TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

I2C

Inter-Integrated Circuit (I2C) Module

강사
: Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
2 차 발표 (2018. 07.12~18)

◎함수

```
void i2cInit(void);
void i2cSetOwnAdd(i2cBASE_t *i2c, uint32 oadd);
void i2cSetSlaveAdd(i2cBASE_t *i2c, uint32 sadd);
void i2cSetBaudrate(i2cBASE_t *i2c, uint32 baud);
uint32  i2cIsTxReady(i2cBASE_t *i2c);
void i2cSendByte(i2cBASE_t *i2c, uint8 byte);
void i2cSend(i2cBASE_t *i2c, uint32 length, uint8 * data);
uint32  i2cIsRxReady(i2cBASE_t *i2c);
uint32  i2cIsStopDetected(i2cBASE_t *i2c);
void i2cClearSCD(i2cBASE_t *i2c);
uint32  i2cRxError(i2cBASE_t *i2c);
uint8   i2cReceiveByte(i2cBASE_t *i2c);
void i2cReceive(i2cBASE_t *i2c, uint32 length, uint8 * data);
void i2cEnableNotification(i2cBASE_t *i2c, uint32 flags);
void i2cDisableNotification(i2cBASE_t *i2c, uint32 flags);
void i2cSetStart(i2cBASE_t *i2c);
void i2cSetStop(i2cBASE_t *i2c);
void i2cSetCount(i2cBASE_t *i2c ,uint32 cnt);
void i2cEnableLoopback(i2cBASE_t *i2c);
void i2cDisableLoopback(i2cBASE_t *i2c);
void i2cSetMode(i2cBASE_t *i2c, uint32 mode);
void i2cSetDirection(i2cBASE_t *i2c, uint32 dir);
bool i2cIsMasterReady(i2cBASE_t *i2c);
bool i2cIsBusBusy(i2cBASE_t *i2c);
void i2c2GetConfigValue(i2c_config_reg_t *config_reg,
config_value_type_t type);

void i2cInit(void)
{
  /** - i2c Enter reset */
  i2cREG2->MDR = (uint32)((uint32)0U << 5U);

  /** - set i2c mode */
  i2cREG2->MDR =   (uint32)((uint32)0U << 15U)     /* nack mode  */
        | (uint32)((uint32)0U << 14U)    /* free running */
        | (uint32)((uint32)0U << 13U)    /* start condition - master mode only */
        | (uint32)((uint32)1U <<11U)     /* stop condition   */
        | (uint32)((uint32)1U <<10U)     /* Master/Slave mode  */
        | (uint32)((uint32)I2C_TRANSMITTER)     /* Transmitter/receiver */
        | (uint32)((uint32)I2C_7BIT_AMODE)     /* Expanded address */
        | (uint32)((uint32)0 << 7U)      /* repeat mode */
        | (uint32)((uint32)0U << 6U)     /* digital loopback */
        | (uint32)((uint32)0U << 4U)     /* start byte - master only */
        | (uint32)((uint32)0U << 3U)     /* free data format */
        | (uint32)(I2C_2_BIT);           /* bit count */

  /** - set i2c extended mode */
  i2cREG2->EMDR = (uint32)0U << 1U; /* Ignore Nack Enable/Disable */

  /** - set i2c Backward Compatibility mode */
  i2cREG2->EMDR |= 0U;

  /** - Disable DMA */
  i2cREG2->DMACR = 0x00U;
```

```
  /** - set i2c data count */
  i2cREG2->CNT = 8U;

  /** - disable all interrupts */
  i2cREG2->IMR = 0x00U;

  /** - set prescale */
  i2cREG2->PSC = 8U;

  /** - set clock rate */
  i2cREG2->CKH = 5U;
  i2cREG2->CKL = 5U;

  /** - set i2c pins functional mode */
  i2cREG2->PFNC = (0U);

  /** - set i2c pins default output value */
  i2cREG2->DOUT = (uint32)((uint32)0U << 1U)     /* sda pin */
       | (uint32)(0U);     /* scl pin */

  /** - set i2c pins output direction */
  i2cREG2->DIR = (uint32)((uint32)0U << 1U)     /* sda pin */
       | (uint32)(0U);     /* scl pin */

  /** - set i2c pins open drain enable */
  i2cREG2->PDR = (uint32)((uint32)0U << 1U)     /* sda pin */
       | (uint32)(0U);     /* scl pin */

  /** - set i2c pins pullup/pulldown enable */
  i2cREG2->PDIS = (uint32)((uint32)0U << 1U)     /* sda pin */
      | (uint32)(0U);     /* scl pin */

  /** - set i2c pins pullup/pulldown select */
  i2cREG2->PSEL = (uint32)((uint32)1U << 1U)     /* sda pin */
       | (uint32)(1U);     /* scl pin */

  /** - set interrupt enable */
  i2cREG2->IMR    = (uint32)((uint32)0U << 6U)     /* Address as slave interrupt
*/
        | (uint32)((uint32)0U << 5U)     /* Stop Condition detect interrupt */
        | (uint32)((uint32)0U << 4U)     /* Transmit data ready interrupt   */
        | (uint32)((uint32)0U << 3U)     /* Receive data ready interrupt    */
        | (uint32)((uint32)0U << 2U)     /* Register Access ready interrupt */
        | (uint32)((uint32)0U << 1U)     /* No Acknowledgment interrupt    */
        | (uint32)(0U);     /* Arbitration Lost interrupt      */

  i2cREG2->MDR |= (uint32)I2C_RESET_OUT; /* i2c out of reset */

  /** - initialize global transfer variables */
  g_i2cTransfer_t[1U].mode   = (uint32)0U << 4U;
  g_i2cTransfer_t[1U].length = 0U;
}
```

→ HALCOGEN 에서 세팅한 초기값이 세팅된다.

```
void i2cSetOwnAdd(i2cBASE_t *i2c, uint32 oadd)
{
  i2c->OAR = oadd;
}
```

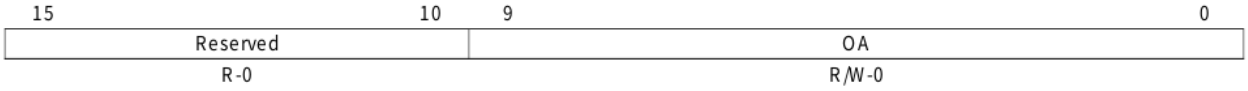
→ own address 를 셋팅하는 것으로 마스터 주소를 셋팅하는 함수이다.

i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

oadd : 7 혹은 10 bit 의 자신 주소(마스터 주소)를 넣어주면 된다.

→ OAR 은 Own Address Manager 로 16 bit memory-mapped I2C own address register 에 자체 주소(own address 마스터 주소)를 기입하는 것이다.

Figure 31-13. I2C Own Address Manager Register (I2COAR) [offset = 00]



LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

10-15 비트는 어떤 영향도 미치지 못한다.

0-9 비트는 I2C 모듈의 버스 주소를 반영한다. 확장 주소 (XA) 비트 I2CMDR.8 이 1 로 설정된 경우 I2C 는 확장 주소 모드 (10 비트 주소 지정 모드)에 있다.(= When the expand address (XA) bit I2CMDR.8 is set to 1, the I2C is in expand address mode (10-bit addressing mode)/ 확장비트는 보통 7 비트로 설정되어 있는데 10 비트로 쓰고 싶을 때 설정해주기 위해 사용하는 것이다.) 7 또는 10 비트 주소 모드에서 모든 10 비트는 모두 읽고 쓸 수 있다. 비트 7, 8 및 9 는 10 비트 주소 모드에서만 사용해야 한다. 표 31-5 는이 비트에 대한 correct mode 를 제공한다. 사용자는 시스템의

Table 31-5. Correct Mode for OA Bits

Bits Used	Mode	Value of XA
OA.6:0	7 Bit Addressing	0
OA.9:0	10 Bit Addressing	1

다른 구성 요소와 충돌하지 않는 한 I2C own addres 를 임의의 값으로 프로그래밍 할 수 있다.

```
void i2cSetSlaveAdd(i2cBASE_t *i2c, uint32 sadd)
{
  i2c->SAR = sadd;
}
```

→ 슬레이브 주소를 셋팅하는 함수이다.

i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

sadd : 7 혹은 10 bit 의 자신 주소(슬레이브 주소)를 넣어주면 된다.

→ I2C 슬레이브 주소 레지스터는 I2C 버스에서 통신 할 슬레이브 장치의 주소를 지정하는데 사용되는 16 비트 메모리 맵핑 레지스터이다. OAR 과 마찬가지로 10-15 는 설정하지 못하고 0-9 비트에 저장할 수 있다.

```
void i2cSetBaudrate(i2cBASE_t *i2c, uint32 baud)
{
  uint32 prescale;
  uint32 d;
  uint32 ck;
  float64 vclk = 75.000F * 1000000.0F;
  float64 divider= 0.0F;
  uint32 temp = 0U;

  divider = vclk / 8000000.0F;
  prescale = (uint32)divider - 1U;

  if(prescale>=2U)
  {
    d = 5U;
  }
  else
  {
    d = (prescale != 0U) ? 6U : 7U;
  }

  temp = 2U * baud * 1000U * (prescale + 1U);
  divider = vclk / ((float64)temp);
  ck = (uint32)divider - d;

  i2c->PSC  = prescale;
  i2c->CKH = ck;
  i2c->CKL = ck;
}
```

→ 간단한 런타임 전송속도(Baudrate) 변경 함수이다.

Baudrate : 데이터 통신에서 직렬 전송의 변조 속도를 1 초간에 전송되는 신호의 수를 나타낸 값을 말한다. 단말 장치와 모뎀 간의 신호는 2 치 신호이므로 신호 수와 비트 수는 일치하여 변조 속도와 전송 속도(1 초간에 전송되는 비트 수로, 단위는 b/s)는 같다. 모뎀과 모뎀간에서는 신호는 2 치 신호라고만은 할 수 없고, PSK 등에 의해서 다치화함으로써(1 신호에 복수 비트를 싣는) 전송 효율을 높일 수 있다. 즉, 컴퓨터와 다른 장치에서 1 초 동안에 전송할 수 있는 자료 비트의 크기를 자료 전송 속도라 한다. 이 것은 모뎀과 모뎀사이나 컴퓨터와 프린터 사이에서의 자료 전송속도를 나타내는데 사용된다.

→ I2C 런타임 전송속도(baudrate)를 변경한다. I2C 모듈은 baudrate 변경을 이해하려면 리셋(nIRS=0 in I2CMDR)을 이해해야 한다.

→ i2cBASE_t *i2c : I2C 모듈 base address 로 HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

uint32 baud : KHz 의 baudrate. 변경하고자 하는 baudrate

```
→  temp = 2U * baud * 1000U * (prescale + 1U);
  divider = vclk / ((float64)temp);
  ck = (uint32)divider - d;

  i2c->PSC  = prescale;
  i2c->CKH = ck;
  i2c->CKL = ck;
```

*PSC = Prescale Register
VBUS_CLK 을 나누어서 6.7MHz ~ 13.3MHz 사이의 module clock 주파수를 사용하는 16 비트 메모리 맵핑 레지스터이다. 전치 분주기이다.
*CKH = Clock Control High Register
마스터 클록을 나누어서 I2C 시리얼 클록 하이 타임을 사용하는 16 비트 메모리 맵핑 레지스터이다. SCL 핀에 나타날 마스터 클럭 신호의 high-time portion 을 생성하기 위해 모듈 클록을 나누는데 사용된다.

$$HighTime = \left(\frac{I2CCLKH + d}{ModuleClockFrequency} \right)$$

d 값은 I2CPSC 에 의해 결정되는 값(의존하는 값)이다.
*CKL = Clock Control Low Register

```
uint32 i2cIsTxReady(i2cBASE_t *i2c)
{
  return i2c->STR & (uint32)I2C_TX_INT;
}
```

→Tx buffer ready flag 가 설정되었는지 확인하는 함수이다. 플래그가 설정되지 않았으면 0 을 반환하지 않고 자기 자신을 반환한다.
→ i2c module base address 를 인자를 주면 된다.

5	SCD		Stop condition detect interrupt flag. This bit is set to 1 when the I2C receives or sends a STOP condition. **This bit is cleared to 0 by writing a 1 to this bit or reading the value 0x0006 from I2CIVR. Writing a 1 to this bit will clear the value 0x0006 from I2CIVR.**
		0	No STOP condition has been sent or received.
		1	A STOP condition has been sent or received.

I2C_TX_INT =16(*I2C_TX_INT* = 0x0010U) 으로 정의 되어 있다. &연산이므로 5bit 를 확인하면 되는데 이 비트는 정지 조건 검출 인터럽트 플래그로 I2CIVR 에서 0x0006 값을 읽거나 1 로 쓰여질 경우 비트가 0 으로 클리어 된다.이 비트에 1 을 쓰면 I2CIVR 에서 0x0006 값이 지워진다.
0 일 경우 STOP 조건이 송수신 되지 않고 1 일 경우 STOP 조건이 송수신 된다. 따라서 값은 확인하고자 하는 I2C 모듈의 주소를 주면 된다.HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

```
void i2cSendByte(i2cBASE_t *i2c, uint8 byte)
{
  /*SAFETYMCUSW 28 D MR:NA <APPROVED> "Potentially infinite loop found - Hardware Status check for execution sequence" 잠재적인 무한 루프로 실행 순서에 따른 하드웨어 상태 체크하는 곳이다,*/
  while ((i2c->STR & (uint32)I2C_TX_INT) == 0U)
  {
  } /* Wait */
  i2c->DXR = (uint32)byte;
}
```

→ 폴링 모드에서 byte 하나 씩 보내는 함수이다. 바이트를 보내기 전, 전송 버퍼가 비어있을 때까지 루틴에서 대기한다. 이 때, i2cIsTxReady 를 사용해서 버퍼를 확인하면 대기를 피할 수 있다.
→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.
uint8 byte : 전송할 바이트(크기)를 넣어주면기 된다.
→ DXR 레지스터는 데이터를 전송한다. 이 레지스터에 기록 된 데이터는 I2C 버스를 통해 전송된다. 이 레지스터에 쓰면 TXRDY 비트가 지워지고 코드 0x05 가 I2CIVR 레지스터에서 지워진다.

```
void i2cSend(i2cBASE_t *i2c, uint32 length, uint8 * data)
{
  uint32 index = i2c == i2cREG1 ? 0U : 1U;
  if ((g_i2cTransfer_t[index].mode & (uint32)I2C_TX_INT) != 0U)
  {
    /* we are in interrupt mode */
    /*SAFETYMCUSW 45 D MR:21.1 <APPROVED> "Valid non NULL input
parameters are only allowed in this driver" */
    g_i2cTransfer_t[index].data   = data;

    /* start transmit by sending first byte */
    /*SAFETYMCUSW 45 D MR:21.1 <APPROVED> "Valid non NULL input
parameters are only allowed in this driver" */
    i2c->DXR = (uint32)*g_i2cTransfer_t[index].data;
    /*SAFETYMCUSW 45 D MR:21.1 <APPROVED> "Valid non NULL input
parameters are only allowed in this driver" */
    g_i2cTransfer_t[index].data++;
    /* Length -1 since one data is written already */
    g_i2cTransfer_t[index].length = (length - 1U);
    /* Enable Transmit Interrupt */
    i2c->IMR |= (uint32)I2C_TX_INT;
  }
  else
  {
    /* send the data */
    /*SAFETYMCUSW 30 S MR:12.2,12.3 <APPROVED> "Used for data count in
Transmit/Receive polling and Interrupt mode" */
    while (length > 0U)
    {
      /*SAFETYMCUSW 28 D MR:NA <APPROVED> "Potentially infinite loop found
- Hardware Status check for execution sequence" */
      while ((i2c->STR & (uint32)I2C_TX_INT) == 0U)
      {
      } /* Wait */
      /*SAFETYMCUSW 45 D MR:21.1 <APPROVED> "Valid non NULL input
parameters are only allowed in this driver" */
      i2c->DXR = (uint32)*data;
      /*SAFETYMCUSW 45 D MR:21.1 <APPROVED> "Valid non NULL input
parameters are only allowed in this driver" */
      data++;
      length--;
    }
  }
}
```

→ 'data'및 'length'바이트가 가리키는 데이터 블록을 보낸다. 인터럽트가 활성화 된 경우 인터럽트 모드를 사용하여 데이터가 전송되고, 그렇지 않으면 폴링 모드가 사용된다. 인터럽트 모드에서 첫 번째 바이트의 전송이 시작되고 루틴이 즉시 반환된다. i2cNotification 콜백이 호출 될 때, 전송이 완료 될 때까지 i2cSend 를 다시 호출할 수 없다. 폴링 모드에서 i2cSend 는 전송이 완료 될 때까지 리턴하지 않는다.
→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.
uint32 length : 전송한 데이터의 길이, 글자수
uint8 * data : 전송할 데이터 포인터
→ i2cREG 에 따라 배열 구조체가 정해진다. Mode, length, *data 가 들어있다. 여기서 mode 값에 따라 정의 된 *I2C_TX_INT* = 0x0010U 값을 and 연산한다. 인터럽트 모드 일 경우 if 문 아닐 경우 else 로 들어가게 된다. 그러면 코드에 따라 데이터가 전송된다. DXR 레지스터는 데이터를 전송하는 레지스터이므로 구조체 data 에 저장된 데이터를 저장하고 전송한다. 이미 길이에서 1 를 써서 길이에 -1 을 저장하고 IMR 에 4 번 비트에 1 을 셋팅한다. 이는 Transmit data ready interrupt 를 허용하는 것이다.
인터럽트 모드가 아닌 경우 폴링모드로 빠지는데 STR 의 4 번째 비트로 Transmit data ready interrupt flag 이다. 이 플래그가 1 로 셋팅되면 I2CDXR 이 비어있는 것이고 0 이면 I2CDXR 상태 데이터를 전송하는 것이다. else 에서 이 비트가 0 일 경우 참이므로 I2CDXR 이 비어있으므로 무한루프를 돌면서 대기하게 된다. 1 이면 while 문이 조건에 맞지 않으므로 나와서 if 문과 같은 작업으로 데이터를 전달해주면서 데이터와 길이 후속 계산을 해준다. 다만 IMR 에 I2C_TX_INT 값을 대입해주는 작업을 하지 않을 뿐이다.

```
uint32 i2cIsRxReady(i2cBASE_t *i2c)
{
  return i2c->STR & (uint32)I2C_RX_INT;
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.
→ Rx 버퍼가 차있는지 확인하는 함수이다. Rx buffer flag 가 다 셋팅 되어있는지 확인하는 것으로 플래그가 설정되지 않았으면 0 을 반환하지 않고 자기 자신을 반환한다. *I2C_RX_INT* = 0x0008U

3	RXRDY		Receive data ready interrupt flag. This bit is set to 1 to indicate when the data in the receive shift register has been copied into the data receive register (I2CDRR). This bit can also be polled by the device to indicate when to read the received data in the I2CDRR. **Writing a 1 to this bit or reading from I2CDRR will clear this bit, and will also clear code 0x0004 from I2CIVR. This bit cannot be cleared by reading the I2CIVR register.**
		0	The I2CDRR has been read.
		1	The received data has been written into the I2CDRR.

으로 설정되어 있으므로 3 비트를 확인하면 된다.
데이터 수신 준비 인터럽트 플래그. 이 비트가 1 로 셋팅 되었을 때, receive shift register 에 있는 데이터가 data receive register (I2CDRR)가 복사되었을 때를 나타낸다. 이 비트는 I2CDRR 에 속한 수신 데이터를 읽을 때를 나타내는 디바이스에 의해 폴링될 수 있다.

이 비트에 1 을 쓰거나 I2CDRR 에서 읽으면 이 비트가 지워지고 I2CIVR 에서 코드 0x0004 도 지워진다. I2CIVR 레지스터를 읽음으로써 클리어 될 수 없다. 0 으로 셋팅 될 때, I2CDRR 을 읽는다. 1 로 셋팅 되면 수신 된 데이터가 I2CDRR 에 기록된다.

```
uint32 i2cIsStopDetected(i2cBASE_t *i2c)
{
  return i2c->STR & (uint32)I2C_SCD_INT;
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

5	SCD		Stop condition detect interrupt flag. This bit is set to 1 when the I2C receives or sends a STOP condition. **This bit is cleared to 0 by writing a 1 to this bit or reading the value 0x0006 from I2CIVR. Writing a 1 to this bit will clear the value 0x0006 from I2CIVR.**
		0	No STOP condition has been sent or received.
		1	A STOP condition has been sent or received.

→ 정지 상태가 되었는지 확인하는 함수이다. Stop condition flag 가 설정되었는지 체크하는 것으로 플래그가 설정되지 않았으면 0 을 반환하지 않고 자기 자신을 반환한다. *I2C_SCD_INT* = 0x0020U 로 설정되어있다.

→ I2C 가 STOP condition 을 송수신할 때, 이 비트는 1 로 셋팅되어 있다. 이 비트는 1 을이 비트에 쓰거나 I2CIVR 에서 0x0006 값을 읽음으로써 0 으로 클리어 된다. 이 비트에 1 을 쓰면 I2CIVR 에서 0x0006 값이 지워진다.

```
void i2cClearSCD(i2cBASE_t *i2c)
{
  i2c->STR = (uint32)I2C_SCD_INT;
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

→ STOP condition detect(SCD) 플래그를 클리어하는 함수이다. 즉, SCD 플래그를 지우기 위해 호출된다. STR 에서 SCD 에 0 으로 셋팅 되면 STOP condition 이 송수신 되지 않는다. 그러나 1 로 셋팅되면 STOP condition 이 송수신된다. str 에 *I2C_SCD_INT* 이 대입되므로 이 함수가 실행되는 것이다.

```
uint32 i2cRxError(i2cBASE_t *i2c)
{
  uint32 status = i2c->STR & ((uint32)I2C_AL_INT | (uint32)I2C_NACK_INT);
  i2c->STR = (uint32)((uint32)I2C_AL_INT | (uint32)I2C_NACK_INT);
  return status;
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

→ Rx 에러 플래그를 리턴하는 함수이다. 리턴하기 전에 에러 플래그를 클리어 해준다. *I2C_AL_INT* = 0x0001U, *I2C_NACK_INT* = 0x0002U 로 정의 되어있으므로 0,1 비트를 확인하면 된다.

1	NACK		No acknowledgement interrupt. This bit is set to 1 when the master I2C does not receive an acknowledgement from the receiver. This bit is set only when the I2C has received a no-acknowledge in master mode. This bit is NOT set by no-acknowledgement after Start byte. In master start byte mode, the first byte (address of all zeroes) receives a NACK but does not clear the stop bit. **Writing a 1 to this bit or reading the value 0x0002 from I2CIVR will clear this bit.** 0 An acknowledge was detected. 1 No acknowledge was detected or the I2C is operating in the general call, even though an acknowledgement was received. This value clears the STP bit.
0	AL		Arbitration lost interrupt flag. This bit is set to 1 when arbitration has been lost. **Writing a 1 to this bit or reading the value 0x0001 from I2CIVR will clear this bit.** 0 No loss of arbitration has been detected. 1 The device in the master transmitter mode senses it has lost an arbitration. This occurs when two or more transmitters start a transmission almost simultaneously or when the I2C attempts to start a transfer while BB=1. When this is set to 1 due to arbitration lost, the device becomes a slave receiver and the MST, STT and STP bits in I2CMDR are cleared to 0.

0 번 비트는 arbitration(중재)이 없을 때(잃어버릴 때), 설정되는 인터럽트 플러그이다. 이 비트에 1 이 써지거나 I2CIVR 의 0x0001 값을 읽으면 이 비트를 클리어해준다. 0 은 중재가 사라진 것이 감지되지 않았을 때, 셋팅되는 값이다. 1 은 마스터 전송기 모드의 장치가 arbitaration 을 잃어버린 것을 감지했을 때, 셋팅된다. 이는 2 개 이상의 송신기가 거의 동시에 전송을 시작하거나 I2C 가 BB=1 일 동안 전송을 시작하려고 시도할 때, 일어난다. arbitration 이 손실되어 1 로 셋팅이 될 때, 장치는 슬레이브 수신기가 되고 I2CMDR 에 있는 MST, STT, STP 비트가 0 으로 클리어 된다.
1 번 비트는 acknowledgement 인터럽트가 없을 때 사용하는 비트이다. 이 비트는 마스터 I2C 가 슬레이브에서 확인(acknowledgement)을 수신 못했을 때 1 로 셋팅 된다. 마스터 모드에서 no-acknowledge 를 수신했을 때만 셋팅된다. Start byte 뒤, no-acknowledge 에 의해 셋팅되지 않는다. 마스터 시작 바이트 모드에서 첫번째 바이트(모든 0 의 주소)는 NACK 수신하지만 정지 비트는 클리어하지 않는다.
I2CIVR 에서 0x002 값을 읽거나 1 을 이 비트에 쓰면 비트가 클리어 된다. 이 비트가 0 이 되면 인지가 감지된 것이다. 1 이면 acknowledgement(승인)을 수신했지만 승인이 감지되지 않거나 I2C 가 작동 중인 것이다. 이 값은 STP 비트를 지운다(클리어 한다).

*Arbitration(중재)
I2C 에서 arbitration 은 둘 이상의 마스터 송신기가 동시에 동일한 버스에서 전송을 시작하면 중재 절차가 호출된다. 그림 31-11 은 두 장치 간의 중재 절차(중재 진행상의 순서, 중재 순서)를 보여준다. 중재 절차는 경쟁 송신기가 SDA 버스에 제시 한 데이터를 사용한다. high 를 생성하는 첫 번째 마스터 트랜스미터는 low 를 생성하는 다른 마스터에 의해 기각된다(무효화 된다). 중재 절차는 가장 낮은 이진법 값을 갖는 serial data stream 을 전송하는 장치에 우선 순위를 부여한다. arbitration(중재)을 손실하면 master transmitter(마스터 송신기)는 slave receiver mode(슬레이브 수신 모드)로 전환되고 Arbitration lost(AL) flag 를 셋팅하며, arbitration-lost interrupt(중재 손실 인터럽트)를 생성한다. 다른 마스터 모듈에서 전송 된 데이터는 복구되고 I2C 는 마스터 모듈에서 계속 데이터를 수신한다. 두 개 이상의 장치가 동일한 첫 번째 바이트를 보내면 후속 바이트에서 중재가 계속된다.
반복 된 START 조건 또는 STOP 조건이 I2C 버스로 전송 될 때, serial transfer 중 중재 절차가 여전히 진행 중이면 관련된 마스터 전송기는 형식 프레임의 동일한 위치에서 반복 된 START 조건 또는 STOP 조건을 전송해야한다. 즉, 중재는 다음 상태에선 허용되지 않는다.
• 반복 된 START 조건 및 데이터 비트

- STOP 조건과 데이터 비트
- 반복 된 START 조건 및 STOP 조건

슬레이브는 중재 절차에 관여하지 않는다.

```
uint8 i2cReceiveByte(i2cBASE_t *i2c)
{
  /*SAFETYMCUSW 28 D MR:NA <APPROVED> "Potentially infinite loop found - Hardware Status check for execution sequence" */
  while ((i2c->STR & (uint32)I2C_RX_INT) == 0U)
  {
  } /* Wait */
  return ((uint8)i2c->DRR);
}
```

→ 수신된 바이트를 리턴하는 함수다. 폴링모드에서 단일 바이트를 수신받는다. 만약 수신 버퍼에 바이트가 없으면 루틴은 수신될 때까지 기다린다. 대기를 피하기 위해선 i2cIsRxReady 를 사용해서 버퍼가 꽉 차있는지 확인하면 된다.

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

I2C_RX_INT = 0x0008U 로 설정되어 있고 and 연산을 하므로 STR 에 값이 있으면 실행되고 아니면 빠져나가므로 3 번 비트를 보면 된다.

3	RXRDY		Receive data ready interrupt flag. This bit is set to 1 to indicate when the data in the receive shift register has been copied into the data receive register (I2CDRR). This bit can also be polled by the device to indicate when to read the received data in the I2CDRR. **Writing a 1 to this bit or reading from I2CDRR will clear this bit, and will also clear code 0x0004 from I2CIVR. This bit cannot be cleared by reading the I2CIVR register.**
		0	The I2CDRR has been read.
		1	The received data has been written into the I2CDRR.

Data ready(데이터 준비) 수신 인터럽트 플래그로 이 비트가 1 로 셋팅되면,수신 시프트 레지스터의 데이터가 데이터 수신 레지스터 (I2CDRR)로 복사되었을 때를 나타낸다. 이 비트는 I2CDRR 에서 수신 된 데이터를 읽을 때를 표시하기 위한 디바이스(장치)에 의해 폴링 될 수 있다. 이 비트에 1 을 쓰거나 I2CDRR 에서 읽으면 이 비트가 지워지고, I2CIVR 에서 코드 0x0004 또한 지워집니다(클리어 된다). 이 비트는 I2CIVR 레지스터를 읽음으로써 지워지지 않는다.
0 으로 셋팅되면 I2CDRR 을 읽고, 1 로 셋팅되면 수신 된 데이터가 I2CDRR 에 기록된다.

```
void i2cReceive(i2cBASE_t *i2c, uint32 length, uint8 * data)
{
  uint32 index = i2c == i2cREG1 ? 0U : 1U;
  if ((i2c->IMR & (uint32)I2C_RX_INT) != 0U)
  {
    /* we are in interrupt mode */
    /* clear error flags */
    i2c->STR = (uint32)I2C_AL_INT | (uint32)I2C_NACK_INT;

    g_i2cTransfer_t[index].length = length;
    /*SAFETYMCUSW 45 D MR:21.1 <APPROVED> "Valid non NULL input parameters are only allowed in this driver" */
    g_i2cTransfer_t[index].data   = data;
  }
```

```
  else
  {   /*SAFETYMCUSW 30 S MR:12.2,12.3 <APPROVED> "Used for data count in
Transmit/Receive polling and Interrupt mode" */
    while (length > 0U)
    {
      /*SAFETYMCUSW 28 D MR:NA <APPROVED> "Potentially infinite loop found
- Hardware Status check for execution sequence" */
      while ((i2c->STR & (uint32)I2C_RX_INT) == 0U)
      {
      } /* Wait */
      /*SAFETYMCUSW 45 D MR:21.1 <APPROVED> "Valid non NULL input
parameters are only allowed in this driver" */
      *data = ((uint8)i2c->DRR);
      /*SAFETYMCUSW 45 D MR:21.1 <APPROVED> "Valid non NULL input
parameters are only allowed in this driver" */
      data++;
      length--;
    }
  }
}
```

→ 'length'길이 만큼 블록을 수신하여 'data'가 가리키는 데이터 버퍼에 넣는다. 인터럽트가 활성화 된 경우 인터럽트 모드를 사용하여 데이터를 수신하고 그렇지 않으면 폴링 모드가 사용된다. 인터럽트 모드에서는 수신이 설정되고 루틴이 즉시 반환된다. i2cNotification 콜백이 호출 되었을 때, i2cReceive 는 전송이 완료 될 때까지 다시 호출할 수 없다. 폴링 모드에서 i2cReceive 는 전송이 완료 될 때까지 리턴할 수 없다.

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

uint32 length : 전송한 데이터의 길이, 글자수

uint8 * data : 전송할 데이터 포인터

i2c 가 i2cREG1 일 경우, 구조체 배열 인덱스 0 에 저장이 되고 아니면 1 에 저장된다. *I2C_RX_INT* = 0x0008U 의 값을 가지기 때문에 IMR 의 3 번 비트를 확인 하면 된다.

3	RXRDYEN		Receive Data Ready Interrupt Enable.
		0	RXRDYEN interrupt is disabled.
		1	RXRDYEN interrupt is enabled.

수신 데이터 준비 인터럽트를 허용하는지에 대한 비트이다. 1 이면 허용된 것이고 if 문 또한 참이므로 인터럽트 모드로 들어가 이를 이용해서 데이터를 수신하게 된다. 먼저 STR 의 에러 비트를 클리어해준다. 구조체 length 와 data 에 값을 넣어준다.

반대로 0 이면 인터럽트가 허용되지 않아 폴링모드로 빠진다. STR 에서 I2C_RX_INT 값을 and 연산으로 while 문을 작동한다. 이 비트가 설정될 경우 I2CDRR 에 수신데이터가 작성된다. 그리고 while 문을 나와 DRR 값을 데이터에 넣어주고 길이, 데이터의 후속 연산을 해준다.

3	RXRDY		Receive data ready interrupt flag. This bit is set to 1 to indicate when the data in the receive shift register has been copied into the data receive register (I2CDRR). This bit can also be polled by the device to indicate when to read the received data in the I2CDRR. **Writing a 1 to this bit or reading from I2CDRR will clear this bit, and will also clear code 0x0004 from I2CIVR. This bit cannot be cleared by reading the I2CIVR register.**
		0	The I2CDRR has been read.
		1	The received data has been written into the I2CDRR.

```
void i2cEnableNotification(i2cBASE_t *i2c, uint32 flags)
{
  uint32 index = i2c == i2cREG1 ? 0U : 1U;
  g_i2cTransfer_t[index].mode |= (flags & (uint32)I2C_TX_INT);
  i2c->IMR                    = (flags & (uint32)(~(uint32)I2C_TX_INT));
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

uint32 flags : 인터럽트를 활성화하려면 다음 값을 넣어준다.

```
i2c_FE_INT    - framing error,
i2c_OE_INT    - overrun error,
i2c_PE_INT    - parity error,
i2c_RX_INT    - receive buffer ready,  0x0008U
i2c_TX_INT    - transmit buffer ready, 0x0010U
i2c_WAKE_INT  - wakeup,
i2c_BREAK_INT - break detect
```

i2c 가 i2cREG1 이면 인덱스가 0, 1 로 정해지고 그 구조체 안의 변수인 mode 에 플래그와 and 연산한 값을 or 연산으로 대입힌다. IMR 에 *I2C_TX_INT* = 0x0010U 을 NOT 연산한 값을 플래그와 AND 연산하여 대입한다. 플래그에 어떠한 값이 들어가더라도 IMR 의 4 비트는 0 으로 셋팅된다.

```
void i2cDisableNotification(i2cBASE_t *i2c, uint32 flags)
{
  uint32 index = i2c == i2cREG1 ? 0U : 1U;
  uint32 int_mask;
  g_i2cTransfer_t[index].mode &= (uint32)~(flags & (uint32)I2C_TX_INT);
  int_mask                    = i2c->IMR &(uint32)(~(uint32)(flags |
(uint32)I2C_TX_INT));
  i2c->IMR                    = int_mask;
}
```

→ 위의 함수와 같은 인자를 사용한다. 다만 mode 에 대입하는 값과 IMR 에 대입하는 값이 다르게 된다. 플래그와 AND 연산한 값을 NOT 연산으로 0, 1 을 바꾸어 준 뒤에 mode 의 값과 and 연산을 해서 대입해준다. mask 라는 변수에는 플래그와 or 연산한 값을 not 연산한 후, IMR 과 AND 연산한 값을 대입해준다.

```
void i2cSetStart(i2cBASE_t *i2c)
{
  i2c->MDR |= (uint32)I2C_START_COND;  /* set start condition */
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

→ 마스터모드에서만 사용할 수 있으며 I2C 의 start bit 생성해서 셋팅하는 함수이다. *I2C_START_COND* = 0x2000U 의 값을 or 연산해서 MDR 에 대입한다.

Bit	Field	Value	Description
15	NACKMOD		No-acknowledge (NACK) mode. This bit is used to send an acknowledge (ACK) or a no-acknowledge (NACK) to the transmitter. This bit is only applicable when the I2C is in receiver mode. In master receiver mode, when the internal data counter decrements to 0, the I2C sends a NACK. The master receiver I2C finishes a transfer when it sends a NACK. The I2C ignores ICCNT when NACKMOD is 1. The NACKMOD bit should be set before the rising edge of the last data bit if a NACK must be sent, and this bit is cleared once a NACK has been sent.
		0	The I2C sends an ACK to the transmitter during the acknowledge cycle.
		1	The I2C sends a NACK to the transmitter during the acknowledge cycle.

```
void i2cSetStop(i2cBASE_t *i2c)
{
  i2c->MDR |= (uint32)I2C_STOP_COND;  /* generate stop condition */
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

→ 마스터모드에서만 사용할 수 있으며 I2C 의 stop bit 생성해서 셋팅하는 함수이다.

I2C_STOP_COND = 0x0800U 의 값을 or 연산해서 MDR 에 대입한다.

11	STP		Stop condition (Master mode only). This bit can be set to a 1 by the CPU to generate a stop condition. It is reset to 0 by the hardware after the stop condition has been generated. The stop condition is generated when ICCNT passes 0 when the I2C is in non-repeat mode (RM=0). In repeat mode (RM=1), the stop condition is generated if STP bit is 1. In transmitter mode, I2CTXRDY needs to be 1 (that is, you have to set STP bit unless you write data into I2CDXR).
		0	STP is reset to 0 by the hardware after the STOP condition has been generated.
		1	STP is set to 1 by the device to generate a STOP condition.

```
void i2cSetCount(i2cBASE_t *i2c ,uint32 cnt)
{
  i2c->CNT = cnt;  /* set i2c count  */
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

uint32 cnt : data count 의 값을 넣어주면 된다.

→ 마스터모드에서만 사용할 수 있으며 i2c 카운트를 전송 값으로 설정하면 그 이후에 중지 조건을 생성하는 작업이 필요하다.

Bit	Field	Value	Description
15-0	CNT		Data counter. This down counter is used to generate a stop condition if a stop condition is specified (STP = 1). **Note: ICCNT is a don't care when RM is set to 1.**
		0	The data counter is 65536.
		1	The data counter is 1.

이 down counter 는 정지 조건이 지정된 경우 (STP = 1) 정지 조건을 생성하는 데 사용된다. RM 이 1 로 셋팅된 경우 ICCNT 는 상관하지 않는다. 0 으로 셋팅되면 data counter 는 65536 이고 1 로 셋팅되면 1 된다.

```
void i2cEnableLoopback(i2cBASE_t *i2c)
{
  /* enable digital loopback    */
  i2c->MDR |= ((uint32)1U << 6U);
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.
→ 자가테스트에서 loopback 모드를 허용해주는 함수이다. MDR 의 6 비트를 확인하면 된다.

6	DLB		Digital loop back enable bit.
			This bit enables the digital loopback mode of the I2C. This bit only applies in Master transmitter mode.
		0	Digital loop back mode is disabled.
		1	Digital loop back mode is enabled. In digital loop back mode, data transmitted out of the I2CDXR will be received in the I2CDRR. The address of the I2COAR is output on SDA.

1 으로 셋팅되면 Digital loop back mode 를 허용한다. Digital loop back mode 에서 I2C DXR 에서 전송 된 데이터는 I2CDRR 에서 수신된다. I2COAR 의 주소는 SDA 에서 출력된다.

```
void i2cDisableLoopback(i2cBASE_t *i2c)
{
  /* Disable Loopback Mode */
  i2c->MDR &= 0xFFFFFFBFU;
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.
→ i2cEnaableLoopback 함수와 반대이다. F = 1111 B = 1011 이므로 MDR 에서 6 번째 비트를 0 으로 셋팅하여 Digital loop back mode 를 허용하지 않는 것(억제)이다.

```
void i2cSetMode(i2cBASE_t *i2c, uint32 mode)
{
      uint32  temp_mdr;
  /* set Master or Slave Mode */
      temp_mdr  = (i2c->MDR & (~I2C_MASTER));
  i2c->MDR  = (temp_mdr | mode);
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.
uint32 mode : I2C_MASTER, I2C_SLAVE 를 적어주면 된다. 이는 *I2C_MASTER* = 0x0400U, *I2C_SLAVE* = 0x0000U 로 enum i2cMode 가 정의되어 있기 때문에 가능하다.
→ 마스터나 슬레이브 모드를 셋팅하는 함수이다. MDR 에서 10 비트를 어떤 식으로 셋팅하는지에 따라 마스터나 슬레이브 모드가 결정된다. I2C_MASTER 값을 not 연산하고 MDR 값과 AND 연산을 하여 기존에 셋팅된 값은 유지하면서 10 비트만 비교할 수 있도록 mdr 이라는 변수에 저장을 해준 것이다. 그래서 그 값으로 or 연산을 해서 MDR 에 대입한 것이다.

10	MST		Master/slave mode bit.
			This bit determines whether the module will operate in master or slave mode; see Table 31-17. This bit is cleared after generating a STOP condition. The BB bit is cleared first, and MST bit is cleared second. Before starting the next transaction in master mode, this bit must be confirmed to be cleared.
		0	The module is in the slave mode and the clock is received from the master device.
		1	The module is in the master mode and it generates the clock. This bit is cleared when the transfer has completed.

이 비트는 STOP 조건을 생성 한 후 지워진다. BB 비트가 먼저 지워지고 MST 비트가 두 번째로 지워진다. 마스터 모드에서 다음 transaction(처리)을 시작하기 전에 이 비트를 지워야 하는 것을 확인해야 한다. 0 으로 셋팅되면 슬레이브 모드이고 마스터 장치에서 클럭을 수신한다. 반면 1 은 마스터 모드이고 클럭을 발생시킨다. 전송이 완료되면 비트가 지워진다.

```
void i2cSetDirection(i2cBASE_t *i2c, uint32 dir)
{
  /* set Transmit/Receive mode */
  i2c->MDR  &= ~I2C_TRANSMITTER;
  i2c->MDR  |= dir;
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

uint32 dir : I2C_TRANSMITTER, I2C_RECEIVER 를 사용하면 된다.

i2cSetMode 함수와 같이 *I2C_TRANSMITTER* = 0x0200U, *I2C_RECEIVER* = 0x0000U 로 정의되어 있다.

→ 송수신을 결정하는 함수다. MDR 9 비트는 송수신을 결정해주는 비트로, I2c 모듈에 데이터 전송 방향을 결정하는 비트이다. 0 이면 수신 모드이고 SDA line 의 데이터가 데이터 레지스터 I2CDRR 로 시프트한다. 1 이면 모듈이 전송 모드에 있고 I2CDXR 의 데이터가 SDA line 에서 시프트를 한다.

9	TRX		Transmit/receive bit. This bit determines the direction of data transmission of the I2C module. See Table 31-17.
		0	The module is in the receive mode and data on the SDA line is shifted into the data register I2CDRR.
		1	The module is in the transmit mode and the data in the I2CDXR is shifted out on the SDA line.

```
bool i2cIsMasterReady(i2cBASE_t *i2c)
{
  bool retVal = 0U;
  /* check if MST bit is cleared. */
  if((i2c->MDR & I2C_MASTER) == 0)
  {
    retVal = true;
  }
  else
  {
    retVal = false;
  }
  return retVal;
}
```

→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

```
bool i2cIsBusBusy(i2cBASE_t *i2c)
{
  bool retVal = 0U;
  /* check if BB bit is set. */
  if((i2c->STR & I2C_BUSBUSY) == I2C_BUSBUSY)
  {
    retVal = true;
  }
  else
  {
    retVal = false;
  }
  return retVal;
```

```
}
```
→ i2cBASE_t *i2c : HALCOGEN 에서 설정한 I2Cxdriver 로 i2cREG1, i2cREG2 중 선택해서 사용하면 된다.

```
void i2c2GetConfigValue(i2c_config_reg_t *config_reg,
config_value_type_t type)
{
  if (type == InitialValue)
  {
    config_reg->CONFIG_OAR  = I2C2_OAR_CONFIGVALUE;
    config_reg->CONFIG_IMR  = I2C2_IMR_CONFIGVALUE;
    config_reg->CONFIG_CLKL = I2C2_CLKL_CONFIGVALUE;
    config_reg->CONFIG_CLKH = I2C2_CLKH_CONFIGVALUE;
    config_reg->CONFIG_CNT  = I2C2_CNT_CONFIGVALUE;
    config_reg->CONFIG_SAR  = I2C2_SAR_CONFIGVALUE;
    config_reg->CONFIG_MDR  = I2C2_MDR_CONFIGVALUE;
    config_reg->CONFIG_EMDR = I2C2_EMDR_CONFIGVALUE;
    config_reg->CONFIG_PSC  = I2C2_PSC_CONFIGVALUE;
    config_reg->CONFIG_DMAC = I2C2_DMAC_CONFIGVALUE;
    config_reg->CONFIG_FUN  = I2C2_FUN_CONFIGVALUE;
    config_reg->CONFIG_DIR  = I2C2_DIR_CONFIGVALUE;
    config_reg->CONFIG_ODR  = I2C2_ODR_CONFIGVALUE;
    config_reg->CONFIG_PD   = I2C2_PD_CONFIGVALUE;
    config_reg->CONFIG_PSL  = I2C2_PSL_CONFIGVALUE;
  }
  else
  {
  /*SAFETYMCUSW 134 S MR:12.2 <APPROVED> "Register read back support" */
    config_reg->CONFIG_OAR  = i2cREG2->OAR;
    config_reg->CONFIG_IMR  = i2cREG2->IMR;
    config_reg->CONFIG_CLKL = i2cREG2->CKL;
    config_reg->CONFIG_CLKH = i2cREG2->CKH;
    config_reg->CONFIG_CNT  = i2cREG2->CNT;
    config_reg->CONFIG_SAR  = i2cREG2->SAR;
    config_reg->CONFIG_MDR  = i2cREG2->MDR;
    config_reg->CONFIG_EMDR = i2cREG2->EMDR;
    config_reg->CONFIG_PSC  = i2cREG2->PSC;
    config_reg->CONFIG_DMAC = i2cREG2->DMACR;
    config_reg->CONFIG_FUN  = i2cREG2->PFNC;
    config_reg->CONFIG_DIR  = i2cREG2->DIR;
    config_reg->CONFIG_ODR  = i2cREG2->PDR;
    config_reg->CONFIG_PD   = i2cREG2->PDIS;
    config_reg->CONFIG_PSL  = i2cREG2->PSEL;
  }
}
```