Notes

By Harpinder Singh

Introduction

The I^2C interface in the RC32355, an integrated communication processor, facilitates connection to a number of standard I^2C external peripherals. The RC32355 I^2C interface supports 100 Kb/s (kilobits per second) speeds as well as 400 Kb/s (fast mode). As a master, it can address 7-bit or 10-bit I^2C slave devices. One of the most commonly used applications of the I^2C interface is to connect to a Serial EEPROM which is often used to store system configuration information. This application note discusses the interface to the Microchip 24AA64/24LC64 Serial Electrically Erasable PROM. Since the configuration information stored in the Serial EEPROM is required during the bootstrapping phase of the operating system, the sample program uses polling mode instead of interrupts in order to determine the completion of various I^2C interface operations. The RC32355 I^2C interface has the additional capability of functioning in slave mode, which is not covered in this application note.

RC32355 I^2C Interface Initialization

The I^2C controller needs to be initialized before it can perform any read/writes to slave I^2C devices connected on the I^2C bus. The required initialization sequence is described below:

Step 1: The I^2C interface connects to the external I^2C bus using two pins: I^2C clock (SCLP) and I^2C data (SDAP). Since these two pins are multiplexed with the GPIO pins, the initialization code enables I^2C signaling on these pins by selecting alternate function in the GPIO Function register. The program initializes the GPIO as shown below:

```
/* Enable GPIO pins for I2C Bus Alternate function  */
gpio.gpiofunc |= GPIO_I2C_SEL;
 The "gpio" data structure definition is:
typedef struct {
  unsigned int gpiofunc;          /* GPIO Function register                    */
  unsigned int gpiocfg;           /* GPIO Configuration register               */
  unsigned int gpiod;             /* GPIO data register  register               */
  unsigned int gpioilevel;        /*  GPIO interrupt level  register           */
  unsigned int gpioistat;         /*  GPIO interrupt status  register           */
  unsigned int gpionmien;         /*  GPIO nonmaskable interrupt enable register  */
} GPIO;
#define gpio (*((volatile GPIO *) GPIO_BASE))
where the GPIO_BASE is defined as :
#define GPIO_BASE       0xB8040000
```

The value of GPIO_I2C_SEL is 0xC000 because bits 15 and 14 in the GPIO Function register correspond to the I^2C SCLP and SDAP signals, respectively. When bits 15 and 14 are set, the SCLP and SDAP signals are enabled.

Notes

Step 2: The I^2C interface contains a 16-bit clock pre-scalar which is used to generate an internal I^2C bus pre-scalar clock that is used as a time base by the Master I^2C interface. The pre-scalar value (I^2CCP) is calculated from the following equation:

I^2C transfer rate = system clock frequency / (I^2CCP * 8)

I^2CCP = system clock frequency / (I^2C transfer rate * 8)

For I^2C transfer rates of 100 Kb/s on a board using 75 MHZ system clock, the value of I^2CCP can be obtained as:

I^2CCP = 75 * 10^6 / (100 * 10^3 * 8) = 94

The initialization code programs the I^2C Bus Clock Prescalar register with appropriate value as per the I^2C transfer rates.

```
/* Setup the PRESCALAR register: I2C data rate is
   = SYSCLK % (PRESCALAR * 8)                    */
i2c.i2ccp = PSL_75CLK_100KBPS  ;  /* PSL_75CLK_100KBPS = 94 for 100
        kbits/sec I2C transfer rate with board
        running at 75 MHZ system clock */
```

where the "i2c" data structure is defined as :

```
typedef struct {
  unsigned int i2cc;/* I2C bus control register */
  unsigned int i2cdi;/* I2C data input register   */
  unsigned int i2cdo;/* I2C data output register */
  unsigned int i2ccp;/* I2C bus clock prescalar register */
  unsigned int i2cmcmd;/* I2C bus master command register */
  unsigned int i2cms;/* I2C bus master status register */
  unsigned int i2cmsm;/* I2C bus master status mask register */
  unsigned int i2css;/* I2C bus slave status register */
  unsigned int i2cssm;/* I2C bus slave status mask register */
  unsigned int i2csaddr;/* I2C bus slave address register */
  unsigned int i2csack;/* I2C bus slave acknowledge register */
  unsigned int sysid ;              /* I2C system identification register  */
} I2C;
#define i2c (*((volatile I2C *) I2C_BASE))
```

where the "i2c" data structure is defined as :

```
typedef struct {
  unsigned int i2cc;/* I2C bus control register */
  unsigned int i2cdi;/* I2C data input register   */
  unsigned int i2cdo;/* I2C data output register */
  unsigned int i2ccp;/* I2C bus clock prescalar register */
  unsigned int i2cmcmd;/* I2C bus master command register */
  unsigned int i2cms;/* I2C bus master status register */
```

Notes

```
  unsigned int i2cmsm;/* I2C bus master status mask register */
  unsigned int i2css;/* I2C bus slave status register */
  unsigned int i2cssm;/* I2C bus slave status mask register */
  unsigned int i2csaddr;/* I2C bus slave address register */
  unsigned int i2csack;/* I2C bus slave acknowledge register */
  unsigned int sysid ;          /* I2C system identification register  */
} I2C;
#define i2c (*((volatile I2C *) I2C_BASE))
where  I2C_BASE is defined as :
#define I2C_BASE        0xB8070000
```

Step 3: Since in this example the RC32355 I^2C interface operates as a master, be sure the I^2C Master is enabled. Because we chose the poll mode of operation, all the interrupts at the I^2C device level must be masked. The initialization code lines implementing this are shown below:

```
/* Enable Master & Slave interfaces */
i2c.i2cc = I2CC_MEN;    /*  I2CC_MEN = 0x1 */
/* Mask off the I2C interrupts: Master / Slave   */
        i2c.i2cmsm = I2C_MASTER_INT_MASK;       /* I2C_MASTER_INT_MASK
      is 0xF*/
    /* Mask the Slave interrupts even though Slave interface not being used */
        i2c.i2cssm = I2C_SLAVE_INT_MASK ;         /* I2C SLAVE_INT_MASK is
      0x7F */
/* Flush the CPU write buffer as a precaution */
        wbflush( );      /* wbflush does dummy read from uncached address space
(tmp= *(0xbfc00000 );) which results in flushing of CPU write buffers */
```

Implementing Byte Writes to EEPROM

For performing byte writes, EEPROM requires a particular sequence of commands to be issued by the RC32355 I^2C controller. The command sequence is shown below:

1. I^2C controller starts the write byte operation with a start command.

```
 /* Issue a Start command to I2C */
i2c.i2cmcmd = I2CMCMD_CMD(START);/* Start Command */
/* Flush the CPU Write Buffers */
wbflush();
```

/* Read the master status register to determine if the command has completed. The completion is signaled by setting of the DONE bit in the I^2C master status register with Not Acknowledge (NA), the Lost Arbitration (LA) and Error (ERR) bits being reset. If the command does not complete within a short time, an error message is returned */

```
        status = sysChkI2CDone();
        if (status == ERROR)
```

Notes

```
return(ERROR);
```

2. Followed by this, a control byte is written to the EEPROM device. The upper 4 bits of the control byte constitute EEPROM device specific control code, the next 3 bits are for chip select ,and the last bit is the Read/Write* bit. Since this is a write operation, the Read/Write bit is held low. At the completion of the receipt of the control byte, the EEPROM device generates an ACK.

```
/* Write Control Byte. For 24AA64/24LC64 the value is 0xAE with the
      chip-select bits  "A2A1A0" being  7 in the case of 79IDTEB355 board */
i2c.i2cdo = (unsigned int )( I2C_24LC64_CNTL_BYTE ) ;
i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
     /* Flush the CPU Write Buffers */
     wbflush( );
     status = sysChkI2CDone();
     if (status == ERROR)
          return(ERROR);
```

3. I^2C controller sends the high address byte of the location in the EEPROM device where the data needs to be written on the I^2C bus and waits for the ACK from the EEPROM device.

```
    /* Write the most significant portion of the address.
       "address" points to the location in the  EEPROM */
i2c.i2cdo = (unsigned int) ( (address >> 8) & 0xFF ) ;
i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
     /* Flush the CPU Write Buffers */
     wbflush();
     status = sysChkI2CDone();
     if (status == ERROR)
          return(ERROR);
```

4. I^2C sends the least significant byte to the EEPROM.

```
    /* Write the least significant portion of the address */
 i2c.i2cdo = (unsigned int) ( address & 0xFF ) ;
     i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
     /* Flush the CPU Write Buffers */
     wbflush();
     status = sysChkI2CDone();
     if (status == ERROR)
          return(ERROR);
```

5. Followed by this, I^2C controller places on the I^2C bus the data that is to be written to the previously sent address location in the EEPROM device.

```
/* Write Data */
i2c.i2cdo = byte ;
i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
     /* Flush the CPU Write Buffers */
     wbflush();
```

Notes

```
status = sysChkI2CDone();
if (status == ERROR)
    return(ERROR);
```

6. Finally, the I^2C Controller sends a "stop" command. This initiates an internal write cycle in the EEPROM which completes the write operation.

```
/* Send a Stop Command to make the data byte to be written internally. Wait for
    some time to let this internal operation finish */
status = sysI2CStop();
if (status != OK)
return(ERROR);
WAIT(0x10000);
```

Sequential Writes to the EEPROM: Page Write

While writing a string to the EEPROM, single byte writes would be costly. A good alternative is to write a full page at a time. The page is 32 bytes in length and the range is determined by the lower 5 bits of the address (A4-A0). The I^2C controller follows the same sequence of commands as in byte write except that prior to generating a stop condition, the master transmits up to 31 additional bytes. The EEPROM continues to increment the address internally during these writes. The EEPROM increments only the lower 5 bits of the address using an internal counter which rolls over if more than 32 bytes are written. This assumes that the page write began at the page boundary (A4-A0 = 00000B). If this is not the case, then the program has to ensure that the I^2C controller sends just enough bytes that prevent the rollover of the EEPROM address counter. If care is not taken, this could cause corruption of the data written to the EEPROM. For example, if the starting address of the EEPROM is 0x1010 (A4A3A2A1A0 being 10000B), only 16 bytes can be written at a time. After the I^2C Master sends 16 bytes, the address counter will become A4A3A2A1A0: 11111B. Any further writes will result in the rollover of the count. So, the 17th byte, if written by I^2C, goes to the address 0x1000 instead of 0x1020.

The page write is initiated by calling function "sysI2CpageOpen" which is passed the starting address of the page as a parameter. This function performs the following steps:

1. Send a start command and checks its completion.

```
/* Send a Start command */
    i2c.i2cmcmd = I2CMCMD_CMD(START);/* Start Command */
    /* Flush the CPU Write Buffers */
    wbflush();
    /* Check if the command is done & no error bits are set */
    status = sysChkI2CDone();
    if(status == ERROR)
        return (ERROR);
```

2. Write control byte with Read/Write bit reset.

```
    /* Write Control Byte */
    i2c.i2cdo = (unsigned int )( I2C_24LC64_CNTL_BYTE ) ;
    i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
    /* Flush the CPU Write Buffers */
    wbflush();
```

Notes

```
status = sysChkI2CDone();
if (status == ERROR)
    return(ERROR);
```

3. Send the address of the location in the EEPROM where write begins.

```
/* Address */
i2c.i2cdo = (unsigned int) ( (address >> 8) & 0xFF) ;
i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
/* Flush the CPU Write Buffers */
wbflush();
status = sysChkI2CDone();
if (status == ERROR)
    return (ERROR);
i2c.i2cdo = (unsigned int) (address & 0xFF) ;
i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
/* Flush the CPU Write Buffers */
wbflush();
status = sysChkI2CDone();
if (status == ERROR)
    return (ERROR);
```

Once the page is opened, it is followed by the appropriate number of byte writes based on the starting address. The function "sysI2CWriteInPage" is called to do these byte writes. The operation performed in the "sysI2CWriteInPage" is shown below:

```
/* Write Data to the I2C output I2C register */
i2c.i2cdo = byteData; /* data to be written to the EEPROM */
i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
/* Flush the CPU Write Buffers */
wbflush();
/* Check if the command completed correctly */
status = sysChkI2CDone();
if (status == ERROR)
    return(ERROR);
```

The I^2C Master controller closes the "page write" by sending a Stop command. On receiving the Stop command, the I^2C device begins the internal write operation. An adequate delay needs to be inserted here to allow the internal EEPROM write operation to complete. Note that the sample program in this application note does not carry optimized delay values, with most of the values being in excess of what may be required.

Notes

Implementing Random Byte Reads

The random byte reads of EEPROM allow the I^2C controller to access any memory location in a random manner. The RC32355 I^2C controller issues a specific sequence of commands as given below to randomly read a location in EEPROM:

1. Setting of the EEPROM address: In this phase the I^2C controller issues a sequence of commands to set the internal address of the EEPROM to the location from where the data needs to be read.
   a) I^2C sends the Start command
   b Start command is followed by the control byte (Read/Write* bit as low)
   c) I^2C controller sends the address location in next two cycles to EEPROM.
2. Reading the addressed location: In this phase, I^2C controller initiates the read cycle as described below:
   a) I^2C sends yet another Start command
   b) Start command is followed by the control byte (Read/Write* bit is high)
   c) Once the Control command completes, I^2C controller sends the read command
   d) On completion of the Read command, data is available in the I^2C controller input register.

Implementing Sequential Reads from EEPROM

The Sequential command is very similar to the Random Read command. The I^2C controller issues Read With Acknowledge (RDACK) command instead of the Read command. This causes the I^2C controller to issue an ACK to the EEPROM device which responds by incrementing the internal address pointer. By using the address pointer, the entire EEPROM memory can be read during one single operation.

Code Listing

Disclaimer: Code examples provided by IDT are for illustrative purposes only and should not be relied upon for developing applications. IDT does not assume liability for any loss or damage that may result from the use of this code.

```
/* Wait Macro */
#define WAIT(x) { volatile int i=0; while (++i < (x)) ; }
/*******************************************************************
*
* sysI2CInit - initialize the I2C interface on the board
*
* This routine initializes I2C interface & GPIO pins
*
* RETURNS: N/A
*/
void sysI2CInit (void)
{
/* Enable GPIO pins for I2C Bus Alternate function  */
gpio.gpiofunc |= GPIO_I2C_SEL   ;
/* Enable Master & Slave interfaces */
i2c.i2cc = I2CC_MEN;
```

Notes

```
/* Setup the PRESCALAR register: I2C data rate is
   = SYSCLK % (PRESCALAR * 8)                    */
i2c.i2ccp = PSL_75CLK_100KBPS  ;
/* Mask off the I2C interrupts: Master/ Slave     */
i2c.i2cmsm = I2C_MASTER_INT_MASK;
          i2c.i2cssm = I2C_SLAVE_INT_MASK ;
/* Flush the CPU write buffer */
wbflush();
}
 /*****************************************************************
*
* sysChkI2CDone: Check if the I2C device has done condition
*
* This function reads the I2C Command Status register to check if the DONE bit is
* set.
*
* RETURNS: STATUS (OK if Acknowledge otherwise ERROR)
*/
STATUS  sysChkI2CDone
      (
      void
      )
      {
      unsigned int status                        ; /* I2C status */
      unsigned int writeI2CWaitNum1=0 ; /* To avoid endless loop */
      unsigned int writeI2CWaitNum2=0 ; /* To avoid endless loop */
      BOOL writeI2CError = FALSE       ; /* variable to flag an error */
      /* Wait for Acknowledge from the Device */
      while(1) {
            status = i2c.i2cms ;
            /* Check the status for Done */
            if ((status & 0xF) == I2CMS_D)
                  break ;
            /* Avoid Waiting forever */
            writeI2CWaitNum1 ++ ;
            if (writeI2CWaitNum1 > 100000){
                  writeI2CWaitNum2 ++ ;
                  writeI2CWaitNum1 = 0;
                  if (writeI2CWaitNum2 > 1000){/*Ack Should have come by now */
```

Notes

```
                        writeI2CError = TRUE;
                        break;
                        }
                    }
                }
        if (writeI2CError == TRUE)
            return(ERROR);
        else
            return(OK);
        } /* End of the sysChkI2CDone */
/*********************************************************************\***
*
* sysI2CWriteByte - Writes a byte into the I2C device
*
* This function writes a byte at a given address. It uses Write Acknowledge
* command. The function invokes the Stop command at the end.
*
* RETURNS: OK on completion else ERROR
*/
STATUS sysI2CWriteByte
(
unsigned short address , /* I2C Address */
unsigned char byte      /* Data       */
)
{
volatile unsigned int status ;  /* used to check the I2C Status state */

/* Issue a Start command to I2C */
i2c.i2cmcmd = I2CMCMD_CMD(START);/* Start Command */
/* Flush the CPU Write Buffers */
wbflush();
        status = sysChkI2CDone();
        if(status == ERROR)
return (ERROR);
/* Write Control Byte */
i2c.i2cdo = (unsigned int)(I2C_24LC64_CNTL_BYTE) ;
i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
        /* Flush the CPU Write Buffers */
        wbflush();
```

Notes

```
        status = sysChkI2CDone();
        if (status == ERROR)
                return (ERROR);
/* Address */
i2c.i2cdo = (unsigned int) ((address >> 8) & 0xFF) ;
i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
        /* Flush the CPU Write Buffers */
        wbflush();
        status = sysChkI2CDone();
        if (status == ERROR)
                return(ERROR);
i2c.i2cdo = (unsigned int) (address & 0xFF) ;
        i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
        /* Flush the CPU Write Buffers */
        wbflush();
        status = sysChkI2CDone();
        if (status == ERROR)
                return(ERROR);
/* Write Data */
i2c.i2cdo   =  byte ;
i2c.i2cmcmd =  I2CMCMD_CMD(WD) ;
        /* Flush the CPU Write Buffers */
        wbflush();
        status = sysChkI2CDone();
        if (status == ERROR)
                return(ERROR);
/* Generated Stop command allows the write byte to be written internally.
Wait for some time to let this internal operation finish */
status = sysI2CStop();
if (status != OK)
return(ERROR);

/* Wait for some time for the internal operation to finish */
WAIT(0x10000);
        return(OK);
}
/*******************************************************************
*
* sysI2CWritePageOpen - Opens a page(32 bytes) in I2C for subsequent writeByte
```

Notes

```
* operation.
*
* This function starts the write operation to the specified address. It is followed by
* up to 32 sysI2CWriteByte operations ending with sysI2CStop command.
*
* PARAMETERS:
*address: Address in I2C where page writes need to be done
*
* RETURNS: OK on completion else ERROR
*/
STATUS sysI2CWritePageOpen
(
unsigned short address  /* I2C Address */
)
{
        volatile unsigned int status;   /* used to check the I2C Status state */
 /* Send a Start command */
        i2c.i2cmcmd = I2CMCMD_CMD(START);/* Start Command */
        /* Flush the CPU Write Buffers */
        wbflush();
        status = sysChkI2CDone();
        if(status == ERROR)
                return(ERROR);
        /* Write Control Byte */
        i2c.i2cdo = (unsigned int )( I2C_24LC64_CNTL_BYTE) ;
        i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
        /* Flush the CPU Write Buffers */
        wbflush();
        status = sysChkI2CDone();
        if (status == ERROR)
                return(ERROR);
        /* Address */
        i2c.i2cdo = (unsigned int) ((address >> 8) & 0xFF) ;
        i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
        /* Flush the CPU Write Buffers */
        wbflush();
        status = sysChkI2CDone();
        if (status == ERROR)
                return(ERROR);
```

Notes

```
        i2c.i2cdo = (unsigned int) (address & 0xFF) ;
        i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
        /* Flush the CPU Write Buffers */
        wbflush();
        status = sysChkI2CDone();
        if (status == ERROR)
              return(ERROR);
return(OK);
}
/*******************************************************************
*
* sysI2CWriteInPage - Writes a byte in already opened I2C Page (32 sequential
* bytes starting at an address passed to sysI2CWritePageOpen).
*
* This function writes a byte. The address is maintained internally by I2C based
* on the address for which address the I2C write command was issued. This allows
* writing up to 32 bytes without feeding address each time before write operation.
*
* PARAMETERS:
*              byteData: Data to be written
*
* RETURNS: OK on completion else ERROR
*/
STATUS sysI2CWriteInPage
(
unsigned char byteData/* Data to be written */
)
     {
       volatile unsigned int status;   /* used to check the I2C Status state */

       /* Write Data */
       i2c.i2cdo   =  byteData;
       i2c.i2cmcmd =  I2CMCMD_CMD(WD);
       /* Flush the CPU Write Buffers */
       wbflush();
       status = sysChkI2CDone();
       if (status == ERROR)
              return(ERROR);
WAIT(1000);
```

Notes

```
        return(OK);
       }

/*********************************************************************
*
* sysI2CStop - Issues a stop command to the I2C Slave
*
* RETURNS: OK / ERROR
*/
STATUS sysI2CStop
(
void
)
{
volatile unsigned int status ; /* Check I2C Status */

/* Issue Stop Command */
i2c.i2cmcmd = I2CMCMD_CMD(STOP) ;
       /* Flush the CPU Write Buffers */
       wbflush();
       WAIT(0x2000);

       status = i2c.i2cms;
if(  ( (status & 0xF) == 0x1) ||
((status & 0xF) == 0x3) )
return(OK);
else {
/* Check for Error Condition */
if ((status & 0x8) != 0){
/* Clear Error Condition by Start Command */
status = sysI2CStart();
if (status != OK){
/* Something really bad happened. This should not happen.
   Try starting once again & return with error*/
   sysI2CStart( );
   return(ERROR);
}
}
/* OK we could start but caller should still be returned Error */
```

Notes

```
return(ERROR);
}
}
/*************************************************************************
*
* sysI2CStart - Issues a start command to the I2C Slave
*
* RETURNS: OK / ERROR
*/
STATUS sysI2CStart
      (
            void
     )
      {
      volatile unsigned int status ; /* Check I2C Status */
      i2c.i2cmcmd = I2CMCMD_CMD(START) ;
      /* Flush the CPU Write Buffers */
      wbflush();
      status = sysChkI2CDone();
      if (status == ERROR)
            return (ERROR);
      return(OK);
    }
/*************************************************************************
*
* sysI2CReadByte - Reads a byte from the specified location in the I2C eeprom
*
* PARAMETERS:
*address       Address from where data to be read
*
* RETURNS: Read Byte (as part INT32 read from register i2.i2cdi) or ERROR
*
*/
int sysI2CReadByte
(
unsigned short address/* I2C Address from where byte is read */
)
{
      volatile unsigned int status    ; /* used to check the I2C status state */
```

Notes

```
/* Send a Start Command */
sysI2CStart();
    /* Flush the CPU Write Buffers */
    wbflush();
    /* Write Control Byte */
    i2c.i2cdo = (unsigned int )( I2C_24LC64_CNTL_BYTE) ;
    i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
    /* Flush the CPU Write Buffers */
    wbflush();
    status = sysChkI2CDone();
    if (status == ERROR)
        return(ERROR);
    /* Address */
    i2c.i2cdo = (unsigned int) ( (address >> 8) & 0xFF) ;
    i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
    /* Flush the CPU Write Buffers */
    wbflush();
    status = sysChkI2CDone();
    if (status == ERROR)
        return(ERROR);
    i2c.i2cdo = (unsigned int) (address & 0xFF) ;
    i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
    /* Flush the CPU Write Buffers */
    wbflush();
    status = sysChkI2CDone();
    if (status == ERROR)
        return(ERROR);
/* Send Start command once again */
    sysI2CStart();
/* Send the Read Command Control Byte */
    i2c.i2cdo = (unsigned int)(I2C_24LC64_CNTL_BYTE | I2C_24LC64_RD_EN) ;
    i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
    /* Flush the CPU Write Buffers */
    wbflush();
    status = sysChkI2CDone();
    if (status == ERROR)
        return(ERROR);
    /* Read data */
    i2c.i2cmcmd = I2CMCMD_CMD(RD);
```

Notes

```
        /* Flush the CPU Write Buffers */
        wbflush();
        status = i2c.i2cms ;
        if ( !(status & I2CMS_D)){ /* Should have been done */
              /* Give some more time */
              WAIT(0x1000);
              status = i2c.i2cms ;
              if (!(status & I2CMS_D)){ /* waited long enough */
                                        /* return ERROR */
                   return(ERROR);
                  }
             }
        return((int)i2c.i2cdi);
}
/*********************************************************************
* sysI2CSeqReadStart: Sends the Read command to I2C for subsequent sequential reads.
* This function is used to start bulk read from I2C from a specified address EEPROM
* device. This function could be followed by up to (I2C Memory space in bytes -
* starting address) reads. Its left to the caller to ensure that no rollover (reading
* beyond the capacity of I2C) occurs.
*
* PARAMETERS :
*address       Starting address from where to begin read
*
* RETURNS   : OK / ERROR
*
*/
STATUS sysI2CSeqReadStart
(
unsigned short address   /* address of I2C beginning address */
)
{
        volatile unsigned int status  ; /* used to check the I2C status state */
        /* Send a Start command to I2C */
        i2c.i2cmcmd = I2CMCMD_CMD(START); /* Start Command */
        /* Flush the CPU Write Buffers */
        wbflush();
        status = sysChkI2CDone();
        if (status == ERROR)
```

Notes

```
        return(ERROR);
    /* Write Control Byte */
    i2c.i2cdo = (unsigned int )( I2C_24LC64_CNTL_BYTE)  ;
    i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
    /* Flush the CPU Write Buffers */
    wbflush();
    status = sysChkI2CDone();
    if (status == ERROR)
        return(ERROR);
    /* Address */
    i2c.i2cdo = (unsigned int) ((address >> 8) & 0xFF) ;
    i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
    /* Flush the CPU Write Buffers */
    wbflush();
    status = sysChkI2CDone();
    if (status == ERROR)
        return(ERROR);
    i2c.i2cdo = (unsigned int) (address & 0xFF) ;
    i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
    /* Flush the CPU Write Buffers */
    wbflush();
    status = sysChkI2CDone();
    if (status == ERROR)
        return(ERROR);
    i2c.i2cmcmd = I2CMCMD_CMD(START) ;
    /* Flush the CPU Write Buffers */
    wbflush();
    status = sysChkI2CDone();
    if (status == ERROR)
        return(ERROR);
    /* Read Command */
    i2c.i2cdo = (unsigned int)(I2C_24LC64_CNTL_BYTE | I2C_24LC64_RD_EN) ;
    i2c.i2cmcmd = I2CMCMD_CMD(WD) ;
    /* Flush the CPU Write Buffers */
    wbflush();
    status = sysChkI2CDone();
    if (status == ERROR)
        return(ERROR);
return(OK);
```

Notes

```
}
/*********************************************************************
* sysI2CSeqRead     Reads a byte from I2C
*
* This function reads a byte from the I2C internally maintained address. Call to
* this function is part of sequential bulk data read from I2C beginning at an
* address specified by the previous call to sysI2CSeqReadStart function.
*
* PARAMETERS : None
*
* RETURNS  : Read Data if successful ( as UINT32 ) else ERROR
*/
int sysI2CSeqRead
(
void
)
{
      volatile unsigned int status  ; /* used to check the I2C status state */
      /* Read data */
      i2c.i2cmcmd = I2CMCMD_CMD(RDACK);
      /* Flush the CPU Write Buffers */
      wbflush();
      status = sysChkI2CDone();
      if (status == ERROR)
            return(ERROR);
      WAIT(1000);
      return (i2c.i2cdi);
    }
/*********************************************************************
* sysI2CSeqReadLast     Reads the last byte in a sequential read from I2C
*
* PARAMETERS: None
*
* RETURNS   : Read Data if successful ( as INT32 ) else ERROR
*/
int sysI2CSeqReadLast
      (
            void
    )
```

Notes

```
        {
        volatile unsigned int status   ; /* used to check the I2C status state */
        /* Read data */
        i2c.i2cmcmd = I2CMCMD_CMD(RD);
        /* Flush the CPU Write Buffers */
        wbflush();
        /* Check if Done flag is set in I2C Status, Please note that the slave
          would not send the Ack for RD command */
        WAIT(100);
        status = i2c.i2cms ;
        if (!(status & I2CMS_D)){ /* Should have been done */
              /* Give some more time */
              WAIT(1000);
              status = i2c.i2cms ;
              if (!(status & I2CMS_D)){ /* waited long enough */
                    /* return ERROR */
                    return(ERROR);
                    }
              }
        return((int)i2c.i2cdi);
       }
/*********************************************************************
* sysI2CWritePage     Writes a page worth of data to I2C
*
* This function writes a page (32 bytes) of information to the I2C at the
* specified address.
*
* PARAMETERS:
*             address       address of I2C
*             string         Input string
*
* RETURNS   : OK / ERROR
*/
STATUS sysI2CWritePage
(
unsigned short address,  /* Address in the I2C */
          unsigned char* string    /* String pointer */
)
{
```

Notes

```
unsigned int status ; /* Used to check the status of I2C */
int i             ; /* loop count */
status = sysI2CWritePageOpen(address);
if (status != OK)
return(ERROR);

for(i=0;i<32;i++)
{
status = sysI2CWriteInPage(string[i]);
if (status != OK)
return(ERROR);
wbflush();
WAIT(0x2000);
}
/* Generate Stop to allow writing of data from the I2C internal
   buffer to the Eeprom */
sysI2CStop();
wbflush();
/* Perform Write Acknowledge Poll */
WAIT(0x100000) ;
return(OK);
}
/*******************************************************************
* sysI2CWriteBytes     Writes up to 31 bytes to I2C
*
* This function writes the input string to the specified I2C location. For all
* the writes, the Write Acknowledge command is used.
*
* PARAMETERS:
*             address       address of I2C
*             string        Input string
*             numOfBytes    number of bytes to be written
*
* RETURNS   : OK / ERROR
*/
STATUS sysI2CWriteBytes
(
unsigned short address,  /*address of I2C*/
unsigned char* string,   /* Input string */
```

Notes

```
unsigned int numOfBytes /* number of bytes to be written*/
)
      {
unsigned int status ; /* Used to check the status of I2C */
int i             ; /* loop count */
      status = sysI2CWritePageOpen ( address );
if (status != OK)
return(ERROR);
      for(i=0;i< numOfBytes;i++)
            {
                  status = sysI2CWriteInPage(string[i]);
if (status != OK)
return(ERROR);
wbflush();
WAIT(0x2000);
            }
      /* Generate Stop to allow writing of data from the I2C internal
        buffer to the Eeprom */
      sysI2CStop();
wbflush();
      /* Perform Write Acknowledge Poll */
      WAIT(0x100000) ;
return(OK);
    }
/*****************************************************************
* sysI2CReadString     Reads a string from I2C
*
* This function reads a string from the specified address of length numBytes.
*
* PARAMETERS:
*             address       address of I2C
*             string        Output string ( sizeof string > (numBytes+1))
*             numBytes      number of bytes to read
*
* RETURNS   : OK / ERROR
*/

STATUS sysI2CReadString
(
```

Notes

```
unsigned short address, /* address of I2C */
unsigned char* string , /* Output String  */
unsigned int   numOfBytes /* number of bytes to read */
)
        {
        unsigned int status ; /* Used to check the status of I2C */
        int i               ; /* loop count */
        /* Sanity check */
        if (numOfBytes == 0)  /*  Zero bytes to be written */
                return(ERROR);
        if (string == NULL) /* Null input string */
                return(ERROR);
/* Check address range */
        if ((address+(UINT16)numOfBytes) > (UINT16)I2C_MEM_SIZE)
                return(ERROR);
        status = sysI2CSeqReadStart(address);
/* check the returned status */
        if (status != OK)
return(ERROR);
        for (i=0 ;i < numOfBytes ; i++){
string[i] = sysI2CSeqRead();
                }
WAIT(0x2000);
/* Perform the last read operation */
string[i+1] = sysI2CSeqReadLast();
/* Append End of String */
string [i+2] = EOS ;
        return(OK);
        }
/*******************************************************************
* sysI2CWriteString     Writes the input string to I2C
*
* This function writes the input string to the specified I2C location. For all
* the writes, the Write Acknowledge command is used.
*
* PARAMETERS:
*               address       address of I2C
*               string        Input string
*
```

Notes

```
* RETURNS   : OK / ERROR
*/
STATUS sysI2CWriteString
(
unsigned short address, /* address of I2C */
unsigned char* string , /* Input string   */
unsigned int   numOfBytes /* Number of bytes */
)
{
        unsigned int status ; /* Used to check the status of I2C */
        int i               ; /* loop count */
unsigned int numOfPages, numOfRemWrites ;/* To determine no of pages
   & remaining byte writes */
unsigned int memI2CRef=0 ; /* To reference I2C memory */
unsigned int pageAlign, numOfPageAlignWrites=0 ; /* Used for data page
 aligning */
        /* Sanity check */
        if (string == NULL) /* Null input string */
                return(ERROR);
        /* Check address  */
        if (address > (UINT16)I2C_MEM_SIZE)
                return(ERROR);
        /* Check address range */
        if ((address+numOfBytes) > (UINT16)I2C_MEM_SIZE)
                return(ERROR);
/* Align the input data correctly on page boundary */
pageAlign = (unsigned int) (address & 0x1F) ;
numOfPageAlignWrites = 32 - pageAlign ;
        if (numOfPageAlignWrites > numOfBytes) { /* call writeBytes and return*/
status =sysI2CWriteBytes(address,&string[0],
 numOfBytes) ;
return(status);
}
else{ /* first write numOfPageAlign bytes */
status =sysI2CWriteBytes(address,&string[0],
                                        numOfPageAlignWrites) ;
/* bump up I2C memory address reference */
memI2CRef += numOfPageAlignWrites ;
}
```

Notes

```
/* Proceed with the remaining writes */
numOfPages = (numOfBytes - numOfPageAlignWrites) / 32 ;
numOfRemWrites = (numOfBytes - numOfPageAlignWrites) % 32 ;
/* Start writing the pages first */
for (i=0; i< numOfPages ; i++ ){
status = sysI2CWritePage((address + memI2CRef),&string[memI2CRef]);
if (status != OK)
return (ERROR);
memI2CRef += 32 ;
}
/* Now write the remaining bytes */
if (numOfRemWrites){
status = sysI2CWriteBytes((address + memI2CRef),
                                                (&string[memI2CRef]),
                                                 numOfRemWrites) ;
}
if ( status != OK )
return(ERROR);
return(OK);
}
 /*******************************************************************************
*  wbflush     :     Flushes the CPU write buffer
*
* This function loads a dummy value from the uncached space to flush out any pending
* CPU write cycles.
*
* PARAMETERS  :  NONE
* RETURNS         :  VOID
*/
void  wbflush()
{
volatile unsigned int   dummyData ;
volatile unsigned int * ptrToUncachedMem = (unsigned int*)0xA0000000 ;

/* Load a dummy data from the uncached location to flush CPU write buffers */
dummyData = *ptrToUncachedMem ;
} /* end of wbflush */
```

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01 Jan 2024)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.