Software Creates Compatible 64Kbit 2-Wire Serial EEPROMs

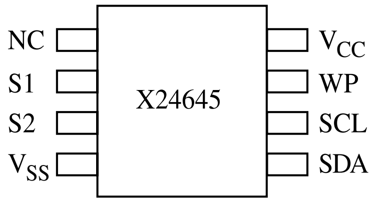
by Applications Staff

Introduction

In recent years, nonvolatile memory manufacturers have pushed the envelope on serial EEPROM performance. Devices now exist with higher densities, extra data protection features, and lower operating voltages, while at the same time consuming less power. In many applications, these newer generation devices are ideal because they satisfy important system requirements. Among these devices, 64Kbit 2-wire serial EEPROMs in 8-lead SOIC packages have been particularly well received. Unfortunately, not all manufacturers have implemented a common set of features within their devices. This causes problems for a designer who desires multiple sources. Although none of these devices could ever be considered as second sources for one another, there are some steps that can be taken by a designer to guarantee that any of these 64Kbit devices can be used interchangeably in a system.

In this note, we'll consider devices from Xicor (X24645), Atmel (AT24C64), and Microchip (24C65). For the addressing of the X24645, Xicor has implemented an approach that differs slightly from the more conventional addressing scheme found on lower density 2-wire devices (e.g. X24C02). The advantage is a simplification of the protocol for accessing the X24645. The AT24C64 and 24C65 adhere to the older standard, however in systems attempting to use multiple sources for 64Kbit devices, certain software modifications and pin connections could be made to ensure compatibility. In this note, the details of this implementation are explained and general purpose C code provided. The code was debugged using Turbo C® and Fig. 3 shows the simple test set-up.

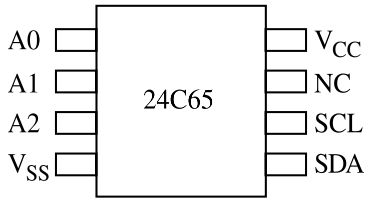
DIP/SOIC



Pin Descriptions

SCL	Serial Clock
SDA	Serial Data
S1	Device Select Input
$\overline{\text{S2}}$	Device Select Input
WP	Write Protect
NC	No Connect
V_{CC}	Supply
V_{SS}	Ground

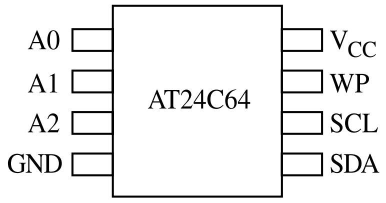
DIP/SOIC



Pin Descriptions

SCL	Serial Clock
SDA	Serial Data
A0-A2	Chip Select Inputs
NC	No Connect
V_{CC}	Supply
V_{SS}	Ground

DIP/SOIC



Pin Descriptions

SCL	Serial Clock
SDA	Serial Data
A0-A2	Address Inputs
WP	Write Protect
V_{CC}	Supply
GND	Ground

Slave Address

The first discrepancy encountered between Xicor's X24645 and the Microchip 24C65 or Atmel AT24C64 is the difference in slave addressing and the protocol for sending upper address bits to the device. Both the 24C65 and AT24C64 use an 8-bit slave address consisting of the device ID (1010), a 3-bit device address (A2,A1,A0), and the R/$\overline{W}$ bit. The 13-bit array address is transferred following the slave address by sending two additional bytes, containing 3 zeroes (000) or 3 don't cares (XXX) and the complete address. This sequence is similar to lower density devices available from all three manufacturers. Xicor's X24645 has a slave address and upper array address protocol consisting of an 8-bit slave address followed by a single address byte. The slave address consists of 2 device select bits (S2,S1), the 5 MSBs of the array address (A12,A11,A10,A9,A8), and the R/$\overline{W}$ bit. This allows for a 1-byte simplification of the software overhead when the X24645 is accessed.

In order to make these 3 devices functional in the same socket, pins 1, 2, and 3 must be tied to Vss. On power-up, the master should transmit (10101110) as the slave address. If the Xicor device is present, the master will receive an acknowledge. If the Microchip or Atmel devices are present, then there will not be an acknowledge. The software routine should then set a "X24645 detected" flag for later use.

Write Protection Register

The X24645 contains an internal Write Protect Register (WPR) that is used to control the state of the device. As soon as the X24645 is detected, a separate routine is used to correctly set the WPR. First the WEL bit is set, then the RWEL bit is set, and then the Block Protect (BP) and WPEN registers are set. If the BP registers are set to protect the upper 1/2 of the array, then control of the array write protection for both the Atmel and Xicor devices will behave similarly, depending on the WP pin (pin 7).

Array Write Protection

All three devices have write protection features, however guaranteeing compatibility with the Microchip device is a problem. In order to protect the upper 1/2 of the array for the Microchip part, a software sequence must be sent to the device. In order to implement this feature, we would have to assume that the WP input pin of the socket is being driven by a microcontroller, which could then also execute a special routine for the Microchip device. If the AT24C64 is actually present when this occurs, data corruption is possible unless the starting block for the Microchip device is defined with the B3 bit set (i.e. B3,B2,B1,B0 = 1XXX). Additionally, with the 24C65, once the write protection mechanism is set, it cannot be undone. This is in contrast to the Xicor and Atmel devices which allow for write protection control by the WP pin. Of special note is the scheme used on the X24645. By using a WPR with a Write Protect Enable bit (WPEN), a system can have the WP pin hard-wired HIGH without the array being protected. During system test, the E2PROM can be loaded with data and at this point, the WPEN bit can be set, which would "lock" the array write protection. The only way to alter the write protection at that point is to take WP LOW and/or reset the WPEN bit. Because of the difficulties, write protection on the Microchip 24C65 has not been implemented in the attached code, though simple modifications can be made to provide for it.

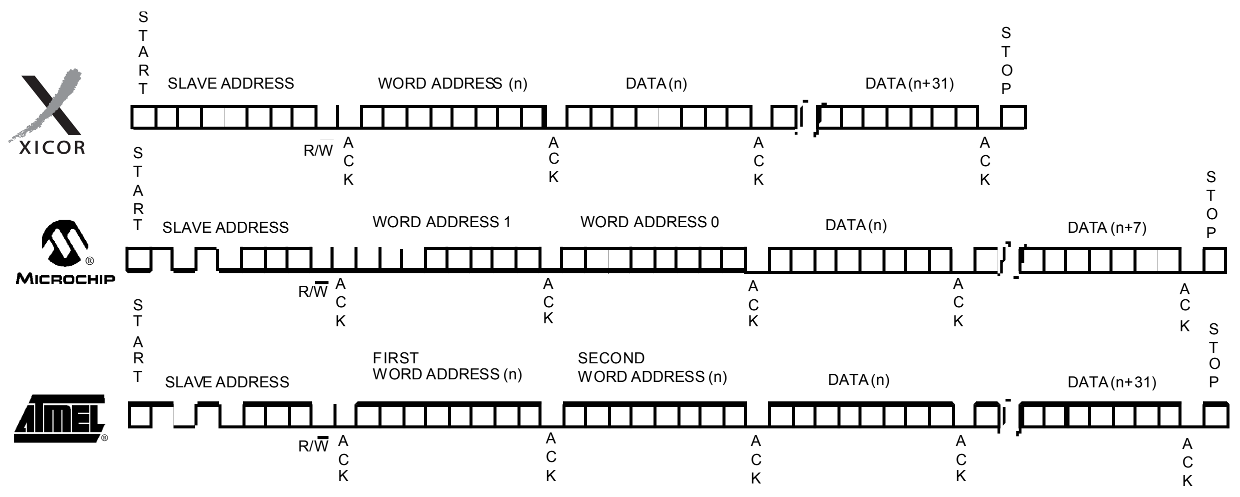


Figure 1. Page Write (Byte Write) operations for each device

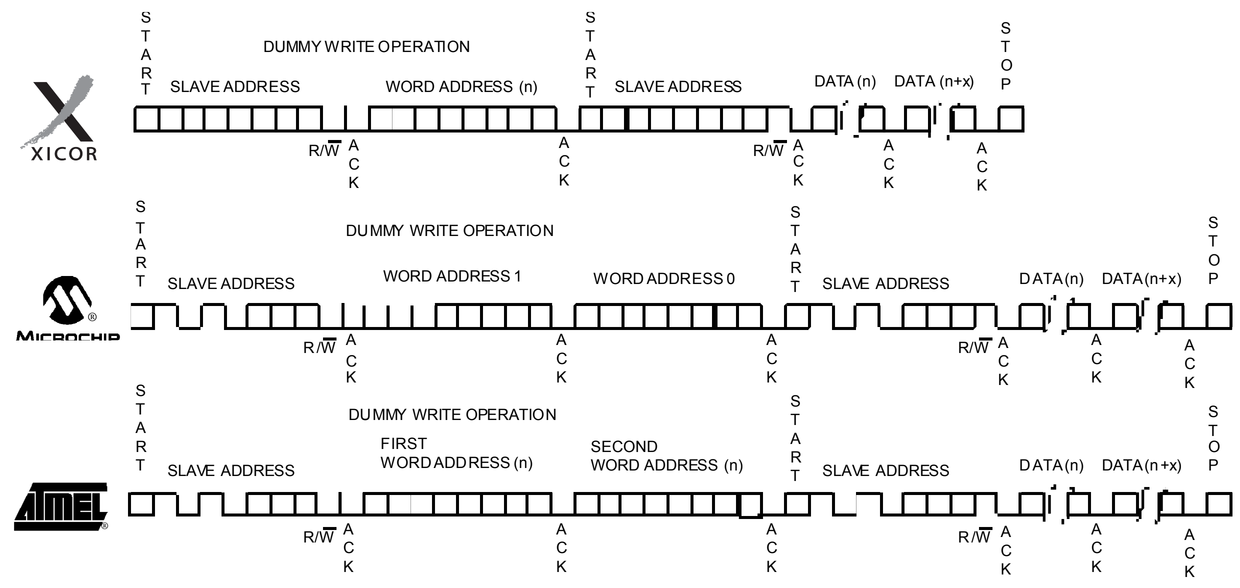


Figure 2. Random Sequential Read (or current address Sequential Read) operations for each device. For a current address read operation, disregard the dummy write operation.

Serial EEPROM Modes of Operation

Once software has detected the X24645, the necessary modifications to the protocol are made automatically. For 2-wire serial EEPROMs, there are really only two modes of operation, with all others being special cases of these modes: Page Write and Random Sequential Read. A Single Byte Write operation is merely a special case of a Page Write operation, where only 1 data byte is sent to the device. Likewise, a Single Byte Read operation is a special case of the Sequential Read operations. Because of the way that the Random Read protocol is implemented in the code, a Current Address Read operation can be seen to be a Random Read operation without the "dummy" address write preceding it. Note that the

X24645 does not internally increment its address counter after the last byte written, which differs from the behavior of the AT24C64 and 24C65. For example, write operations ending on the last byte of a page (e.g. $001F) followed by a current address read, will return data from different locations, depending on the device. For the X24645, data would be read from $001F. For the 24C65, it would be from $0018, but with the AT24C64, it would be from $0000. This should be taken into account when using the current address Sequential Read operation.

Endurance

Another peculiarity with the Microchip 24C65 device (as well as Microchip's 24LC65 and 24AA65 lower voltage versions of the 24C65) is its endurance specification. Although Microchip touts their EEPROMs as having more than a million cycles endurance on the 24C65, this only applies to a small portion of the memory array. A user defined 4Kbit block will provide 1,000,000 cycles, but the rest of the array is only specified as 10,000 cycles. Both Xicor and Atmel specify their devices at 100,000 cycles for every byte in the array. With Microchip, an average endurance of 71,875 endurance cycles per byte is perhaps a more appropriate spec. In reality, few systems require such endurance and this code does not attempt to change the high endurance block position from the Microchip default position. Again, simple modifications to the code are all that are required to implement this feature, if it is desirable to relocate this high endurance block.

Page Size

Though Xicor's X24645 and Atmel's AT24C64 have 32-byte pages, the Microchip 24C65 only has an 8-byte page. To compensate for this flaw, Microchip uses a 64-byte FIFO buffer internally to write consecutive pages. We can use this feature to make the Microchip device compatible with 32-byte page devices, however there could be some problems with the 24C65 due to wrap-around within the FIFO. Careful use of the seq_write() routine will prevent anomalous behavior. Never write more than 32 bytes on a page at a time and never attempt to utilize the wrap-around feature of the Xicor and Atmel devices when a Microchip device could be present. That should be sufficient to avoid any problems.

Conclusion

Though all three devices have been tested and will work with this generic code, the Xicor X24645 and Atmel AT24C64 clearly stand-out as the best design choices. The potential headaches associated with guaranteeing compatibility between the Microchip 24C65 software protocols (i.e. setting the high endurance block or the array write protection) and the protocols of the other devices would prevent the use of some advanced features on the 24C65, possibly limiting its usefulness in a system. Furthermore, the X24645 has some apparent advantages over the AT24C64 because of its more flexible BlockLock™ array write protection and the presence of the WPEN bit in the WPR, which makes this device more attractive for programming during system production.

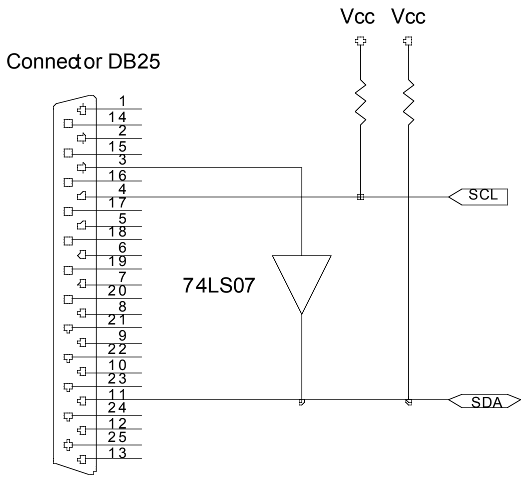


Figure 3: Simple interface between 2-wire serial EEPROMs and a parallel printer port on a PC

```
/******************************************************************/
/*                                                                */
/* Software for creating compatibility between Xicor's X24645,    */
/* Atmel's AT24C64, and Microchip's 24C65 2-wire serial EEPROMs.  */
/* This code contains all of the routines necessary for accessing */
/* these devices.  Simply use the seq_read() and seq_write()      */
/* routines for access, regardless of which device is present.    */
/*
/* Note that in this code, all attempts to read the status of the */
/* SDA pin (by the PC) assume that the SDA level is logically     */
/* inverted, due to this particular test set-up (Fig. 3). If this */
/* is not true for other hardware set-ups, then this code must be */
/* altered accordingly.                                           */
/*                                                         GHC IV */
/******************************************************************/

#include <stdio.h>
#include <stdlib.h>

int data_port = 0x378;              /* printer port output address*/
int status_port = 0x379;            /* printer port input address*/
unsigned char control = 0xFF;
unsigned char xicor_flag;           /* X24645 detected flag */
void SCL_high(){
      control = control | 0x04;     /* set SCL bit at port */
      outportb(data_port, control);
}

void SCL_low(){
      control = control & 0xFB;     /* reset SCL bit at port */
      outportb(data_port, control);
}

void SDA_high(){
      control = control | 0x02;     /* set SDA bit at port */
      outportb(data_port,control);
}

void SDA_low(){
      control = control & 0xFD;     /* reset SDA bit at port */
      outportb(data_port,control);
}

void start(){
      SDA_high();
      SCL_high();
      SDA_low();
      SCL_low();
}

void stop(){
      SDA_low();
      SCL_high();
      SDA_high();
}
```

```
unsigned char clock(){
unsigned char SDA_value;

      SCL_high();
      SDA_value = inportb(status_port);      /* read port */
      SDA_value = SDA_value & 0x80;          /* isolate SDA */
      SCL_low();
      SDA_value = SDA_value >> 7;            /* shift to LSB */
      return(SDA_value);                     /* and return data */
}

void ack(){
      SDA_low();
      clock();                               /* master sends acknowledge */
}

void nack(){
      SDA_high();                            /* master receives acknowledge */
      clock();
}
void out_byte(unsigned char byte){
char count;

      for (count = 0; count <= 7; count++){  /* send data to EEPROM */
            if ((byte & 0x80) == 0)          /* one bit at a time */
                  SDA_low();
            else
                  SDA_high();
            byte = byte << 1;                /* shift for next bit */
            clock();
      }
}

unsigned char get_byte(){
int count;
unsigned char byte,temp;

      byte = 0;
      for (count = 0; count <= 7; count++){  /* read data from EEPROM */
            byte = byte << 1;                /* one bit at a time */
            SDA_high();
            temp = clock();                  /* input bit from port */
            if (temp == 0)
                  byte = byte | 0x01;
      }
      return(byte);                          /* return data byte */
}

unsigned char serial_detect(){
unsigned char temp;

      start();
      out_byte(0xAE);                       /* send slave address */
      SDA_high();
      temp = clock();                      /* check for acknowledge */
      stop();
      return(temp);
}

void ack_poll(){
unsigned char poll;

      do {                                 /* ack polling loop */
            start();
            if (xicor_flag == 1)
                  out_byte(0x80);          /* Xicor slave address */
            else
                  out_byte(0xA0);          /* Microchip and Atmel slave address */
            SDA_high();
            poll = clock();                /* check for acknowledge */
      } while (poll == 0);
      stop();
}

void program_WPR(){
     start();
     out_byte(0xBE);                      /* X24645 slave address for location $1FFF */
     nack();
     out_byte(0xFF);
     nack();
     out_byte(0x02);                      /* set WEL bit */
     nack();
     stop();
     start();
     out_byte(0xBE);
     nack();
     out_byte(0xFF);
     nack();
     out_byte(0x06);                      /* set RWEL bit */
     nack();
     stop();
     start();
     out_byte(0xBE);
     nack();
     out_byte(0xFF);
     nack();
     out_byte(0x92);                     /* set WPEN, BP1, and BP0 bits */
     nack();
     stop();
     ack_poll();
}

void dummy_write(int addr){
int temp;

      if (xicor_flag == 1){
            temp = (((addr >> 7) & 0x3E) | 0x80);     /* construct X24645 */
                                                      /* slave address */
            start();
            out_byte(temp);
            nack();
            temp = (addr & 0xFF);            /* construct X24645 address byte */
            out_byte(temp);
            nack();
      }
      else{
            start();
            out_byte(0xA0);                  /* Microchip and Atmel slave address */
            nack();
            temp = ((addr & 0xFF00) >> 16);  /* construct first address byte */
            out_byte(temp);
            nack();
            temp = addr & 0xFF;              /* construct second address byte */
            out_byte(temp);
            nack();
      }
}

/*********************************************************************/
/*                                                                   */
/* Sequential read routine that handles all read operations.  For   */
/* current address reads, set current != 0 and addr as a don't care */
/* when calling the routine, otherwise addr = starting address and  */
/* *bytes points to buffer where data is to be stored for later     */
/* use. no_bytes is the number of bytes to be sequentially read     */
/* from the EEPROM.                                                  */
/*                                                                   */
/*********************************************************************/

void seq_read(int current,int no_bytes,int addr,unsigned char *bytes){
int n, temp;

      if (current == 0)                             /* random read? */
            dummy_write(addr);                      /* yes, send address */

      if (xicor_flag == 1){                         /* is it the X24645? */
            start();                                /* yes */
            temp = (((addr >> 7) & 0x3E) | 0x81);   /* construct slave address */
            out_byte(temp);
            nack();
      }
      else{
            start();                          /* no */
            out_byte(0xA1);                   /* slave address for Atmel and Microchip */
            nack();
      }
      for (n = 0;n < no_bytes - 1; n++){  /* sequentially read data */
            bytes[n] = get_byte();  /* in loop and send acknowledges */
            ack();
      }
      bytes[no_bytes - 1] = get_byte();  /* for last databyte, don't */
      stop();  /* send acknowledge */
}
```

```
/*******************************************************************/
/*                                                                 */
/* Sequential write routine handles all write operations. no_bytes */
/* is number of bytes to write, starting at address (addr), and    */
/* take the data from the buffer pointed to by *bytes.             */
/*                                                                 */
/*******************************************************************/

void seq_write(int no_bytes,int addr,unsigned char *bytes){
unsigned char temp;
int n;

      dummy_write(addr);                /* send address to EEPROM */
      for (n = 0;n < no_bytes; n++){
            out_byte(bytes[n]);         /* loop and send data bytes */
            nack();
      }
      stop();                           /* begin nonvolatile write cycle */
      ack_poll();                       /* poll for cycle completion */
}

/*****************************************************/
/*                                                   */
/* Simple program to demonstrate these routines.     */
/*                                                   */
/*****************************************************/

main(){
unsigned char data1[]={31,32,33,34,35};       /* buffer of data bytes */
unsigned char data2[]={41,42,43,44,45};       /* buffer of data bytes */
unsigned char data3[512];                     /* buffer to store bytes from EEPROM */
unsigned char data4[512];                     /* buffer to store bytes from EEPROM */

      xicor_flag=serial_detect();             /* check for X24645 */
      if (xicor_flag==1)                      /* is it X24645? */
      program_WPR();                          /* yes, set up WPR */
      seq_write(5,59,&data1);               /* page write ending on last byte of page */
      seq_write(5,64,&data2);               /* page write beginning on 1st byte of next page */
      seq_read(0,6,59,&data3);              /* random read across page boundaries (6 bytes) */
      seq_read(1,4,0,&data4);               /* current address read of remaining 4 bytes */
}
```