ 4-2 การกำหนดค่าลำดับความสำคัญที่ต่ำที่สุด

จากลำดับความสำคัญสามารถวิเคราะห์ได้ว่างานทุกงานที่ผู้พัฒนาสร้างขึ้นจะได้รับการบริการก่อน idle task เสมอและ idle task จะทำงานเมื่องานอื่น ๆ ทำการประมวลผลเสร็จสิ้นแล้ว ฟังก์ชันการทำงานของ idle task อยู่ที่ไฟล์ OS_CORE.C ในฟังก์ชัน OS_TaskIdle ฟังก์ชัน idle task แสดงดังภาพประกอบ 4-3

```
void  OS_TaskIdle (void *parg)
{
#if OS_CRITICAL_METHOD == 3   /* Allocate storage for CPU status register  */
   OS_CPU_SR  cpu_sr;
   cpu_sr = 0;                                  /* Prevent compiler warning                    */
#endif


   parg   = parg;                  /* Prevent compiler warning for not using 'parg'      */


   for (;;)
   {
      OS_ENTER_CRITICAL();
      OSIdleCtr++;
      OS_EXIT_CRITICAL();
      OSTaskIdleHook();                        /* Call user definable HOOK                */
   }
}
```

ภาพประกอบ 4-3 การทำงานของ idle task ในฟังก์ชัน OS_TaskIdle

4.1.1.2 การทดสอบและวิเคราะห์การจัดตารางงาน

การออกแบบการทำงานของโปรแกรมเพื่อทดสอบการจัดตารางงานของ uC/OS-II ด้วยการสร้างงาน 3 งาน คือ TaskA, TaskB และ TaskC ที่มีลำดับความสำคัญ 0, 1 และ 2 ตามลำดับ พิจารณาการทำงานด้วยการโปรแกรมให้ LED เปลี่ยนสถานะกำหนดให้ LED_On เมื่อเริ่มทำงานและ LED_Off ก่อนพักการทำงาน โดย TaskA, B, C และ idle task ควบคุม LED1, 2, 3 และ 5 ตามลำดับ กำหนดให้ LED ทุกตัวมีค่าเริ่มต้นเป็น LED_Off ก่อนการทำงาน ส่วนที่มีการแก้ไขเพิ่มเติมใน uC/OS-II ได้แก่ การกำหนดลำดับความสำคัญและขนาด stack ของงาน, การสร้างงาน และการโปรแกรมการทำงานของงานทั้ง 3 งาน ตัวอย่างการเพิ่มเติมแก้ไขโปรแกรมแสดงดังภาพประกอบ 4-4

กำหนดลำดับความสำคัญและขนาด stack ของงานที่สร้างขึ้นในไฟล์ app_cfg.h

```
#define A_TASK_STK_SIZE    64
#define A_TASK_START_PRIO   0
#define B_TASK_STK_SIZE    64
#define B_TASK_START_PRIO   1
#define C_TASK_STK_SIZE    64
#define C_TASK_START_PRIO   2
```

ตัวอย่างการสร้าง TaskA

```
  OSTaskCreateExt(TaskA,
            (void *)0,
            (OS_STK *)&ATaskStk[A_TASK_STK_SIZE - 1],
            A_TASK_START_PRIO,
            A_TASK_START_PRIO,
            (OS_STK *)&ATaskStk[0],
            A_TASK_STK_SIZE,
            (void *)0,
            OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
```

ภาพประกอบ 4-4 ส่วนของโปรแกรมสำหรับวิเคราะห์การจัดการตารางงาน

ตัวอย่าง TaskA

```
static  void  TaskA (void *p_arg)
{
   unsigned long i;
   for (;;)
   {
      LED_Off(5);
      LED_On(1);
      i = 0;
      while (i < 1000000)
         i = i+1;
      LED_Off(1);
      OSTimeDlyHMSM(0, 0, 5, 0);
  }
```

ตัวอย่าง TaskB

```
static  void  TaskB (void *p_arg)
{
   unsigned long i;
   for (;;)
   {
      LED_On(2);
      i=0;
      while (i < 1000000)
         i = i+1;
      LED_Off(2);
      OSTimeDlyHMSM(0, 0, 5, 0);
   }
}
```

ภาพประกอบ 4-4 (ต่อ)

ตัวอย่าง TaskC

```
static  void  TaskC (void *p_arg)
{
   unsigned long i;
   for (;;)
   {
      LED_On(3);
      i = 0;
      while (i < 1000000)
         i = i+1;
      LED_Off(3);
      OSTimeDlyHMSM(0, 0, 5, 0);
   }
}
```

โปรแกรมเพิ่มเติมในฟังก์ชัน void OS_TaskIdle (void *parg) ของ idle task

```
void  OS_TaskIdle (void *parg)
{
#if OS_CRITICAL_METHOD == 3   /* Allocate storage for CPU status register  */
   OS_CPU_SR  cpu_sr;
   cpu_sr = 0;                                  /* Prevent compiler warning                   */
#endif


   parg   = parg;                  /* Prevent compiler warning for not using 'parg'      */
```

ภาพประกอบ 4-4 (ต่อ)

โปรแกรมเพิ่มเติมในฟังก์ชัน void OS_TaskIdle (void *parg) ของ idle task (ต่อ)

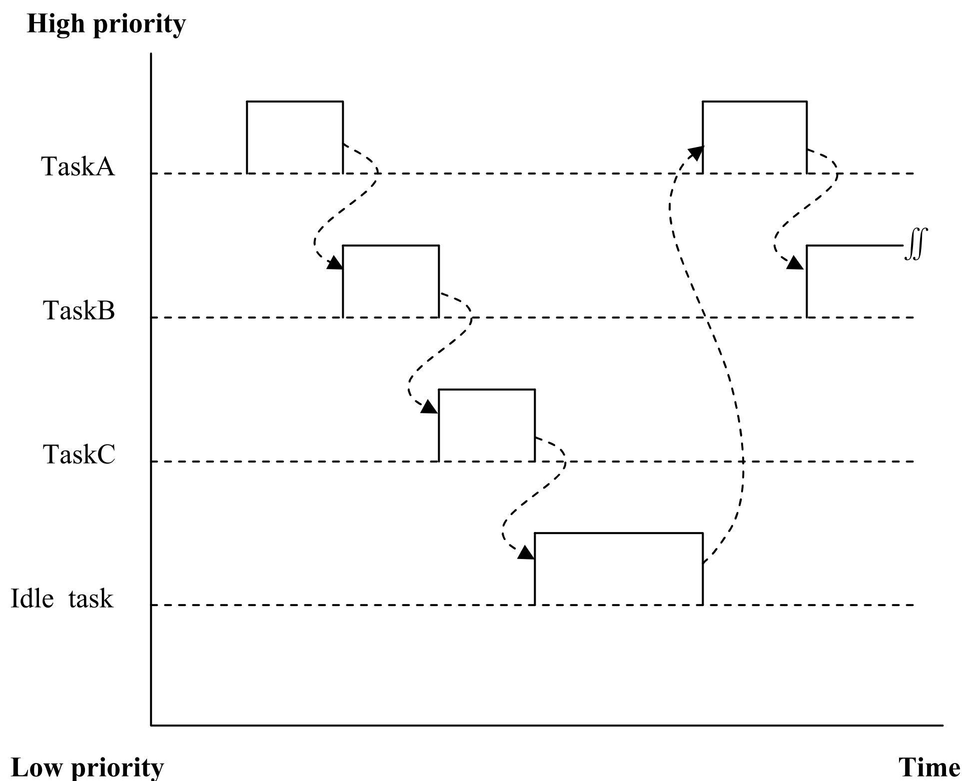
```
  for (;;)
  {
    OS_ENTER_CRITICAL();
    OSIdleCtr++;
    OS_EXIT_CRITICAL();
    OSTaskIdleHook();                /* Call user definable HOOK            */
    LED_On(5);
  }
}
```

ภาพประกอบ 4-4 (ต่อ)

ผลการทำงานของโปรแกรมมีดังนี้

1) LED ทุกตัวอยู่ในสถานะ Off ในขณะที่ LED ตัวที่ 1 On และ Off ในเวลาต่อมา
2) เมื่อ LED1 Off แล้ว LED2 เปลี่ยนสถานะ On และ Off ในเวลาต่อมา
3) เมื่อ LED2 Off แล้ว LED3 เปลี่ยนสถานะ On และ Off ในเวลาต่อมา
4) เมื่อ LED3 Off แล้ว LED5 เปลี่ยนสถานะ On และ Off ในเวลาต่อมา

นำผลการทดสอบที่ได้จากการทำงานทั้งหมดมาทำการวิเคราะห์ได้ว่าการประมวลผลของงานขึ้นอยู่กับลำดับความสำคัญที่กำหนดไว้ TaskA ได้เข้าทำงานก่อน TaskB สามารถเข้าทำงานได้เป็นลำดับที่สองเมื่อ TaskA พักการประมวลผล และ TaskC สามารถเข้าทำงานได้เมื่อ TaskA และ TaskB พักการประมวลผลในเวลาเดียวกันและ idle task เข้าทำงานเมื่อทั้งสามงานไม่มีการทำงานในเวลาเดียวกัน วิเคราะห์การจัดตารางงานตามช่วงเวลาและลำดับความสำคัญของงานบน uC/OS-II ดังภาพประกอบ 4-5



ภาพประกอบ 4-5 การทำงานของงานตามช่วงเวลาและลำดับความสำคัญ

การทดสอบการทำงานของระบบตามช่วงเวลานอกจากทดสอบด้วยการสังเกตจากการติดและดับของ LED ดังที่ได้กล่าวมาแล้วยังสามารถตรวจสอบได้จากการวัดกิจกรรมของแต่ละงานด้วย oscilloscope ลำดับการทำงานของระบบข้างต้นแสดงด้วย oscilloscope ดังภาพประกอบ 4-6

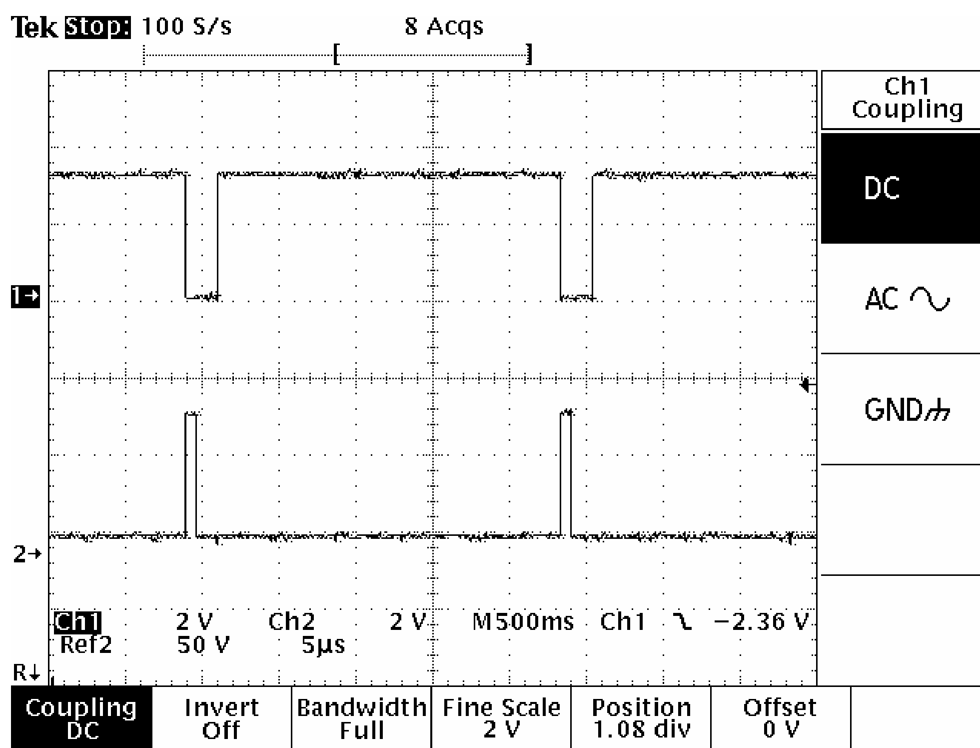
จากภาพประกอบ 4-6 ช่องสัญญาณ 1 แสดงสัญญาณการทำงานของ idle task ส่วนช่องสัญญาณที่ 2 แสดงสัญญาณการทำงานของแอพลิเคชันโปรแกรมของงาน เนื่องจาก oscilloscope ที่ใช้ (TDS 360) แสดงผลได้เพียง 2 ช่องสัญญาณเท่านั้น ดังนั้นจึงแสดงลำดับการเข้าทำงานได้ครั้งละหนึ่งงานโดยเปรียบเทียบกับ idle task

จากภาพประกอบ 4-6a ช่วงขาลงของช่องสัญญาณที่ 1 เป็นช่วงที่ idle task (LED 5) หยุดทำงานและในขณะเดียวกันงานอื่น ๆ จะเข้าใช้หน่วยประมวลผลแทนนั่นโดยพิจารณาจากช่องสัญญาณที่ 2 (ตรวจสอบจากการทำงานของ Task A ได้เพียง 1 งานซึ่งเป็นงานที่ลำดับความสำคัญสูงที่สุด)

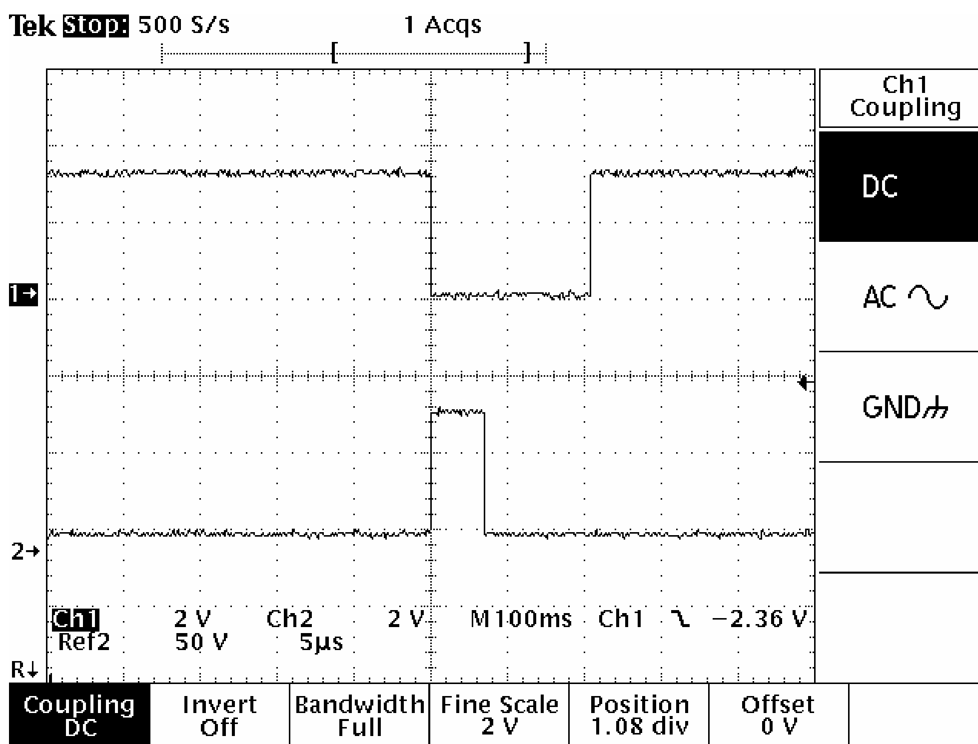
จากภาพประกอบที่ 4-6b ช่องสัญญาณที่ 2 เป็นการทำงานของ Task A (LED 1) เมื่อถึงเวลาที่ Task A (ลำดับความสำคัญสูงที่สุด) ต้องการใช้หน่วยประมวลผล idle task (ลำดับความสำคัญต่ำที่สุด) จะหยุดทำงานและปล่อยให้ Task A เข้าใช้หน่วยประมวลผลก่อน

จากภาพประกอบ 4-6c ช่องสัญญาณที่ 2 เป็นการทำงานของ Task B (LED 2) เมื่องานที่ลำดับความสำคัญสูงกว่า Task B (Task A) ปล่อยหน่วยประมวลผลและในขณะเดียวกันกับที่ Task B กำลังรอเข้าใช้หน่วยประมวลผลก็จะเข้าทำงานในทันทีที่ Task A หยุดทำงาน

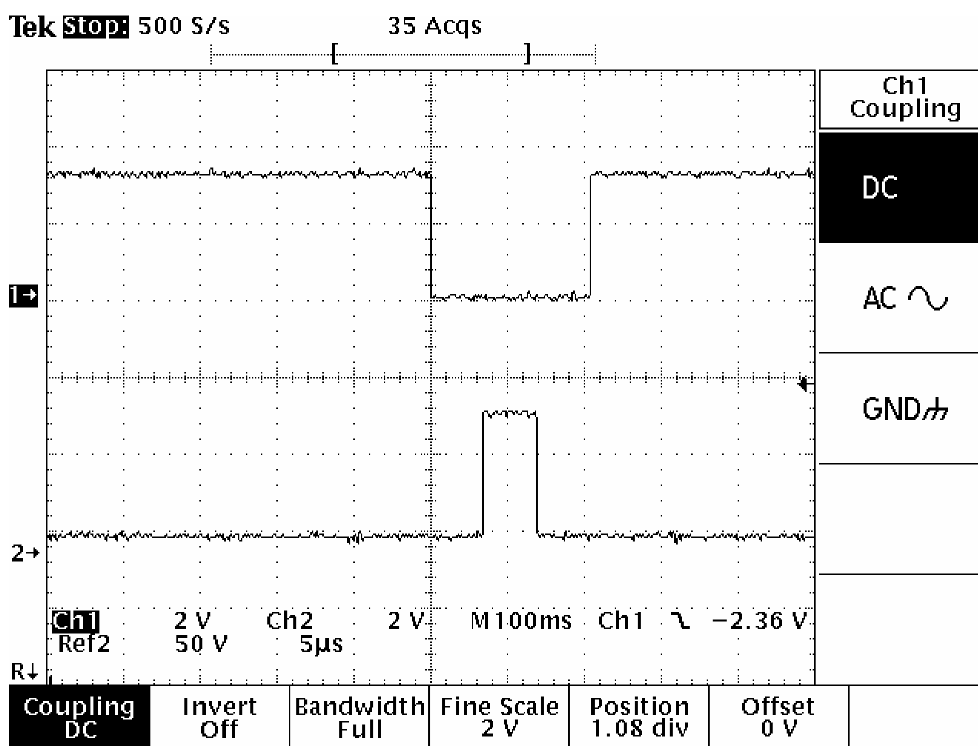
จากภาพประกอบ 4-6d ช่องสัญญาณที่ 2 เป็นการทำงานของ Task C (LED 3) เมื่องานที่มีลำดับความสำคัญสูงกว่า Task C (Task A และ Task B) ปล่อยหน่วยประมวลผลในเวลาเดียวกันและในขณะนั้น Task C กำลังรอทำงาน Task C จะได้เข้าใช้หน่วยประมวลผลทันที



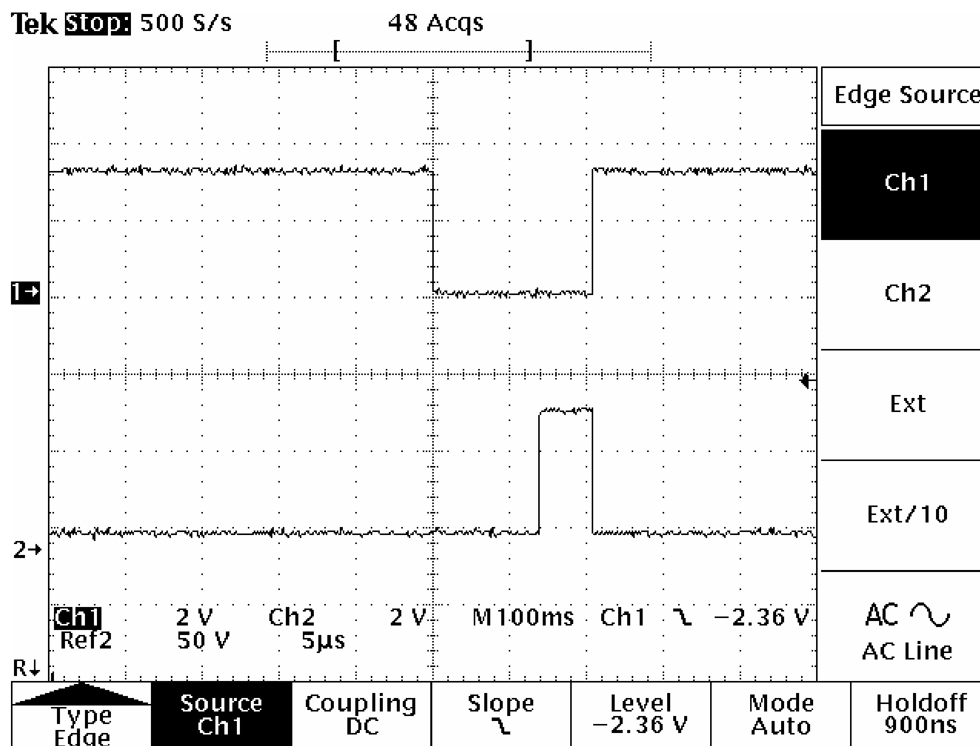
ภาพประกอบ 4-6a รอบของการเข้าใช้หน่วยประมวลผลโดยภาพรวมเมื่อเปรียบเทียบกับ idle task



ภาพประกอบ 4-6b ลำดับการเข้าทำงานของ TaskA เมื่อเปรียบเทียบกับ idle task



ภาพประกอบ 4-6c ลำดับการเข้าทำงานของ TaskB เมื่อเปรียบเทียบกับ idle task



ภาพประกอบ 4-6d ลำดับการเข้าทำงานของ TaskC เมื่อเปรียบเทียบกับ idle task

ภาพประกอบ 4-6 ลำดับการทำงานของระบบที่ตรวจสอบด้วย oscilloscope

เมื่อเปรียบเทียบการเข้าทำงานของทั้ง 3 งานด้วย oscilloscope พบว่าลำดับการเข้าทำงานคือ Task A, Task B, และ Task C ตามลำดับ และเมื่อทั้ง 3 งานปล่อยหน่วยประมวลผลในเวลาเดียวกัน ระบบจะจัดให้ idle task เข้าใช้หน่วยประมวลผลในทันที

4.1.1.3 การปรับปรุงระบบปฏิบัติการ uC/OS-II ให้ใช้กำลังงานต่ำลงจากคุณสมบัติการจัดตารางงาน

การจัดตารางงานของ uC/OS-II จัดตามลำดับความสำคัญของงานที่ได้กำหนดไว้ ระบบจะให้บริการงานที่มีความสำคัญสูงก่อน เมื่องานที่มีความสำคัญสูงหยุดพักงานที่มีความสำคัญรองลงมาก็จะได้รับบริการอย่างต่อเนื่องเรื่อยไปจนกระทั่งถึงช่วงเวลาหนึ่งที่ไม่มีงานใดต้องการเข้าใช้หน่วยประมวลผล idle task จะเข้าไปทำงานในช่วงเวลานั้น การทำงานของ idle task เป็นการทำงานที่สูญเสียกำลังงานโดยเปล่าประโยชน์ และนำไปสู่การเพิ่มความร้อนให้กับระบบ ดังนั้นจึงแก้ไขโปรแกรมเพิ่มเติมในส่วนของ idle task ให้ใช้กำลังงานลดลงโดยแทรกส่วน

ของโปรแกรมให้ระบบเข้าสู่โหมด/สภาวะควบคุมกำลังงานซึ่งประกอบด้วย 2 สภาวะ คือ power down และ idle การปรับให้ระบบอยู่ในสภาวะ power down และ idle ทำโดยการแทรกคำสั่งดังภาพประกอบ 4-7 และ 4-8 ตามลำดับซึ่งอยู่ในไฟล์ OS_CORE.C

```
void  OS_TaskIdle (void *parg)
{
#if OS_CRITICAL_METHOD == 3       /* Allocate storage for CPU status register  */
   OS_CPU_SR  cpu_sr;
   cpu_sr = 0;                     /* Prevent compiler warning                 */
#endif

   parg   = parg;                  /* Prevent compiler warning for not using 'parg'      */

   for (;;)
   {
      OS_ENTER_CRITICAL();
      OSIdleCtr++;
      OS_EXIT_CRITICAL();
      OSTaskIdleHook();            /* Call user definable HOOK              */
      LED_On(5);
      *((char *)0xe01fc0c0) = *((char*)0xe01fc0c0) | 0x02;   /*  power down  */
   }
}
```

ภาพประกอบ 4-7 คำสั่งสำหรับควบคุมสถานะของระบบให้เข้าสู่สถานะ power down

```
void  OS_TaskIdle (void *parg)
{
#if OS_CRITICAL_METHOD == 3       /* Allocate storage for CPU status register  */
   OS_CPU_SR  cpu_sr;
   cpu_sr = 0;                          /* Prevent compiler warning                          */
#endif

   parg   = parg;                         /* Prevent compiler warning for not using 'parg'      */

   for (;;)
   {
      OS_ENTER_CRITICAL();
      OSIdleCtr++;
      OS_EXIT_CRITICAL();
      OSTaskIdleHook();                      /* Call user definable HOOK                 */
      LED_On(5);
     *((char *)0xe01fc0c0) = *((char*)0xe01fc0c0) | 0x01;           /* idle */
   }
}
```

ภาพประกอบ 4-8 คำสั่งสำหรับควบคุมสถานะของระบบให้เข้าสู่สถานะ idle

การทำงานของระบบเมื่อเข้าสู่สภาวะ power down หรือ idle มีความแตกต่างกัน ดังนั้นปริมาณการใช้ไฟฟ้าย่อมต่างกันด้วย สำหรับการทดสอบเมื่อ idle task ทำงานจะไม่เข้าสู่สภาวะ power down หรือ idle ในทันที แต่ตรวจสอบเงื่อนไขในการกดสวิตช์ของบอร์ดก่อนถ้ามีการกดสวิตช์ก็จะยอมให้เข้าสู่สถานะที่ได้กำหนดไว้ เพื่อให้สามารถควบคุมการทำงานของระบบได้ พอร์ต 1.24 เป็นตำแหน่งที่ใช้สำหรับตรวจสอบสถานะของสวิตช์ด้วยเงื่อนไขดังภาพประกอบ 4-9

```
void  OS_TaskIdle (void *parg)
{
#if OS_CRITICAL_METHOD == 3       /* Allocate storage for CPU status register  */
   OS_CPU_SR  cpu_sr;
   cpu_sr = 0;                    /* Prevent compiler warning                  */
#endif

   parg   = parg;                 /* Prevent compiler warning for not using 'parg'      */

   for (;;)
   {
      OS_ENTER_CRITICAL();
      OSIdleCtr++;
      OS_EXIT_CRITICAL();
      if (!(IOPIN1 & 0x01000000))               /* check PORT1.24 */
          คำสั่งในการเปลี่ยนสถานะของระบบ   /* go to power down or idle mode */
```

ภาพประกอบ 4-9 การตรวจสอบการทำงานของพอร์ต 1.24 เพื่อเปลี่ยนสถานะของระบบให้เข้าสู่ power down หรือ idle

ทดสอบกำลังงานที่ระบบใช้ในสภาวะปกติ idle และ power down จากการวัดกระแสไฟฟ้าได้ผลดังตาราง 4-1

ตาราง 4-1 กระแสไฟฟ้าที่ระบบใช้ในสภาวะต่าง ๆ

สภาวะของระบบ	กระแสไฟฟ้า (mA)
ปกติ	58
Idle	22
Power down	1

ผลที่ได้จากการวัดกระแสไฟฟ้าที่ระบบใช้เมื่อเข้าสู่สภาวะแตกต่างกัน power down และ idle ใช้กระแสไฟฟ้าเพียงประมาณ 1.72% และประมาณ 37.93% ของกระแสในสภาวะปกติ ตามลำดับ

การตรวจสอบผลจากการทำงานเมื่อเข้าสู่สภาวะ idle ด้วย oscilloscope ทำได้โดยกำหนดให้ช่องสัญญาณที่ 1 เป็นการทำงานของ idle task และช่องสัญญาณที่ 2 เป็นสัญญาณเมื่อมีการกดสวิตช์ซึ่งต่อกับพอร์ต P1.24 การวิเคราะห์การทำงานของ idle task แสดงดังภาพประกอบ 4-10

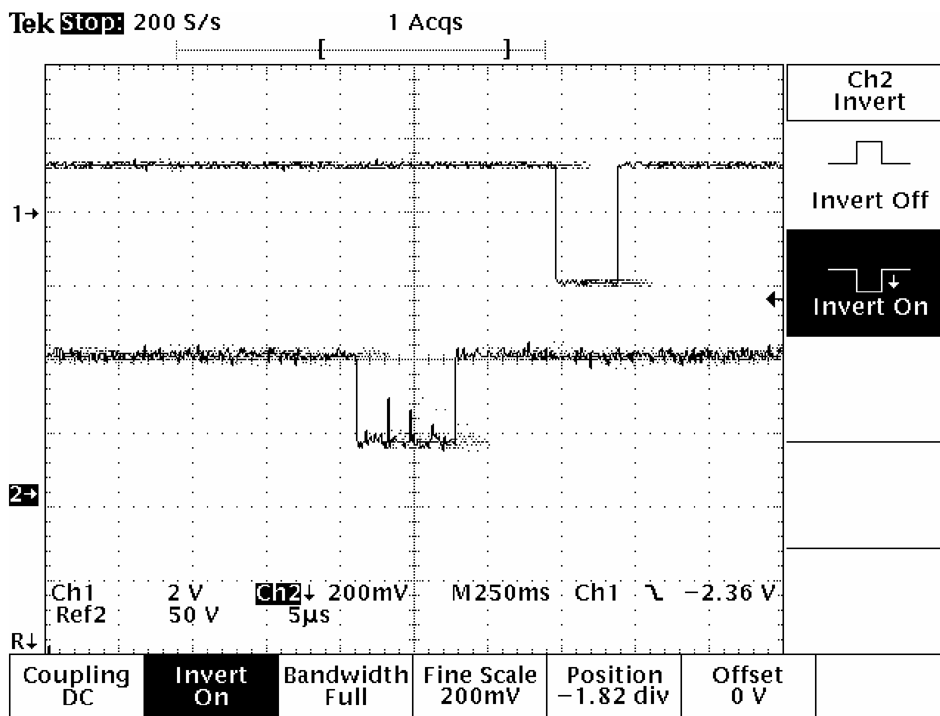
จากภาพประกอบ 4-10a ขณะที่ idle task กำลังทำงานอยู่เมื่อกดสวิตช์เพื่อให้ระบบเข้าสู่สถานะ idle พบว่าลักษณะกระแสไฟฟ้าที่ใช้ลดลงประมาณครึ่งหนึ่งจากสภาวะปกติแต่เมื่อปล่อยสวิตช์ลักษณะของกระแสไฟฟ้าที่ใช้จะเพิ่มขึ้นเป็นปกติ เมื่อพิจารณาภาพประกอบที่ 4-10b กรณีที่กดสวิตช์ค้างไว้พบว่ามีการใช้กระแสไฟฟ้าในลักษณะปกติเฉพาะช่วงเวลาที่ idle task หยุดทำงานเท่านั้น (งานอื่นเข้าใช้หน่วยประมวลผล) ช่วงเวลาที่ idle task ทำงานและเข้าสู่สภาวะ idle (มีการกดสวิตช์) สามารถลดกระแสไฟฟ้าได้ประมาณครึ่งหนึ่งจากสภาวะปกติ

เมื่อพิจารณาช่องสัญญาณที่ 2 ช่วงที่ระบบเข้าสู่สภาวะ idle พบว่าลักษณะกระแสไฟฟ้าที่ใช้ไม่คงที่ โดยกระแสไฟฟ้าเพิ่มสูงขึ้นเป็นช่วง ๆ และลดลงกลับมาที่ระดับเดิมเมื่อขยายอัตราส่วนในการแสดงผล (ภาพประกอบ 4-10c) เพื่อพิจารณาลักษณะดังกล่าวพบว่าช่วงเวลาการเพิ่มขึ้นของกระแสมีค่าคงที่ เมื่อศึกษาเพิ่มเติมค่าดังกล่าว คือ timer tick ซึ่งเป็น internal interrupt ที่เกิดขึ้นเป็นช่วงเวลาที่แน่นอน

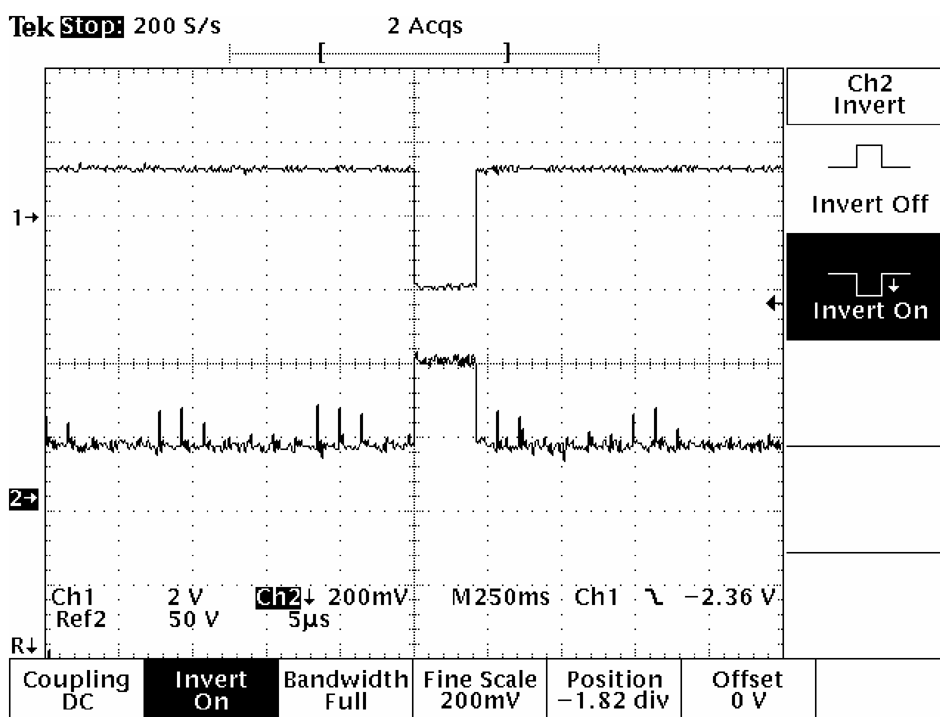
การตรวจสอบผลจากการทำงานเมื่อเข้าสู่สภาวะ power down ด้วย oscilloscope ทำได้โดยกำหนดให้ช่องสัญญาณที่ 1 เป็นสัญญาณการทำงานของ idle task และช่องสัญญาณที่ 2 เป็นสัญญาณเมื่อมีการกดสวิตช์ซึ่งต่อกับพอร์ต P1.24 การวิเคราะห์การทำงานของ idle task แสดงดังภาพประกอบ 4-11

จากภาพประกอบ 4-11 เมื่อมีการกดสวิตช์เพื่อให้ระบบเข้าสู่สถานะ power down พบว่าลักษณะการใช้กระแสไฟฟ้ามีค่าเข้าใกล้ศูนย์หรือฐานของช่องสัญญาณที่ 2

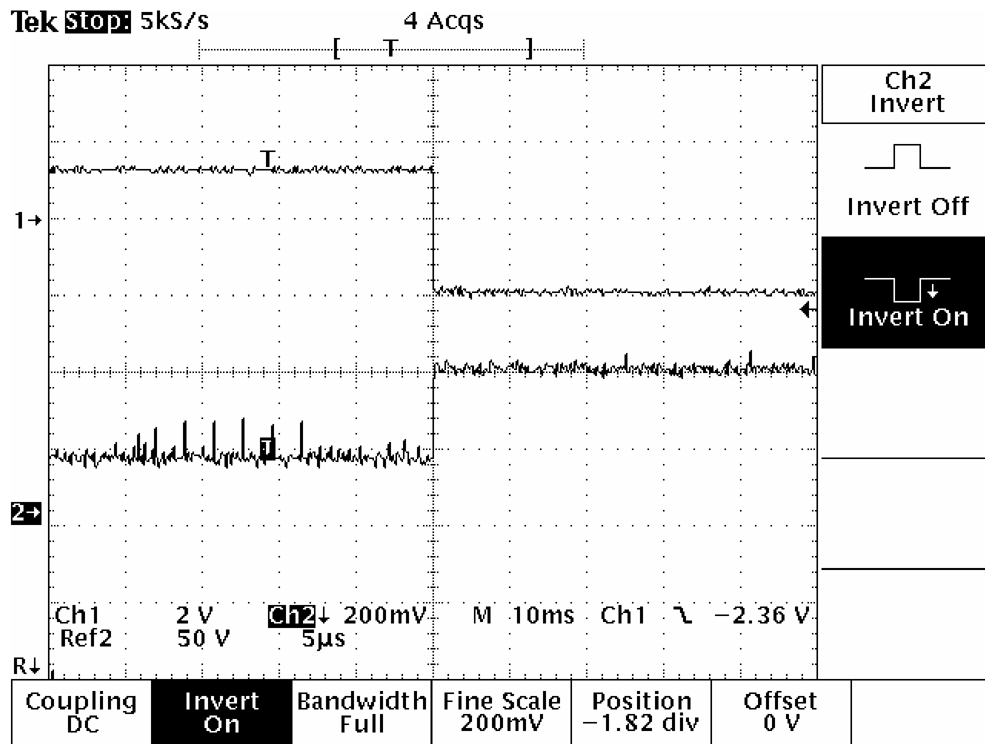
สรุปได้ว่าเมื่อระบบเข้าสู่สภาวะ idle มีกิจกรรมหนึ่งเกิดขึ้นในช่วงเวลาที่สั้นมากและมีความถี่คงที่ กิจกรรมดังกล่าวก็คือการทำงานของ timer tick ที่เป็น internal interrupt ซึ่งเป็นปัจจัยที่ทำให้เกิดการสูญเสียกำลังงานได้ดังที่ได้กล่าวไว้ในบทที่ 3 และจะทำการทดสอบและวิเคราะห์กำลังงานที่ใช้อันเนื่องมาจากสาเหตุดังกล่าวในหัวข้อ 4.1.1.4 Timer tick กับ uC/OS-II



ภาพประกอบ 4-10a การลดลงของกระแสไฟฟ้าเมื่อระบบเข้าสู่สถานะ idle (กรณีกดสวิตช์แล้วปล่อย)

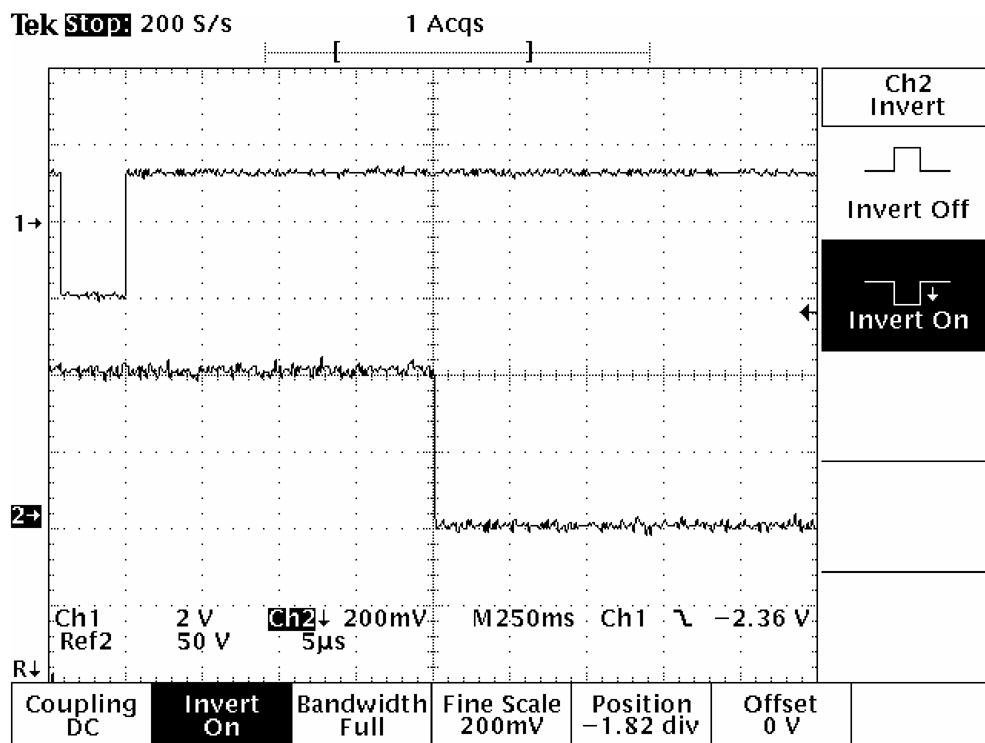


ภาพประกอบ 4-10b การลดลงของกระแสไฟฟ้าเมื่อระบบเข้าสู่สถานะ idle (กรณีกดสวิตช์ค้าง)



ภาพประกอบ 4-10c การขยายอัตราส่วนของรูปในขณะที่ระบบเข้าสู่สภาวะ idle

ภาพประกอบ 4-10 ลักษณะกระแสไฟฟ้าที่ใช้เมื่อระบบเข้าสู่สถานะ idle



ภาพประกอบ 4-11 การทำงานเมื่อเข้าสู่สภาวะ power down

ระบบที่เข้าสู่สภาวะ power down ใช้กำลังงานน้อยมากใกล้เคียงศูนย์ มีเพียง external interrupt เท่านั้นที่สามารถเปลี่ยนให้ระบบกลับมาทำงานได้ปกติซึ่งจะกล่าวต่อไปในหัวข้อ 4.1.1.6 สถานะ Power down กับ External interrupt

4.1.1.4 Timer tick กับ uC/OS-II

Timer tick เป็น internal interrupt ที่เกิดขึ้นเองตามช่วงเวลาด้วยความถี่คงที่ ค่า timer tick ที่ระบบกำหนดคือ 100 tick/s และค่าที่กำหนดเป็นตัวนับให้เพิ่มค่าจนกระทั่งถึงค่าดังกล่าวเพื่อระบุเวลาในการ tick แต่ละรอบคือ 70,000 ตัวนับกำหนดไว้ที่ T0MR0 ในไฟล์ BSP.C และความถี่ของการ tick ใน 1 วินาที กำหนดไว้ที่ OS_TICK_PER_SEC ในไฟล์ os_cfg.h ส่วนของการโปรแกรมแสดงดังภาพประกอบ 4-12 เพื่อความละเอียดในการทดสอบจึงแปลงความถี่ของการ tick ใน 1 ms มีวิธีการดังนี้

1 tick นับ 70,000 ครั้ง

100 tick นับ 7,000,000 ครั้ง

ในเวลา 1 วินาที ตัวนับต้องนับถึง 7,000,000 ครั้ง ใน 100 tick หาจำนวนครั้งของการนับใน 1 ms โดยเทียบดังนี้

1 วินาที จำนวนครั้งของการนับ 7,000,000

1/1,000 วินาที จำนวนครั้งของการนับ 7,000,000/1,000 = 7,000

ในเวลา 1 ms ตัวนับต้องนับถึง 7,000 ครั้ง หาความถี่ของการ tick ใน 1 ms ได้ดังนี้

นับ 7,000,000 ครั้ง มีความถี่ในการ tick 100 tick/s

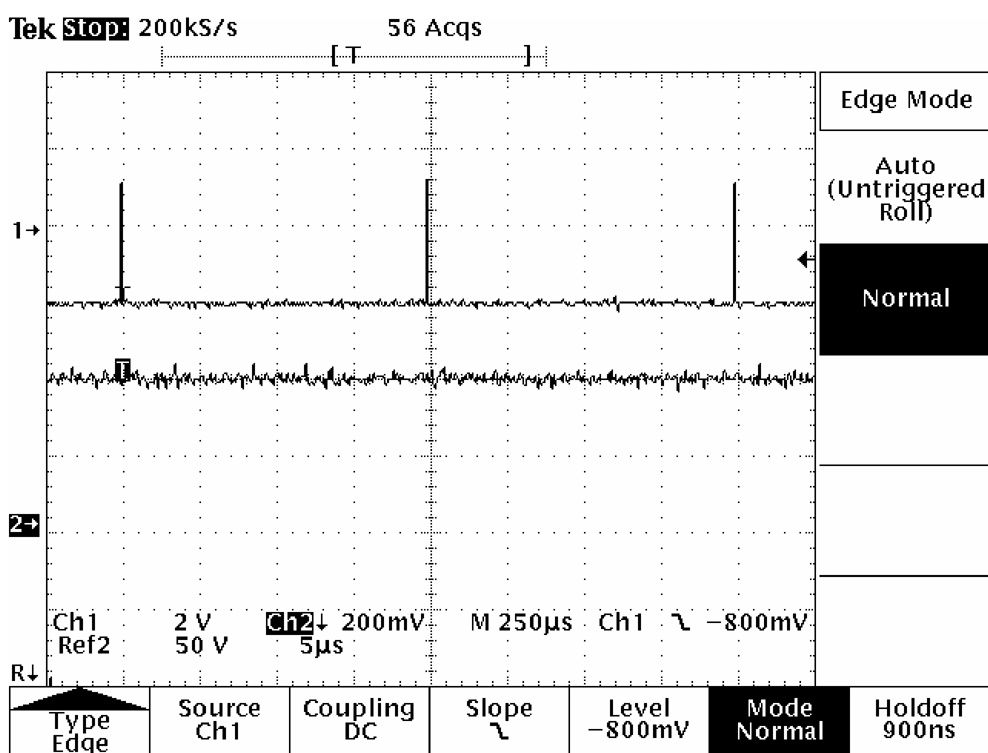
นับ 7,000 ครั้ง มีความถี่ในการ tick $\frac{7{,}000 \times 100}{7{,}000{,}000} = 0.1$ tick/s

การตรวจสอบความถูกต้องในการคำนวณทำได้โดยการวัดความถี่ของการ tick ใน 1 ms ดังภาพประกอบ 4-13 โดยช่องสัญญาณที่ 1 เป็นสัญญาณของ timer tick ที่เกิดขึ้นในแต่ละช่วงเวลาและช่องสัญญาณที่ 2 เป็นสัญญาณของกระแสไฟฟ้าที่ใช้

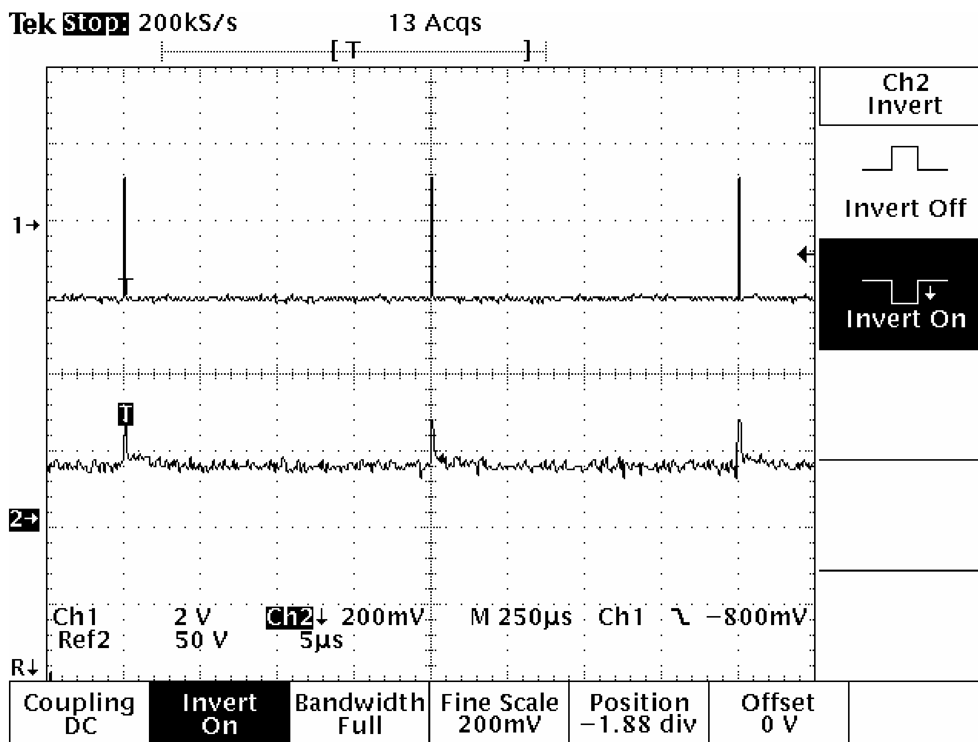
```
T0MR0        = 70000L;                    /* Count up to this value.          */
#define OS_TICKS_PER_SEC   100      /* Set the number of ticks in one second */
```

ภาพประกอบ 4-12 ส่วนของโปรแกรมในการกำหนดความถี่ของ timer tick

จากภาพประกอบ 4-13 เห็นได้ว่าในช่องสัญญาณที่ 1 เกิด timer tick ทุก ๆ 1 ms ตรงตามที่ได้คำนวณไว้ ในสภาวะปกติใช้กระแสไฟฟ้า 58 mA และในสภาวะ idle ใช้กระแสไฟฟ้า 22 mA



ภาพประกอบ 4-13a การเกิด timer tick ในเวลา 1 ms และกระแสไฟฟ้าที่ใช้เมื่อระบบทำงานในสภาวะปกติ



ภาพประกอบ 4-13b ลักษณะการเกิด timer tick ในเวลา 1 ms และกระแสไฟฟ้าที่ใช้เมื่อระบบเข้าสู่สถานะ idle

ภาพประกอบ 4-13 การวัดความถี่และกระแสไฟฟ้าที่ใช้เมื่อมีการ tick ใน 1 ms ด้วย oscilloscope

4.1.1.5 การทดสอบผลจากการปรับค่าความถี่ของ timer tick ต่อการใช้กระแสไฟฟ้า

การปรับความถี่ของ timer tick ต้องปรับค่า 2 ค่าคือ T0MR0 และ OS_TICKS_PER_SEC โดยปรับให้ความถี่ของ timer tick มีค่าลดลง 10 (T0MR0 เพิ่มขึ้น 10 เท่า) เพื่อไม่ให้กระทบกับค่าในฟังก์ชัน OSTimeDly ในไฟล์ OS_TIME.c จึงต้องปรับ OS_TICKS_PER_SEC เพิ่มขึ้น 10 เท่า โปรแกรมดังภาพประกอบ 4-14 วัดกระแสไฟฟ้าที่ใช้และตรวจสอบความถี่ของ timer tick ด้วย oscilloscope พบว่าไม่สามารถตรวจสอบได้ด้วย oscilloscope เนื่องจากช่วงของความถี่มีค่ากว้างมาก ผลจากการวัดกระแสไฟฟ้าที่ใช้ คือ ในสภาวะปกติใช้กระแสไฟฟ้า 58 mA และในสภาวะ idle ใช้กระแสไฟฟ้า 22 mA ดังนั้นสรุปได้ว่าการปรับให้ความถี่ของ timer tick มีค่าต่ำมาก ๆ หรือต่ำกว่า 1 ms ไม่มีผลต่อกระแสไฟฟ้าที่ใช้

```
T0MR0        = 700000L;                  /* Count up to this value.          */
#define OS_TICKS_PER_SEC   10            /* Set the number of ticks in one second */
```

ภาพประกอบ 4-14 การโปรแกรมความถี่เพิ่มขึ้น 10 เท่าจาก 1 ms เป็น 100 us

ทำการวิเคราะห์และตรวจสอบผลการเปลี่ยน timer tick ให้มีความถี่สูงขึ้นกับกระแสไฟฟ้าที่ใช้โดยปรับให้ความถี่เพิ่มขึ้น 10 เท่าจาก 1 ms ให้เป็น 100 us การแก้ไขโปรแกรมแสดงดังภาพประกอบ 4-15

```
T0MR0        = 7000L;                    /* Count up to this value.          */
#define OS_TICKS_PER_SEC   1000          /* Set the number of ticks in one second */
```

ภาพประกอบ 4-15 ส่วนของโปรแกรมที่มีการแก้ไขความถี่ของ timer tick เพิ่มขึ้น 10 เท่าจาก 1 ms เป็น 100 us

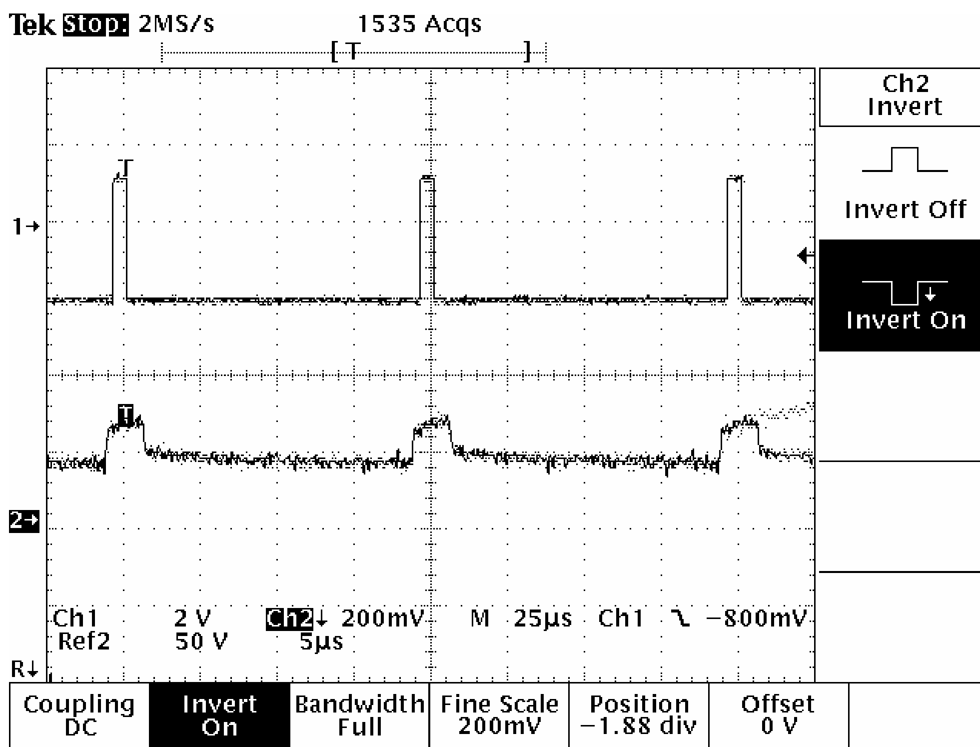
ตรวจสอบผลจากการปรับ timer tick ด้วย oscilloscope ดังภาพประกอบ 4-16 และวัดกระแสไฟฟ้าที่ใช้ ผลจากการวัดกระแสเมื่อระบบทำงานที่สถานะปกติและ idle คือ 58 mA และ 25 mA ตามลำดับ

วิเคราะห์และตรวจสอบผลการเปลี่ยน timer tick ให้มีความถี่สูงขึ้นกับกระแสไฟฟ้าที่ใช้โดยปรับให้ความถี่เพิ่มขึ้น 2 เท่าจาก 100 us ให้เป็น 50 us การแก้ไขโปรแกรมแสดงดังภาพประกอบ 4-17

ตรวจสอบผลจากการปรับ timer tick ด้วย oscilloscope ดังภาพประกอบ 4-18 และวัดกระแสไฟฟ้าที่ใช้ ผลจากการวัดกระแสเมื่อระบบทำงานที่สถานะปกติและ idle คือ 58 mA และ 30 mA ตามลำดับ

จากภาพประกอบ 4.18 ในช่องสัญญาณที่ 1 (การทำงานของ internal interrupt ที่เกิดจาก timer tick) พบว่าช่วงเวลาของการให้บริการ interrupt หรือช่วงเวลาของการเข้าสู่ Interrupt Service Routine (ISR) มีค่าเท่ากับ 0.5 us ซึ่งมีค่าน้อยมากเมื่อเปรียบเทียบกับเวลาในการทำงานในแต่ละรอบ (ความถี่) ของ timer tick

วิเคราะห์และตรวจสอบผลการเปลี่ยน timer tick ให้มีความถี่สูงขึ้นกับกระแสไฟฟ้าที่ใช้โดยปรับให้ความถี่เพิ่มขึ้น 5 เท่าจาก 50 us ให้เป็น 10 us การแก้ไขโปรแกรมแสดงดังภาพประกอบ 4-19



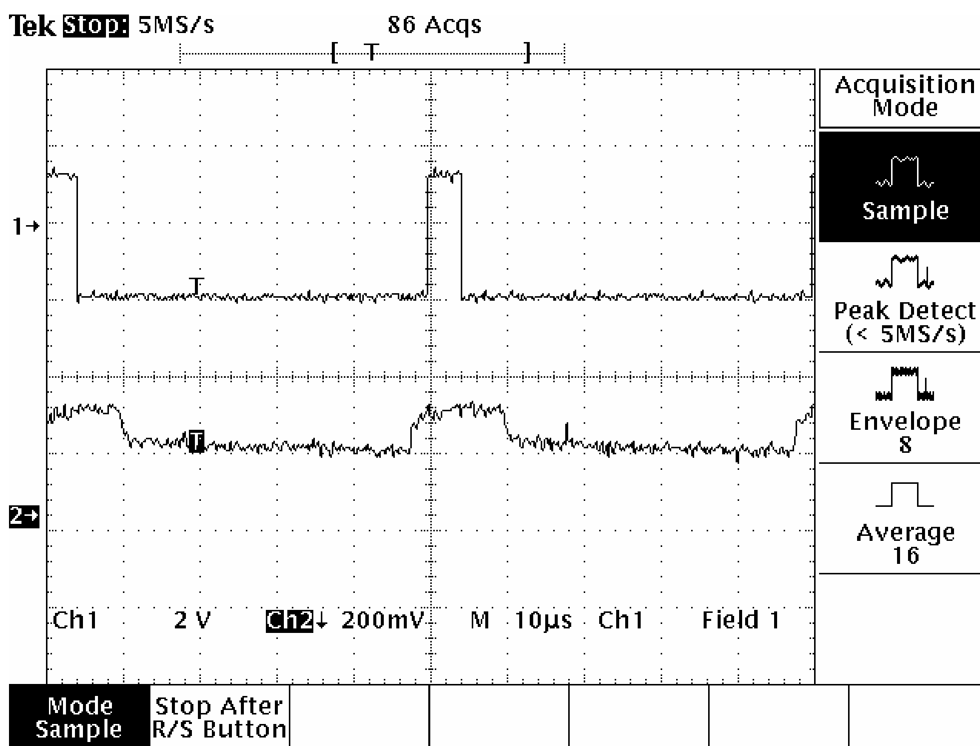
ภาพประกอบ 4-16 ลักษณะการเกิด timer tick และกระแสไฟฟ้าที่ใช้ในสถานะ idle จาก oscilloscope เมื่อปรับความถี่เพิ่มขึ้น 10 เท่าจาก 1 ms เป็น 100 us

```
T0MR0        = 3500L;                    /* Count up to this value.          */
#define OS_TICKS_PER_SEC   2000          /* Set the number of ticks in one second */
```

ภาพประกอบ 4-17 ส่วนของโปรแกรมที่มีการแก้ไขความถี่ของ timer tick เพิ่มขึ้น 2 เท่าจาก 100 us เป็น 50 us

ตรวจสอบผลจากการปรับ timer tick ด้วย oscilloscope และวัดกระแสไฟฟ้าที่ใช้ ผลจากการวัดกระแสเมื่อระบบทำงานที่สถานะปกติและ idle มีค่าเป็น 58 mA ทั้ง 2 สถานะ

นั่นคือ เกิด timer tick บ่อยมากจนกระทั่งระบบไม่ได้รับ idle task (มีการปรับให้ระบบเข้าสู่สถานะ idle) ส่งผลให้การทำงานใช้กระแสไฟฟ้าเต็มที่ตลอดเวลา



ภาพประกอบ 4-18 ลักษณะการเกิด timer tick และกระแสไฟฟ้าที่ใช้ในสถานะ idle จาก oscilloscope เมื่อปรับ ความถี่เพิ่มขึ้น 2 เท่าจาก 100 us เป็น 50 us

```
T0MR0         = 700L;                    /* Count up to this value.          */
#define OS_TICKS_PER_SEC   10000   /* Set the number of ticks in one second */
```

ภาพประกอบ 4-19 ส่วนของโปรแกรมที่มีการแก้ไขความถี่ของ timer tick เพิ่มขึ้น 5 เท่าจาก 50 us เป็น 10 us

จากการปรับโปรแกรมเพื่อเปลี่ยนแปลงค่า timer tick ด้วยความถี่ต่าง ๆ กันและวัดกระแสไฟฟ้าที่ใช้สรุปได้ดังตาราง 4-2

ตาราง 4-2 ความถี่ในการเกิด timer tick กับกระแสไฟฟ้าที่ใช้ในการทำงานปกติกับสภาวะ idle

ความถี่ของ timer tick	T0MR0	OS_TICKS_PER_SEC	กระแสไฟฟ้าที่ใช้ (mA)	
			Idle	ปกติ
10 ms	700,000	10	22	58
1 ms	70,000	100	22	58
100 us	7,000	1,000	25	58
50 us	3,500	2,000	30	58
10 us	700	10,000	58	58

จากการทดสอบกระแสไฟฟ้าที่ใช้กับความถี่ของ timer tick ค่าต่าง ๆ สรุปได้ว่า ความถี่ของ timer tick มีผลต่อกระแสไฟฟ้าที่ใช้นั่นคือ ถ้า timer tick มีความถี่สูงจะส่งผลให้ระบบใช้ปริมาณกระแสไฟฟ้าสูงขึ้นด้วย สำหรับความถี่ที่ใช้กระแสไฟฟ้าต่ำที่สุด (22 mA) คือ 10 ms และ 1 ms ในการพิจารณาเลือกความถี่ที่เหมาะสมจะเลือกใช้ความถี่ที่มีค่าสูงกว่าเพื่อให้ระบบสามารถตอบสนองได้ทันเชิงเวลาจริงดังนั้น ความถี่ของ timer tick ที่เหมาะสมและช่วยตระหนักถึงกำลังงานที่ใช้ได้คือ 1 ms

4.1.1.6 สถานะ Power down กับ External interrupt

จากการทดสอบกระแสไฟฟ้าที่ใช้เมื่อระบบเข้าสู่สถานะที่แตกต่างกัน สถานะ power down ใช้กระแสไฟฟ้าน้อยที่สุดเกือบเข้าใกล้ศูนย์แต่ระบบที่เข้าสู่สภาวะนี้ไม่สามารถให้บริการงานต่าง ๆ ได้ จึงต้องเปลี่ยนสถานะให้กลับเข้าสู่การทำงานปกติเพื่อรอรับงานที่เข้ามาใช้บริการ การเปลี่ยนสถานะจาก power down เข้าสู่สถานะปกติทำได้วิธีเดียวเท่านั้นคือ การใช้ external interrupt สำหรับการพัฒนา external interrupt เพื่อทดสอบการทำงานของระบบที่เข้าสู่สถานะ power down ประกอบด้วยการสร้าง external interrupt จากการกดสวิตช์ และการสร้าง external interrupt จากเครื่องไมโครคอมพิวเตอร์

การสร้าง external interrupt จากการกดสวิตช์

ฟังก์ชันหลักสำหรับการควบคุมการทำงานของ external interrupt ที่ผู้วิจัยพัฒนาเพิ่มเติมในระบบปฏิบัติการ uC/OS-II คือ EINT0_Init และ EINT0_ISR_Handler โดยการแทรกโปรแกรมในไฟล์ BSP.C โดยฟังก์ชัน EINT0_Init มีหน้าที่กำหนดค่าเบื้องต้นสำหรับ external interrupt และถูกเรียกใช้เมื่อมีการกำหนดค่าเริ่มต้นของบอร์ดหรือมีการเรียกฟังก์ชัน BSP_Init ซึ่งถูกเรียกใช้ครั้งแรกก่อนที่งานทุกงานจะเริ่มต้นทำงานแบบ multitasking ฟังก์ชัน EINT0_ISR_Handler มีหน้าที่ควบคุมการทำงานของโปรแกรมเมื่อมี external interrupt เกิดขึ้นตรงกับตำแหน่งที่ได้กำหนดไว้ในเบื้องต้น

ในการพัฒนาโปรแกรมจัดการ interrupt ต้องกำหนดค่าเบื้องต้นให้ตรงตามการทำงานของ Vectored Interrupt Controller ซึ่งประกอบด้วย 4 ส่วนหลักที่สำคัญในการพัฒนาโปรแกรมแสดงในภาพประกอบ 4-20 โดยรายละเอียดหน้าที่การทำงานของรีจิสเตอร์แต่ละส่วนมีดังนี้

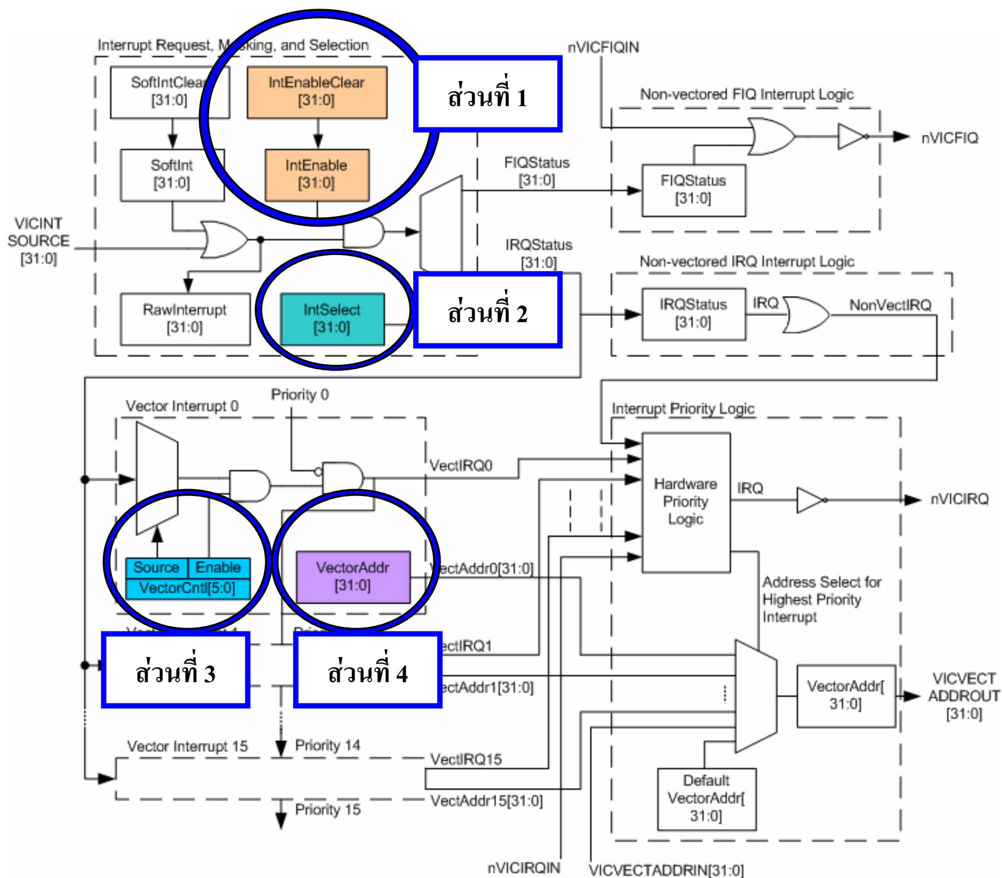
ส่วนที่ 1 IntEnableClear และ IntEnable มีหน้าที่เคลียร์ค่าและ enable interrupt ที่ต้องการ

ส่วนที่ 2 IntSelect มีหน้าที่เลือกชนิดของ interrupt ให้ทำงานที่ FIQStatus หรือ IRQStatus

ส่วนที่ 3 VectorCntl มีหน้าที่กำหนดลำดับความสำคัญของ interrupt ที่อยู่ในตำแหน่งของ IRQ ทั้ง 16 slot คือ 0-15 โดย 0 มีความสำคัญสูงสุดและ 15 มีความสำคัญต่ำที่สุด

ส่วนที่ 4 VectorAddr มีหน้าที่กำหนดตำแหน่ง Interrupt Service Routine (ISR) ของ IRQ ทั้ง 16 slot

External interrupt ที่สามารถเลือกใช้เพื่อควบคุมการทำงานของระบบแบบไม่เฉพาะเจาะจงหรือขึ้นอยู่กับการพัฒนาโปรแกรมสามารถเลือกใช้ได้ 4 ตัว คือ EINT0, EINT1, EINT2 และ EINT3 ทั้ง 4 ตัวมีคุณสมบัติเหมือนกัน การกำหนดค่าเบื้องต้นในการพัฒนาโปรแกรมสำหรับ external interrupt ทั้ง 4 ส่วนดังที่ได้กล่าวมาแล้วนั้นกำหนดไว้ในฟังก์ชัน EINT0_Init ส่วนของโปรแกรมสำหรับการกำหนดค่าทั้ง 4 ส่วนแสดงดังภาพประกอบ 4-21



ภาพประกอบ 4-20 รีจิสเตอร์หลักสำหรับพัฒนา interrupt

```
VICIntEnable  =  (1 << VIC_EINT0);    /* Enable Interrupts                    (1)  */
 VICIntSelect &= ~(1 << VIC_EINT0); /* Enable interrupts                      (2)  */
 VICVectCntl1  = 0x20 | VIC_EINT0;    /* Enable vectored interrupts           (3)  */
 VICVectAddr1  = (INT32U)EINT0_ISR_Handler; /* Set the vector address(4)*/
```

ภาพประกอบ 4-21 ส่วนของโปรแกรมในการกำหนดค่าเริ่มต้นของ External interrupt

การกำหนดขาของ ETNT0 เลือกได้ 2 ขา คือ P0.1 และ P0.16 เนื่องจาก P0.1 ได้ถูกกำหนดให้เป็น external interrupt สำหรับส่งข้อมูลเอาท์พุทสำหรับรีจิสเตอร์ UART0 ซึ่งเป็นหน้าที่ที่ P0.1 รองรับการทำงาน ดังนั้นขา P0.16 จึงถูกเลือกมาใช้ในการพัฒนาโปรแกรม

สำหรับ external interrupt การกำหนดขา การปรับให้หน่วยประมวลผลออกจาก power down และการเคลียร์ค่า EINT0 โดยส่วนของโปรแกรมที่เพิ่มเติมอยู่ในฟังก์ชัน EINT0_Init (ภาพประกอบ 4-22)

```
PINSEL1 = PINSEL1 | 0x00000001;   /* select P0.16 as EINT0 */

/* waken up from power down mode by External Interrupt0 */
*((char *)0xE01FC144) = *((char *)0xE01FC144) | 0x000001;

/* External Interrupt Flag */
*((char *)0xE01FC140) = *((char *)0xE01FC140) | 0x000001;
```

ภาพประกอบ 4-22 ส่วนของโปรแกรมการกำหนดค่าการทำงานของ External interrupt

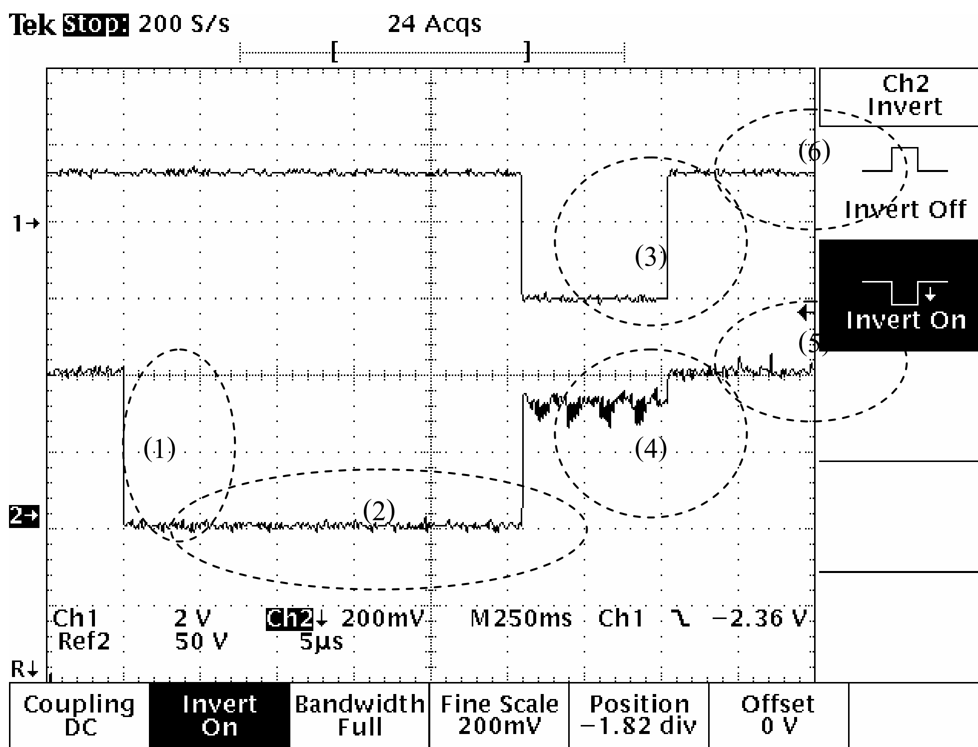
EINT0_ISR_Handler เป็นฟังก์ชันที่ถูกเรียกใช้เมื่อเกิด external interrupt และกำหนดค่าให้กับรีจิสเตอร์ VICVectAddr เพื่อระบุตำแหน่งที่หน่วยประมวลผลต้องไปทำงานเมื่อเกิด external interrupt การทำงานของฟังก์ชันนี้แสดงดังภาพประกอบ 4-23

```
BoardSetup();
Init_A2D();
init_serial0();

/* clear External Interrupt Flag */
*((char *)0xE01FC140) = *((char *)0xE01FC140) | 0x000001;
VICVectAddr = 0;
```

ภาพประกอบ 4-23 การเข้าสู่ Interrupt Service Routine (ISR)

การให้ระบบออกจากสถานะ power down เข้าสู่สภาวะการทำงานปกติด้วยการกดสวิตช์ผลที่ได้จากการทำงานแสดงดังภาพประกอบ 4-24 โดยช่องสัญญาณที่ 1 คือสัญญาณ external interrupt และช่องสัญญาณที่ 2 คือสัญญาณของกระแสไฟฟ้าที่ใช้



ภาพประกอบ 4-24 การเปลี่ยนสถานะจาก power down เข้าสู่การทำงานปกติด้วย external interrupt จากการกดสวิตช์ซึ่งตรวจสอบจาก oscilloscope

ลักษณะสัญญาณของ oscilloscope ในภาพประกอบ 4-24 มีรายละเอียดดังนี้

(1) มีการกดสวิตช์เพื่อให้ระบบเข้าสู่สถานะ power down ลักษณะของกระแสไฟฟ้าจึงลดต่ำลง

(2) ระบบเข้าสู่สถานะ power down ปริมาณกระแสไฟฟ้าลดต่ำลง

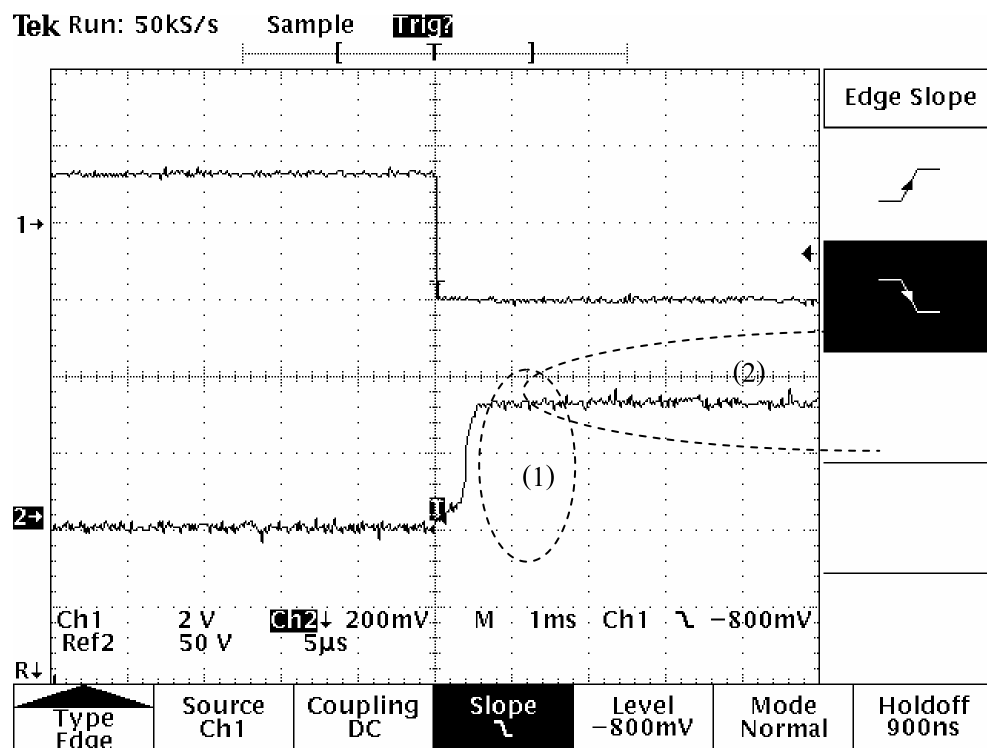
(3) ช่วงที่มีการกดสวิตช์ที่เชื่อมต่อกับขา P0.16 เพื่อทำให้เกิด external interrupt

(4) ลักษณะของกระแสไฟฟ้าที่ใช้ที่เพิ่มสูงขึ้นจากสภาวะก่อนหน้าเพื่อให้บริการงาน interrupt

(5) ลักษณะการใช้กระแสไฟฟ้าสูงขึ้นหลังจากที่ระบบให้บริการงาน interrupt เสร็จสิ้น

(6) เมื่อไม่มีสัญญาณ external interrupt หน่วยประมวลผลกลับมาทำงานตามปกติ (เปลี่ยนจาก power down เข้าสู่การทำงานปกติ)

จากการขยายอัตราส่วนในการแสดงผลของ oscilloscope พบว่าเมื่อมีการกดสวิตช์เพื่อให้เกิด external interrupt ระบบจะไม่ได้ให้บริการ interrupt ในทันทีจะใช้เวลาช่วงหนึ่งก่อนที่จะให้บริการ interrupt ได้ ดังนั้นการที่ระบบเข้าสู่สถานะ power down และกลับเข้าเข้าสู่การทำงานปกติจะสูญเสียเวลาส่วนหนึ่ง (ประมาณ 0.5 ms) คือ เวลาที่ใช้ก่อนให้บริการ interrupt และเวลาอันเนื่องจากการทำงานของ interrupt ดังภาพประกอบ 4-25 ซึ่งเวลาที่ใช้ในส่วนนี้อาจส่งผลกระทบต่อการทำงานเชิงเวลาจริงได้



ภาพประกอบ 4-25 ช่วงเวลาที่ใช้ก่อนที่ระบบสามารถทำงานได้ตามปกติโดยตรวจสอบด้วย oscilloscope

ลักษณะสัญญาณของ oscilloscope ในภาพประกอบ 4-25 มีรายละเอียดดังนี้

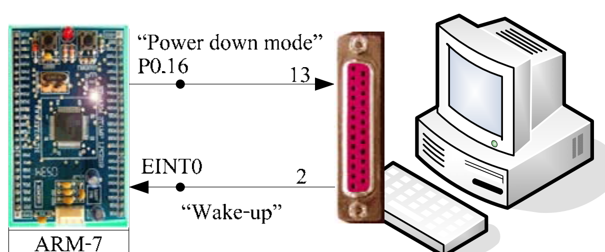
(1) ช่วงเวลาที่ใช้ก่อนการให้บริการ interrupt

(2) ช่วงเวลาที่ใช้ในการให้บริการงาน interrupt

จากการทดสอบการออกจากสถานะ power down ในข้างต้นระบบสามารถออกจากสถานะดังกล่าวได้ด้วยการกดสวิตช์ที่เชื่อมต่อกับพอร์ต P0.16 การพัฒนาและทดสอบระบบในแต่ละครั้งไม่สามารถกำหนดช่วงเวลาที่แน่นอนในการส่งสัญญาณที่เป็น external interrupt ได้ เพราะเป็น interrupt ที่เกิดจากการกดสวิตช์ของผู้วิจัย ดังนั้นจึงนำไปสู่การพัฒนาโปรแกรมบนเครื่องไมโครคอมพิวเตอร์ที่สามารถปรับเปลี่ยนค่า delay เพื่อส่งสัญญาณ external interrupt จากเครื่องไมโครคอมพิวเตอร์ผ่าน parallel port หลังจากรับทราบการเข้าสู่สถานะ power down ของ ARM-7

การสร้าง external interrupt จากเครื่องไมโครคอมพิวเตอร์

การสร้าง external interrupt จากเครื่องไมโครคอมพิวเตอร์ประกอบด้วย 2 ส่วน คือ ARM-7 และเครื่องไมโครคอมพิวเตอร์ดังภาพประกอบ 4-26 การปรับค่าหน่วงเวลา (delay) เพื่อส่งสัญญาณ external interrupt ให้กับ ARM-7 ทำบนเครื่องไมโครคอมพิวเตอร์ ก่อนที่ ARM-7 จะเข้าสู่สถานะ power down จะส่งสัญญาณ “Power down mode” ให้กับเครื่องไมโครคอมพิวเตอร์ผ่านทาง P0.16 เข้าสู่ขา 13 ของ printer port เมื่อไมโครคอมพิวเตอร์รับรู้การเข้าสู่สถานะ power down ของ ARM-7 จะรอตามช่วงระยะเวลาที่ได้กำหนดไว้ก่อนส่งสัญญาณ “Wake-up” มาให้ ARM-7 ผ่านทางขา 2 ของ printer port หลังจากที่ ARM-7 ได้รับสัญญาณปลุกจากเครื่องไมโครคอมพิวเตอร์ก็สามารถเข้าสู่การทำงานปกติได้



ภาพประกอบ 4-26 การใช้ไมโครคอมพิวเตอร์ระยะเวลาในการปลุก ARM-7

การพัฒนาโปรแกรมเพื่อส่งสัญญาณที่ทำหน้าที่เป็น external interrupt จากเครื่องไมโครคอมพิวเตอร์ผ่าน parallel port พัฒนาด้วยภาษาซี ใช้ Turbo C เป็นตัวแปลภาษาและทำงานภายใต้ระบบปฏิบัติการ DOS 6.02 เมื่อโปรแกรมตรวจสอบพบว่าบอร์ดเข้าสู่สภาวะ power down จะส่งสัญญาณที่ทำหน้าที่เป็น external interrupt เพื่อเปลี่ยนสถานะในการทำงานของบอร์ดแต่จะส่งสัญญาณหลังจากครบกำหนดของ delay ดัง pseudo code ในภาพประกอบ 4-27 และผลการทดสอบการใช้กระแสไฟฟ้าด้วย delay ค่าต่าง ๆ แสดงดังตาราง 4-3

1. รอสัญญาณ idle mode ที่ส่งมาจาก ARM-7
2. Delay ช่วงระยะเวลาหนึ่งก่อนที่จะส่งสัญญาณออกผ่าน printer port
3. ส่งสัญญาณออกผ่าน printer port ซึ่งเปรียบเสมือนการกดสวิตช์เพื่อเป็นสัญญาณ interrupt ให้กับ ARM-7
4. รอสัญญาณที่ ARM-7 อยู่ในสถานการณ์ทำงานปกติกลับมาเพื่อส่งสัญญาณเคลียร์ค่า interrupt กลับไปยัง ARM-7
5. ส่งสัญญาณเคลียร์ค่า interrupt
6. กลับไปทำงานที่ข้อ 1.

ภาพประกอบ 4-27 ขั้นตอนวิธีสำหรับส่ง external interrupt จากเครื่องไมโครคอมพิวเตอร์ผ่าน parallel port

ตาราง 4-3 การใช้กระแสไฟฟ้าเมื่อมีการปรับค่า delay ด้วยค่าต่าง ๆ ในระบบที่มีการปรับให้เข้าสู่สถานะ power down และเปรียบเทียบการใช้กระแสไฟฟ้ากับระบบที่มีการทำงานแบบปกติ

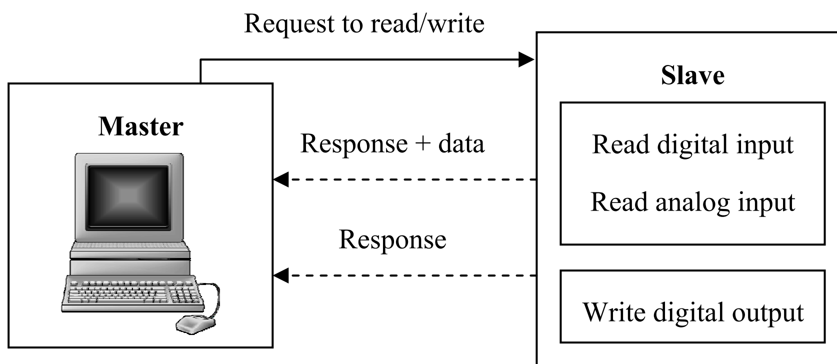
ค่า Delay (us)	**กระแสสภาวะปกติ (mA)**	**กระแสสภาวะ Power down (mA)**
38000	58	0
6800	58	0
3700	58	1
975	58	2
390	58	4
12.5	58	11

4.1.2 การพัฒนาแอพลิเคชันสำหรับทดสอบระบบเชิงเวลาจริง

ในหัวข้อนี้จะกล่าวถึงแอพลิเคชันที่พัฒนาเพื่อทดสอบการทำงานของระบบเชิงเวลาจริงซึ่งประกอบด้วย 2 ส่วน คือ แอพลิเคชันที่ทำงานบนระบบเชิงเวลาจริงและแอพลิเคชันสำหรับทดสอบความถูกต้องในการทำงานของระบบที่พัฒนาขึ้น

4.1.2.1 การพัฒนาแอพลิเคชันที่ทำงานบนระบบเชิงเวลาจริง

แอพลิเคชันที่พัฒนาบนระบบเชิงเวลาจริง คือ อุปกรณ์ควบคุมที่ใช้โพรโตคอล Modbus ทำหน้าที่เป็น slave ทำงานบนระบบปฏิบัติการเชิงเวลาจริง uC/OS-II และรันบนบอร์ด ARM-7 Modbus slave ที่พัฒนาบน uC/OS-II จะทำงานตามที่ master ร้องขอ ได้แก่ การส่งค่าสถานะของ holding register ของ analog input 4 ตัว digital input 4 ตัว และเขียนค่า digital output 4 ตัว ตามตำแหน่งที่ master ต้องการ การทำงานของโปรแกรมแสดงดังภาพประกอบ 4-28

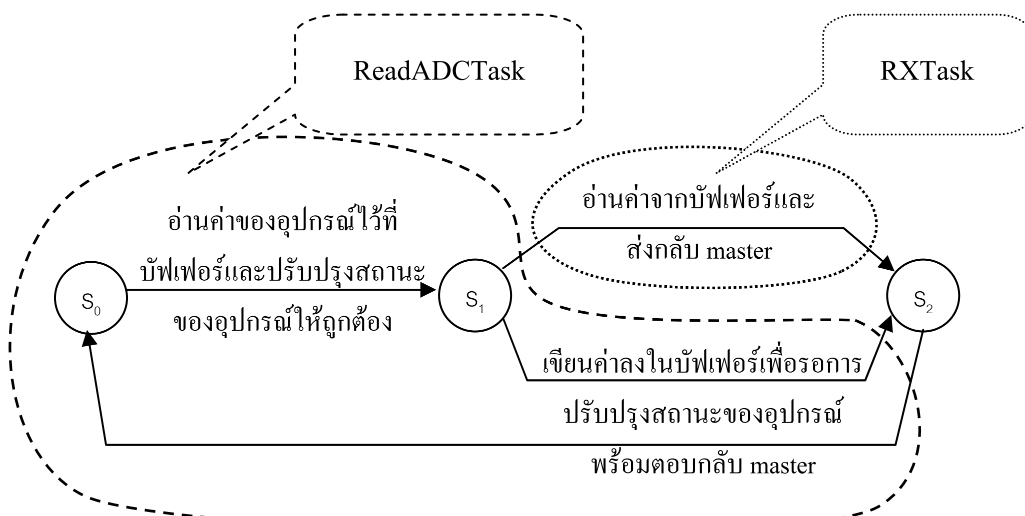


ภาพประกอบ 4-28 การทำงานโดยรวมของ Modbus slave

การทำงานของ Modbus slave ดังที่ได้กล่าวมาข้างต้นเมื่อนำมาพัฒนาโปรแกรมแบบ multitasking แบ่งได้ 2 งาน คือ งานที่มีหน้าที่รับ/ส่งข้อมูลในการถามตอบระหว่าง master และ slave ผ่าน serial port (RXTask) และอีกงานหนึ่งมีหน้าที่เขียนค่า digital output ตามที่ master ร้องขอพร้อมทั้งเก็บข้อมูลสถานะของอุปกรณ์บน slave (ReadADCTask) งานแรกนำค่าเหล่านี้มาใช้ในการส่งข้อมูลเพื่อตอบกลับไปยัง master กรณีที่ master ถามสถานะของอุปกรณ์ต่าง

ๆ ของ slave โดยการทำงานของงานในโปรแกรมแสดงดัง state diagram ในภาพประกอบ 4-29 ส่วนรายละเอียดการโปรแกรมอยู่ในภาคผนวก ง

สำหรับการกำหนดลำดับความสำคัญของงานพิจารณาจากคุณสมบัติของระบบเชิงเวลาจริงดังนั้นการทำงานต้องตอบสนองต่อเวลาได้อย่างถูกต้องภายในระยะเวลาซึ่งถูกกำหนดโดย master นั่นคือ เมื่อ slave ได้รับการร้องขอต้องตอบสนองให้ทันตามเวลาดังกล่าว ดังนั้นจึงกำหนดให้ RXTask มีลำดับความสำคัญสูงสุดเพื่อสนองตอบต่อความต้องการของระบบ แต่อย่างไรก็ตาม ReadADCTask ก็ยังมีโอกาสทำงานแม้ว่าจะมีลำดับความสำคัญต่ำกว่าเพราะสามารถเข้าทำงานได้ในช่วงที่ไม่มีการร้องขอจาก master การกำหนดลำดับความสำคัญของงานทั้งสองและ stack ที่ใช้แสดงดังภาพประกอบ 4-30



ภาพประกอบ 4-29 state diagram การทำงานของโปรแกรม Modbus slave

ReadADCTask : งานสำหรับอ่านและเขียนค่าสถานะของอุปกรณ์

RXTask : งานสำหรับรับคำร้องขอจาก master และทำงานตามที่ต้องการพร้อมตอบกลับ master (RXTask)

```
#define  RX_TASK_STK_SIZE     64
#define  RX_TASK_START_PRIO    0
#define ReadADC_TASK_STK_SIZE 64
#define ReadADC_TASK_START_PRIO 1
```

ภาพประกอบ 4-30 การกำหนดขนาด stack และลำดับความสำคัญของงาน RXTask และ ReadADCTask

การติดต่อสื่อสารกับโฮสต์ (host) ใช้ UART0 เป็นรีจิสเตอร์สำหรับรับข้อมูลจาก master เก็บข้อมูลไว้ที่รีจิสเตอร์ U0RBR จนครบตามจำนวนไบต์ที่กำหนดไว้ในรีจิสเตอร์ U0FCR หลังจากนั้นจะส่งสัญญาณ interrupt เพื่อส่งข้อมูลทั้งหมดใน U0RBR ไปประมวลผล การส่งข้อมูลของ master จะส่งอย่างต่อเนื่องขนาด 8 ไบต์ ถ้าส่งด้วยความเร็วต่ำการประมวลผลทางฝั่ง slave สามารถทำงานได้อย่างถูกต้องแต่กรณีที่ master ส่งข้อมูลร้องขอด้วยความเร็วสูงจะส่งผลให้การประมวลผลผิดพลาดเนื่องจากฝั่งรับไม่สามารถรับข้อมูลได้ทัน เพราะเกิดการ interrupt เมื่อรับข้อมูลทุก ๆ 1 ไบต์ทำให้ส่งผลกระทบกับการทำงานของหน่วยประมวลผลโดยภาพรวม ดังนั้นจึงต้องโปรแกรมขนาดของ FIFO ให้ U0RBR รับข้อมูลจำนวน 8 ไบต์ก่อนทำการ interrupt เพื่อส่งข้อมูลไปประมวลผล การกำหนดขนาดของ FIFO แสดงดังภาพประกอบ 4-31

```
U0FCR |= 0x80; /*  set  8  bytes  of  UART0  FIFO  Control  Register  */
```

ภาพประกอบ 4-31 การกำหนด flag ของ U0FCR เพื่อควบคุมให้ U0RBR รับข้อมูลให้ครบ 8 ไบต์ก่อนเกิดการ interrupt เพื่อส่งข้อมูล

4.1.2.2 การพัฒนาแอพลิเคชันสำหรับทดสอบความถูกต้องในการทำงานของระบบ

แอพลิเคชันสำหรับทดสอบความถูกต้องในการทำงานของระบบเชิงเวลาจริงต้องสามารถทดสอบความถูกต้องของคำตอบและเวลาที่ใช้ในการทำงานได้ แอพลิเคชันทั่วไปที่ทำหน้าที่เป็น master เช่น comDebug หรือ ModScan ไม่สามารถทดสอบความถูกต้องเชิงเวลาจริง

ได้ ดังนั้นจึงต้องพัฒนาแอพลิเคชันที่ทำหน้าที่เป็น master สำหรับทดสอบระบบ และนำโปรแกรมที่มีอยู่แล้วใช้ตรวจสอบความถูกต้องในการทำงานของแอพลิเคชันที่พัฒนาขึ้นมาก่อนนำมาใช้ในการทดสอบจริง

แอพลิเคชันสำหรับทดสอบระบบเชิงเวลาจริงพัฒนาด้วย Microsoft Visual Basic 6.0 เนื่องจากง่ายต่อการสร้าง user interface และมีเครื่องมือสำหรับติดต่อสื่อสารผ่าน serial port โปรแกรมที่พัฒนาเพื่อทดสอบระบบมีขั้นตอนการทำงานดังนี้

1) กำหนดค่าเริ่มต้นสำหรับการทำงาน ได้แก่ พอร์ตและค่าเริ่มต้นที่ใช้ในการรับ/ส่งข้อมูล

2) ส่งแพ็คเก็ต (packet) เพื่อถามค่าของ Holding register ไปยัง slave ด้วยความถี่ที่ได้กำหนดไว้ในส่วนของ Transmit interval พร้อมกับเพิ่มจำนวน Count Tx รูปแบบของข้อมูลที่ส่งเป็นดังนี้

Chr(1) & Chr(3) & Chr(0) & Chr(2) & Chr(0) & Chr(1) & Chr(&H25) & Chr(&HCA)

Chr(1) : ข้อมูลไบต์แรกซึ่งเป็นการติดต่อกับ slave ตำแหน่งที่ 1

Chr(3) : ฟังก์ชันสำหรับอ่านค่าจาก Holding register ของ slave ตำแหน่งที่ 1

Chr(0) & Chr(2) : ตำแหน่งเริ่มต้นของ slave ตำแหน่งที่ 1 ที่ต้องการอ่าน Holding register

Chr(0) & Chr(1) : จำนวนตำแหน่งของข้อมูลที่ต้องการอ่านค่า Holding register จาก slave ตำแหน่งที่ 1 โดยนับจากตำแหน่งเริ่มต้น (ตำแหน่ง 2) ซึ่งมีค่าเป็น 1

Chr(&H25) & Chr(&HCA) : ค่า CRC ที่ได้จากการคำนวณจากข้อมูลข้างต้นทั้ง 6 ไบต์

3) รับข้อมูลตอบกลับจาก slave

3.1) กรณีไม่มีความผิดพลาดเนื่องจากเวลาตอบกลับ (Response time)

- ตรวจสอบความถูกต้องของข้อมูลด้วยการคำนวณ CRC และเปรียบเทียบกับ CRC ที่ได้รับถ้าไม่ตรงกันให้เพิ่มค่า Data error ซึ่งเป็นความผิดพลาดที่เกิดจากข้อมูลที่ได้รับไม่ถูกต้อง

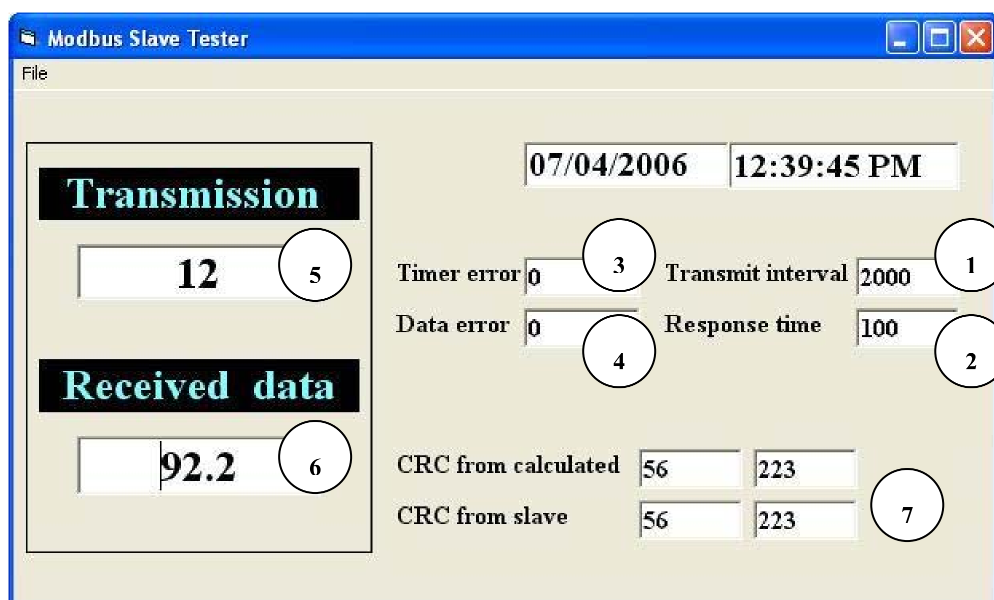
- ถ้าข้อมูลถูกต้องแสดงข้อมูลที่ได้รับในส่วนของ Data

3.2) กรณีที่เกิดความผิดพลาดเนื่องจากเวลาตอบกลับ

- เพิ่มค่า Time out error

หน้าจอของโปรแกรมที่ใช้ติดต่อกับผู้ใช้สำหรับทดสอบการทำงานเชิงเวลาจริง แสดงดังภาพประกอบ 4-32 โดยหมายเลขที่กำหนดเป็นรายละเอียดของโปรแกรมซึ่งมีคุณสมบัติดังนี้

1) สามารถกำหนดช่วงเวลาความถี่สำหรับการส่งข้อมูลไปยัง slave ได้
2) สามารถกำหนด response time จากการตอบกลับของ slave ได้
3) สามารถตรวจสอบความถูกต้องของเวลาที่ใช้อันเนื่องมาจาก slave ไม่สามารถตอบสนองได้ทันตาม response time หรือ time out นับจำนวนครั้งที่มีความผิดพลาดเกิดขึ้นและแสดงผล
4) สามารถตรวจสอบความถูกต้องของข้อมูลที่ได้รับโดยการคำนวณและเปรียบเทียบค่า CRC นับจำนวนครั้งที่มีความผิดพลาดเกิดขึ้นและแสดงผล
5) สามารถนับจำนวนครั้งในการส่งข้อมูลร้องขอและแสดงผล
6) แสดงผลข้อมูลที่ได้รับจากการร้องขอ
7) แสดงค่า CRC ที่ได้รับทั้ง 2 ไบต์ และค่า CRC ที่ได้จากการคำนวณเพื่อตรวจสอบและยืนยันความถูกต้องของข้อมูล



ภาพประกอบ 4-32 โปรแกรมที่พัฒนาสำหรับทดสอบระบบเชิงเวลาจริง

4.2 การทดสอบระบบเชิงเวลาจริงที่ตระหนักถึงกำลังงานที่ใช้

การทดสอบระบบเชิงเวลาจริงที่ตระหนักถึงกำลังงานที่ใช้มีปัจจัยหลักที่ต้องพิจารณา 3 อย่าง คือ ความถูกต้องในการทำงานของเวลา ความถูกต้องของข้อมูลที่ได้รับ และกำลังงานที่ระบบใช้ในการทำงาน

จากผลการวัดกระแสไฟฟ้าดังที่ได้กล่าวไว้ในส่วนของการพัฒนาระบบ power down มีการใช้กระแสไฟฟ้าน้อยที่สุดแต่ต้องมี external interrupt เพื่อเปลี่ยนสถานะในการทำงานดังนั้นจากการพัฒนา external interrupt ที่ส่งมาจากเครื่องไมโครคอมพิวเตอร์โดยผ่าน parallel port นำมาใช้ทดสอบความน่าเชื่อถือของระบบและกระแสไฟฟ้าที่ใช้ด้วยการปรับการหน่วงเวลาของสัญญาณ external interrupt และตรวจสอบการทำงานของระบบในการตอบสนองเชิงเวลาจริงเมื่อปรับให้ระบบเข้าสู่สถานะ power down ผลการทดสอบแสดงดังตาราง 4-4

ตาราง 4-4 ผลการทำงานของระบบและกระแสไฟฟ้าที่ใช้ที่มีการปรับ delay หรือความถี่ของ external interrupt

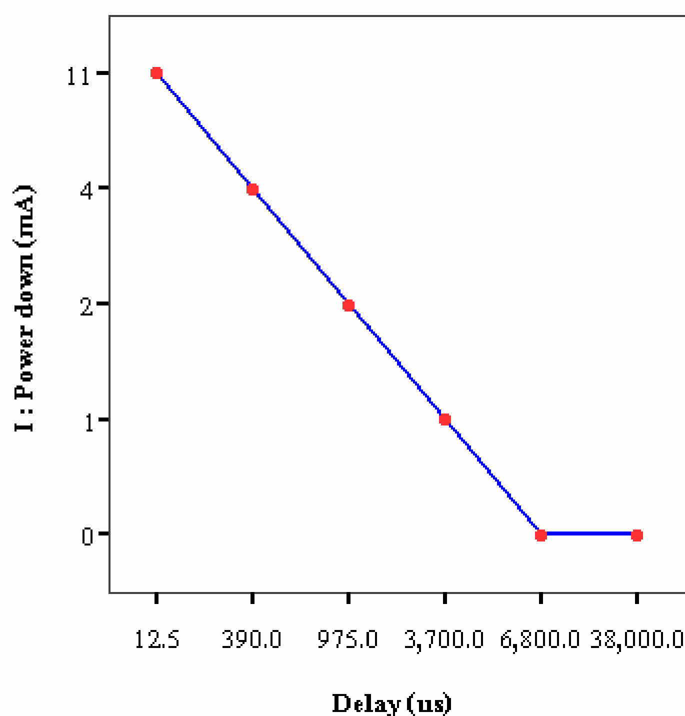
Delay of external interrupt (us)	Normal		Power down	
	กระแส(mA)	สถานะระบบ	กระแส(mA)	สถานะระบบ
38000	58	ปกติ	1	ปกติ
6800	58	ปกติ	1	ปกติ
3700	58	ปกติ	1	ปกติ
975	58	ปกติ	2	ปกติ
390	58	ปกติ	4	ปกติ
12.5	58	ปกติ	11	หยุดการทำงาน

Note : สถานะหยุดการทำงานหมายถึง สถานะที่ระบบไม่สามารถทำงานต่อได้หรือทำงานผิดปกติ (hang)

ค่าที่ได้จากการทดสอบเมื่อระบบเข้าสู่สถานะ power down ในตาราง 4-4 เขียนกราฟแสดงความสัมพันธ์ระหว่างกระแสไฟฟ้าที่ใช้ (mA) กับระยะเวลาที่ระบบเข้าสู่สถานะ power down (us) ได้ดังภาพประกอบ 4-33 ส่วนระบบที่อยู่ในสถานะปกติมีการใช้กระแสไฟฟ้าคงที่

จากกราฟพบว่าเมื่อเพิ่มค่า delay กระแสไฟฟ้าที่ใช้เมื่ออยู่ในสถานะ power down ก็จะลดลง แต่เมื่อ delay ลดลงถึง 12.5 us ระบบไม่สามารถทำงานต่อได้ การที่ระบบเข้าสู่สถานะ power down จะมีความต้องการในใช้กระแสไฟฟ้าที่ต่ำมากแต่ไม่สนับสนุนการทำงานเชิงเวลาจริงที่ต้องการ response time ที่ต่ำมากหรือระบบที่ไม่ยอมให้มีความผิดพลาดเกิดขึ้น โดยเฉพาะอย่างยิ่งระบบ hard real- time หรือระบบที่มีการเข้าใช้หน่วยประมวลผลบ่อยครั้ง

สภาวะ idle สามารถลดกำลังงานที่ใช้ได้ จึงปรับระบบให้เข้าสู่สภาวะ idle เมื่อไม่มีการประมวลผลและทดสอบการทำงานเชิงเวลาจริงด้วยการทดสอบความถูกต้องในขณะรับ/ส่งข้อมูลด้วยโพรโตคอล Modbus ในขั้นตอนแรกมีวัตถุประสงค์เพื่อตรวจสอบ response time ที่ต่ำที่สุดที่การทำงานของระบบสามารถรับได้ และพิจารณาถึงความแตกต่างในการใช้กระแสไฟฟ้าเมื่อมีการรับส่งข้อมูลในสถานะปกติและ idle



ภาพประกอบ 4-33 ความสัมพันธ์ระหว่างกระแสไฟฟ้าที่ใช้ (mA) กับระยะเวลาที่ระบบเข้าสู่สถานะ power down (us)

การหา response time ที่ต่ำที่สุดที่ระบบสามารถรับได้สามารถนำไปใช้ทดสอบในขั้นตอนอื่น ๆ ได้การตั้ง response time ไม่ควรต่ำกว่าค่าที่ระบบสามารถรับได้เพราะจะทำให้

การรับส่งข้อมูลมีความผิดพลาดเกิดขึ้น อีกทั้งเป็นการยืนยัน response time ที่ได้จากการคำนวณ และสามารถประมาณเวลาการประมวลผลของ ARM-7 ได้

การทดสอบหา response time ทดสอบโดยส่งข้อมูลจากโปรแกรมบนเครื่องไมโครคอมพิวเตอร์ซึ่งพัฒนาด้วยโปรแกรม Microsoft Visual Basic 6.0 ดังที่ได้กล่าวไว้แล้ว กำหนดค่าความถี่ในการส่งข้อมูล และ response time โปรแกรมจะแสดงข้อมูลที่ได้รับ ความผิดพลาดที่เกิดขึ้น และจำนวนชุดข้อมูลที่ได้ส่งไปยัง slave ความผิดพลาดที่เกิดขึ้นพิจารณา 2 ส่วน คือ ความผิดพลาดที่เกิดจากการรับส่งเนื่องจากเกินเวลาที่กำหนด และความผิดพลาดของข้อมูล ซึ่งความผิดพลาดของข้อมูลจะเกิดในกรณีที่มีการรับส่งสมบูรณ์แต่ข้อมูลที่ได้รับไม่ถูกต้องเท่านั้น ถ้ามีความผิดพลาดที่เกิดจากการรับส่งจะไม่มีความผิดพลาดเนื่องจากข้อมูลเพราะข้อมูลที่ไปถึงปลายทางไม่ถูกต้องสมบูรณ์ ดังนั้นจึงไม่มีการส่งข้อมูลจาก slave ตอบกลับมา

การทดสอบหา response time ที่เหมาะสม ขั้นแรกกำหนดให้ใช้ความถี่ต่ำในการส่งข้อมูลเพื่อไม่ให้เกิดความผิดพลาดในการส่งและพิจารณาเฉพาะความผิดพลาดจากข้อมูลที่ได้รับมาเท่านั้น กำหนดให้ความถี่ในการส่งข้อมูลแต่ละชุดมีค่าเป็น 2000 ms ใช้เวลาในการทดสอบ 3 นาที ดังนั้นจำนวนเฟรมข้อมูลที่ส่งสำหรับการทดสอบในแต่ละครั้งเท่ากับ 90 เฟรม ผลการทดสอบแสดงดังตาราง 4-5

ตาราง 4-5 ความผิดพลาดที่เกิดขึ้นเมื่อกำหนดค่า response time ที่ต่างกัน

Response Time (ms)	I (mA)	Time Out (Times)	Data Err (Times)
10	58	90	0
15	58	90	0
16	58	14	0
17	58	0	0

จากผลการทดสอบหา response time ที่ไม่ทำให้ระบบทำงานผิดพลาด คือ 17 ms ดังนั้นสำหรับการทดสอบในครั้งต่อไปต้องกำหนดค่า response time สูงกว่า 17 ms ทุกครั้ง

เวลาที่ใช้สำหรับการส่งข้อมูลไปกลับของระบบที่ทำการทดสอบประกอบด้วยเวลาในการส่งข้อมูลไปกลับ และเวลาในการประมวลผล ข้อมูลที่ส่งไปร้องขอจาก master มีขนาด 8 ไบต์ และข้อมูลที่ส่งกลับมามีขนาด 7 ไบต์ ภายใน 1 ไบต์ประกอบด้วย 2 stop bit, 1

start bit และข้อมูล 8 บิต ดังนั้น 1 ไบต์ประกอบด้วยข้อมูลขนาด 11 บิต ความเร็วในการรับส่งข้อมูล คือ 9,600 บิตต่อวินาที ดังนั้นเวลาที่ใช้สำหรับรับส่งข้อมูลสามารถคำนวณได้ดังนี้

ส่งข้อมูล 8 ไบต์ ใช้เวลา $\frac{11 \times 8}{9600} \approx 9ms$

รับข้อมูล 7 ไบต์ ใช้เวลา $\frac{11 \times 7}{9600} \approx 8ms$

เวลาสำหรับรับส่งข้อมูล คือ 9 + 8 = 17 ms

จากการทดสอบเพื่อหา response time ที่ต่ำที่สุดโดยไม่ทำให้ระบบทำงานผิดพลาด คือ 17 ms ดังนั้นสรุปได้ว่าในการประมวลผลเมื่อเทียบกับเวลาที่ใช้ในการรับส่งข้อมูลใช้เวลาน้อยมาก

จากการทดสอบการทำงานเชิงเวลาจริงของระบบในสภาวะปกติและ idle โดยกำหนดให้ส่งข้อมูลทุก ๆ 40 ms และ response time มีค่า 20 ms ผลการทดสอบเป็นเวลา 10 นาทีแสดงดังตาราง 4-6

ตาราง 4-6 กระแสไฟฟ้าที่ใช้และความผิดพลาดที่เกิดขึ้นเมื่อระบบอยู่ในสภาวะปกติและ idle

	I Before sending (mA)	**I (mA)**	**Time Out (Times)**	**Data Err (Times)**
Normal	57	58	0	0
Idle	21	22	0	0

จากผลที่ได้ในตาราง 4-6 สรุปได้ว่าการทำงานที่สภาวะ idle เมื่อเปรียบเทียบกับสภาวะปกติสามารถลดกระแสไฟฟ้าเหลือเพียง 22 mA หรือ 37.93% โดยไม่ส่งผลกระทบต่อการทำงานเชิงเวลาจริงของระบบ นั่นคือไม่มีความผิดพลาดใด ๆ เกิดขึ้นในระหว่างการทำงานและสามารถลดกระแสไฟฟ้าที่ใช้ได้

จากผลการทดสอบในข้างต้นสรุปได้ว่าการทำงานที่สภาวะ idle สามารถตอบสนองเชิงเวลาจริงได้ดีและสามารถลดปริมาณกำลังงานที่ใช้ได้ ส่วนการทำงานที่ปรับให้เข้าสู่สภาวะ power down สามารถลดปริมาณกำลังที่ใช้ลงได้อย่างมากแต่ส่งผลกระทบต่อการตอบสนองเชิงเวลาจริงเนื่องจากขณะที่ระบบเข้าสู่สถานะ power down อุปกรณ์ต่าง ๆ จะถูกตัดไฟทำให้ไม่สามารถให้บริการงานใด ๆ ได้ในช่วงเวลาดังกล่าว