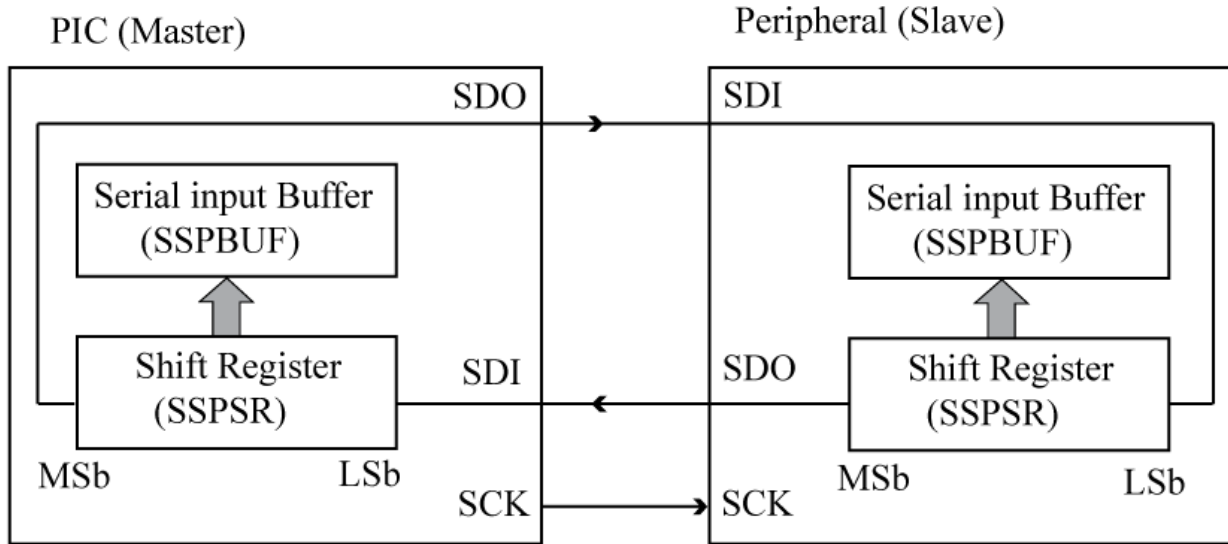


# Synchronous Serial IO

- Send a separate clock line with data
  - SPI (serial peripheral interface) protocol
  - I<sup>2</sup>C (or I2C) protocol
- Encode a clock with data so that clock be extracted or data has guaranteed transition density with receiver clock via Phase-Locked-Loop (PLL)
  - IEEE Firewire (clock encoded in data)
  - USB (data has guaranteed transition density)

# Serial Peripheral Interface (SPI)

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

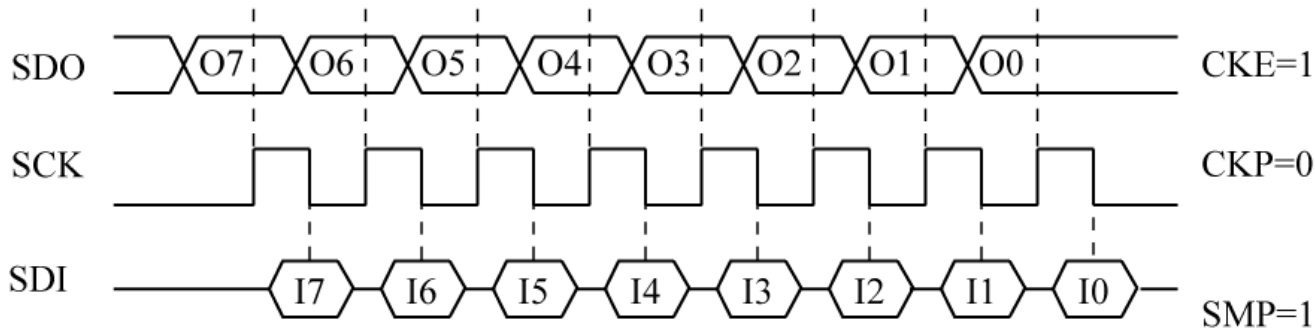


SDI: data in

SDO: data out

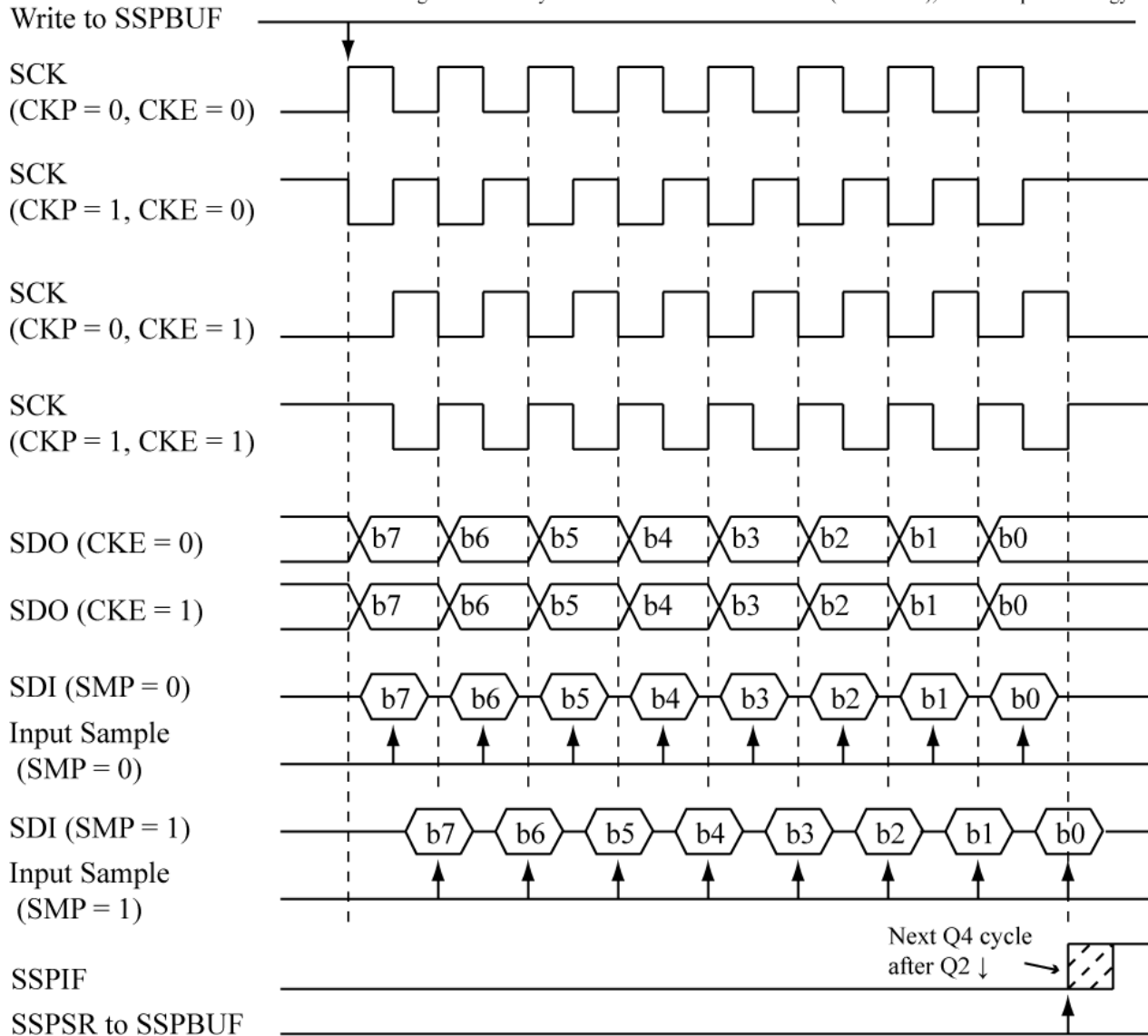
SCK: clock

Data sent MSb first; received data is clocked in as transmitted data is clocked out

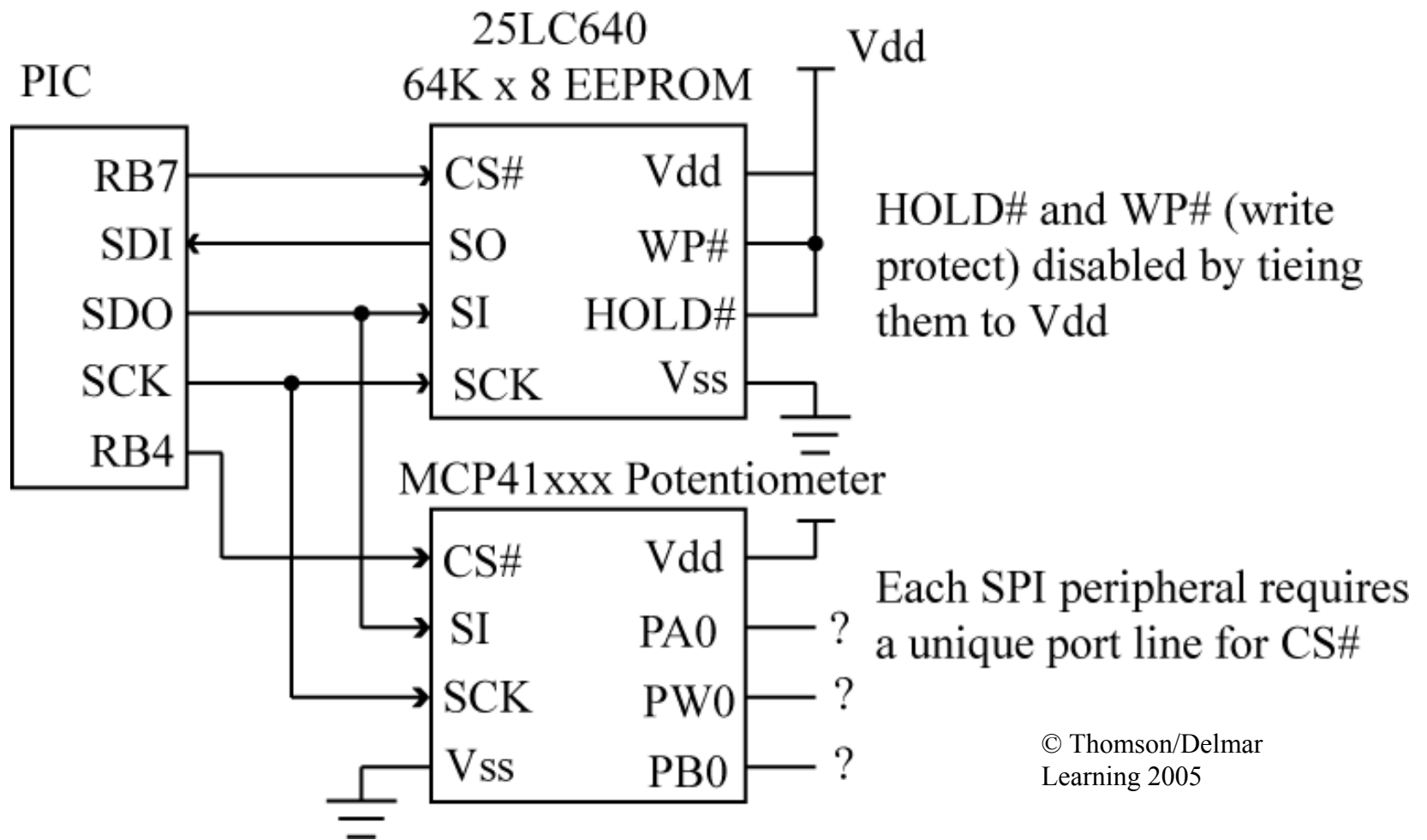


© Thomson/Delmar Learning 2005

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.



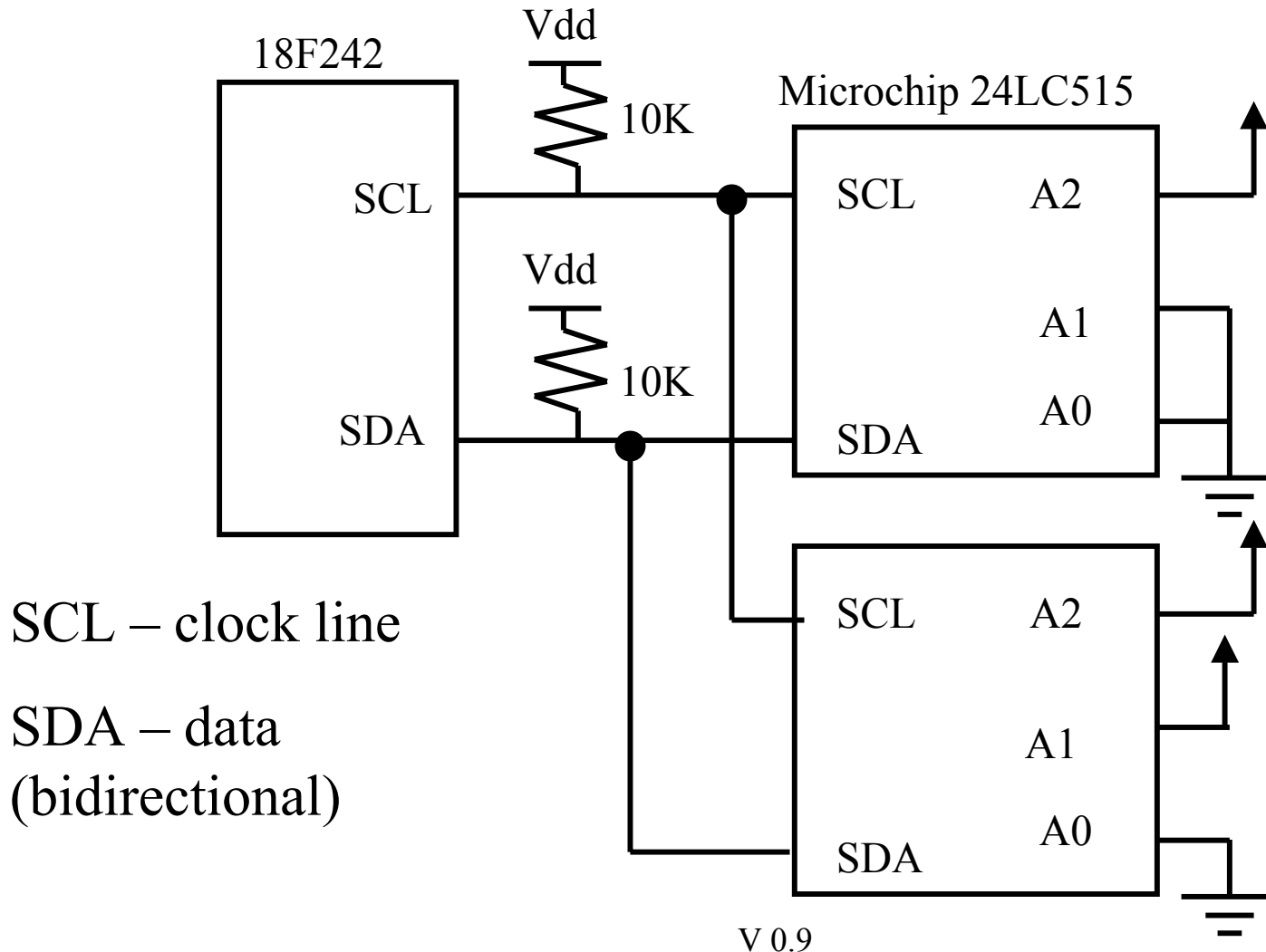
CKE configuration bit allows either falling or rising edge of clock to be used, while CKP selects clock polarity. The SMP bit determines if the SDI input is sample in middle or end of the clock period.



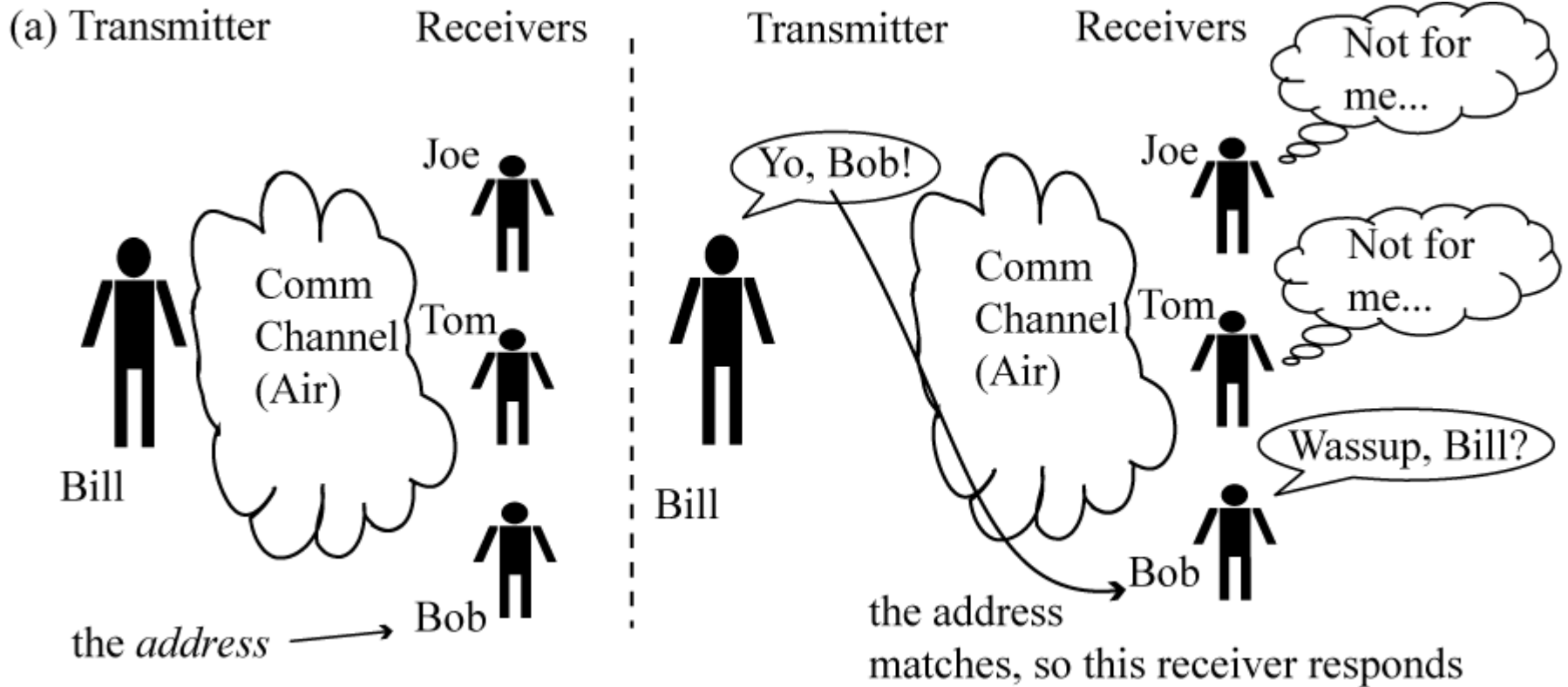
Multiple SPI peripherals each require a separate chip select line via parallel port line. We will concentrate on the I<sup>2</sup>C serial bus as it does not require use of chip selects.

# I<sup>2</sup>C (Inter-Integrated-Circuit) **Bus**

I<sup>2</sup>C is a two wire serial interface.



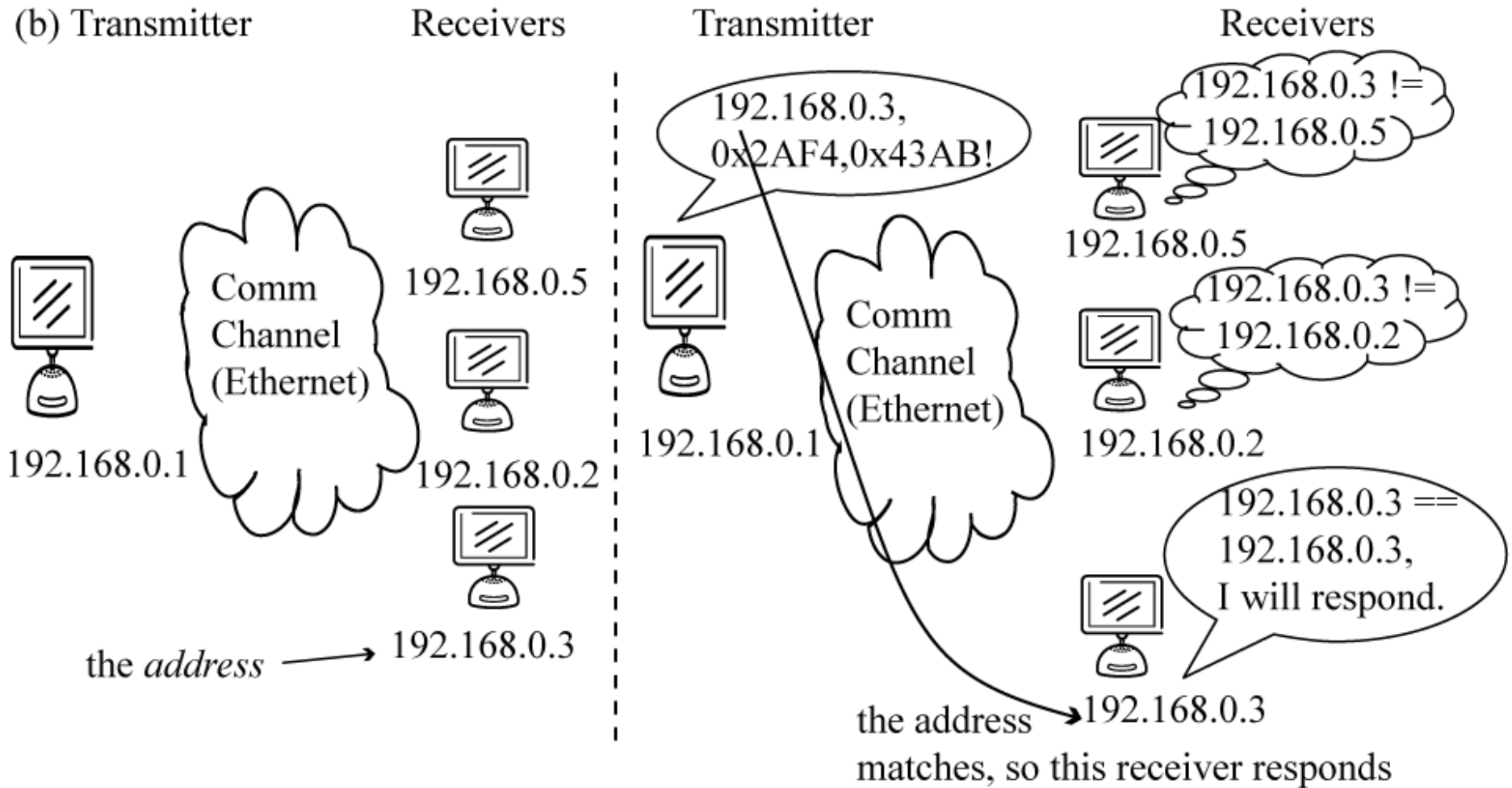
# What is a bus??



© Thomson/Delmar  
Learning 2005

One transmitter over a common channel  
to one or more receivers

# Ethernet is an example of a bus

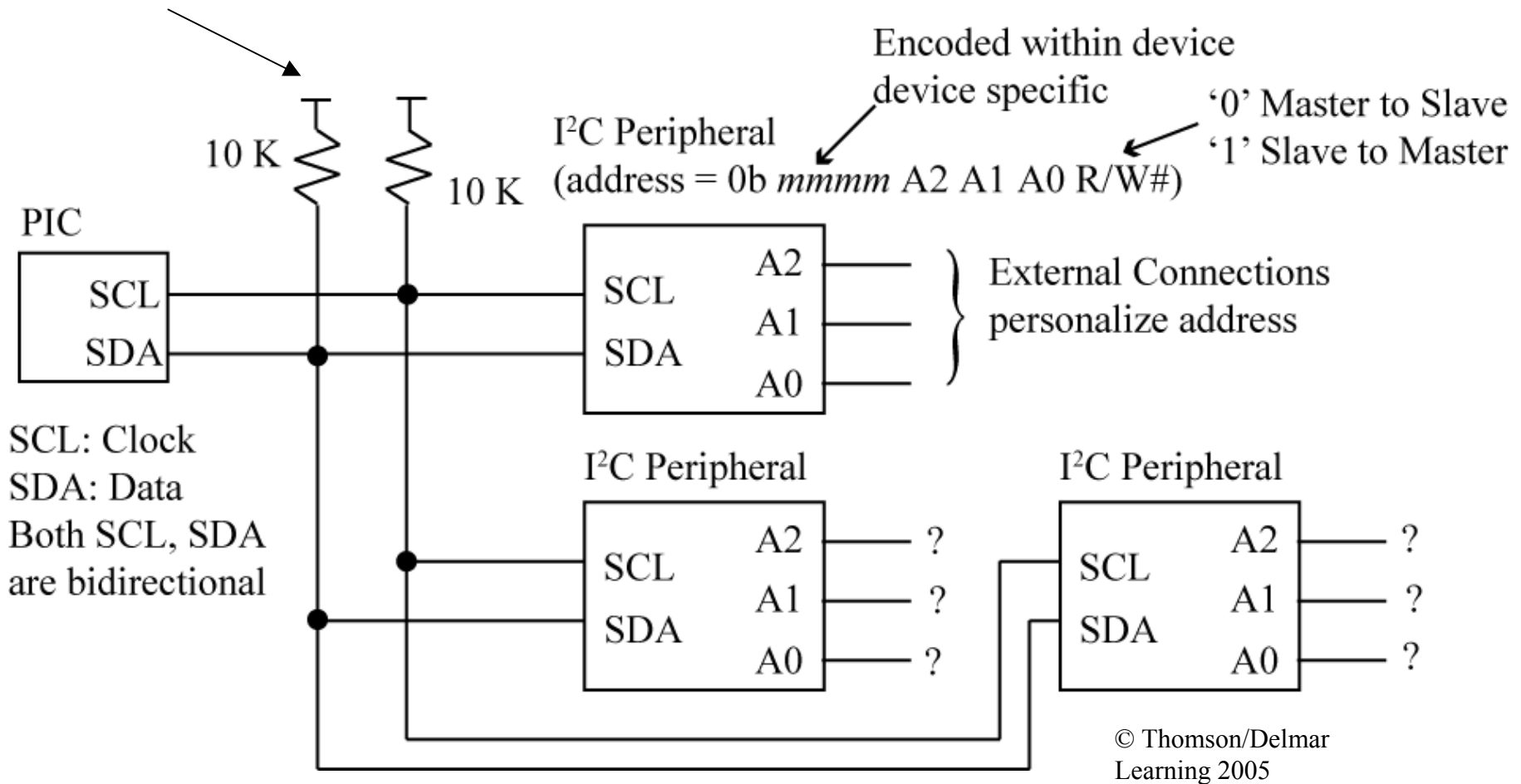


# I<sup>2</sup>C Features

- Multiple receivers do not require separate select lines as in SPI
  - At start of each I<sup>2</sup>C transaction a 7-bit device address is sent
  - Each device listens – if device address matches internal address, then device responds
- SDA (data line) is bidirectional, communication is half duplex
- SDA, SCLK are open-drain, require external pullups
  - Allows multiple bus masters (will discuss this more later).

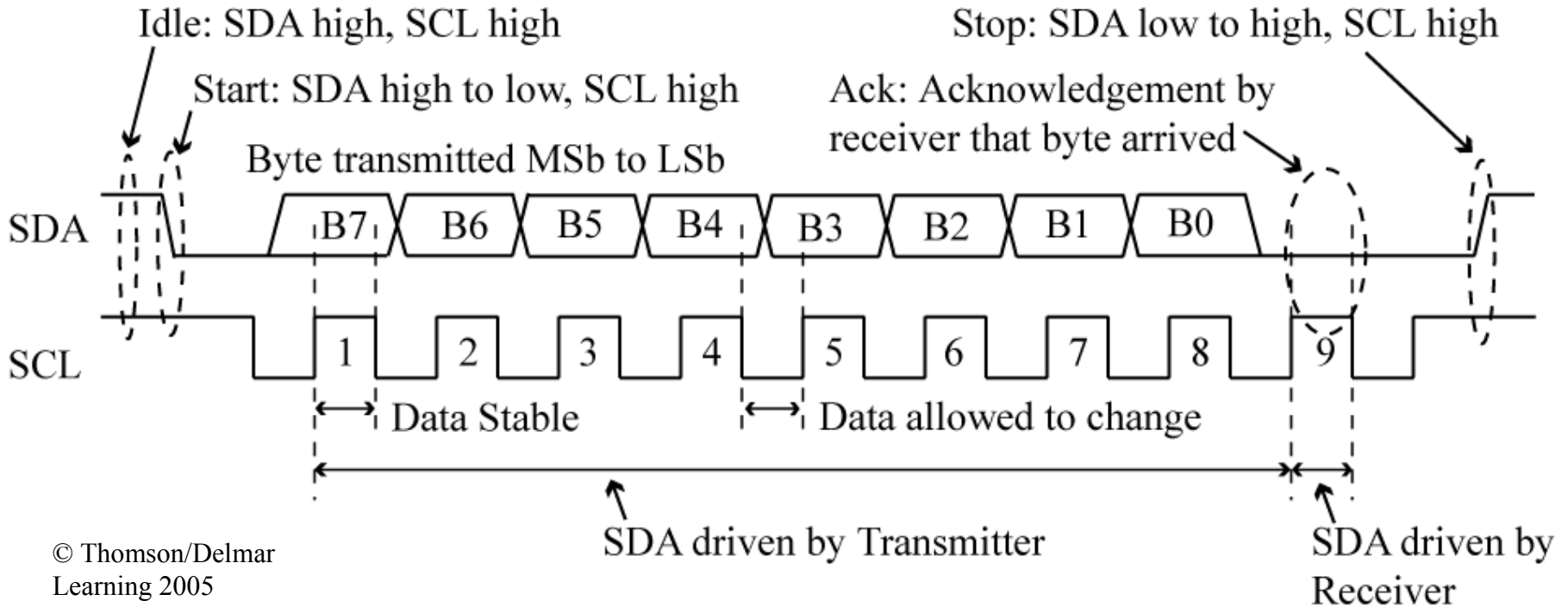
pullups are needed

# I2C Bus Addressing



No chip selects needed!!!!

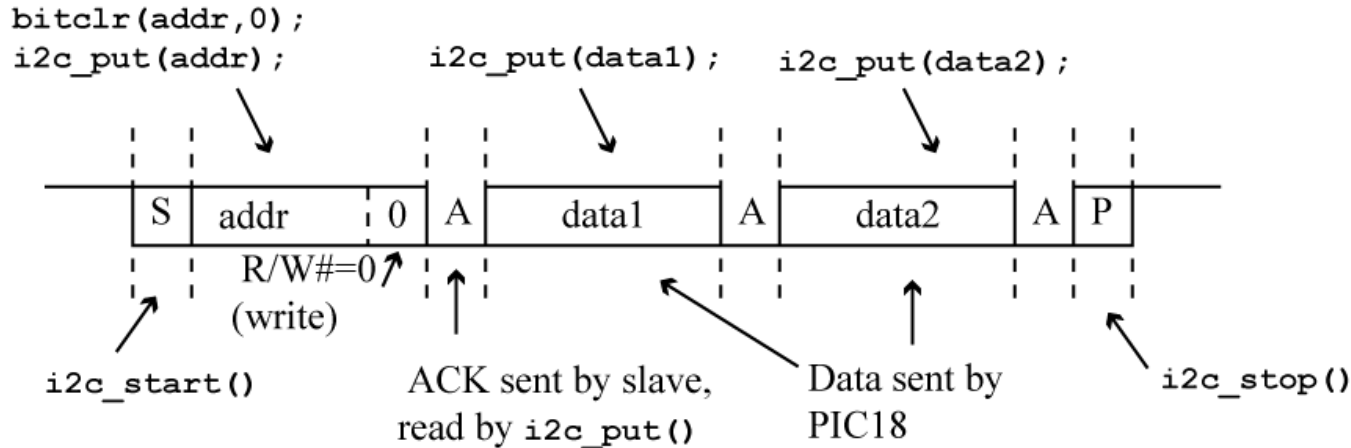
# I<sup>2</sup>C Bus Transfer



Multiple bytes sent in a transaction; every 8 bits has a 9<sup>th</sup> bit that is an acknowledge.

# Write (master to slave)

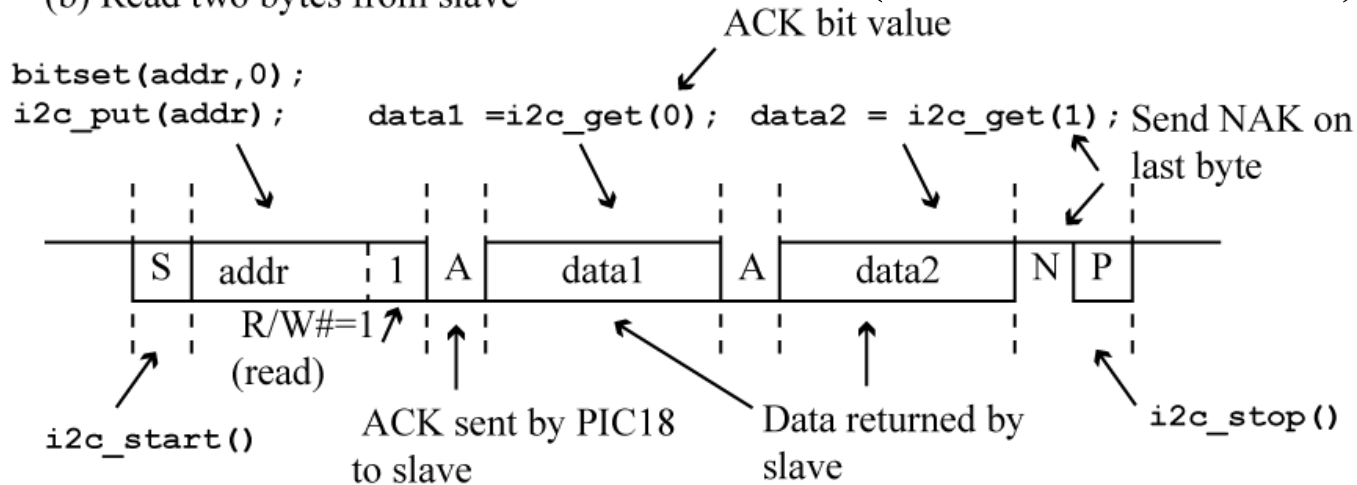
(a) Write two bytes to slave



Master initiates all transactions, read or write.

(b) Read two bytes from slave

# Read (master from slave)



# Example: I<sup>2</sup>C Serial EEPROM

Will use the Microchip 24LC515 Serial EEPROM to discuss I<sup>2</sup>C operation.

The 24LC515 is a 64K x 8 memory. This would require 16 address lines, and 8 data lines if a parallel interface was used, which would exceed the number of available IO pins our PIC18F242!!!

Putting a serial interface on a memory device lowers the required pin count.

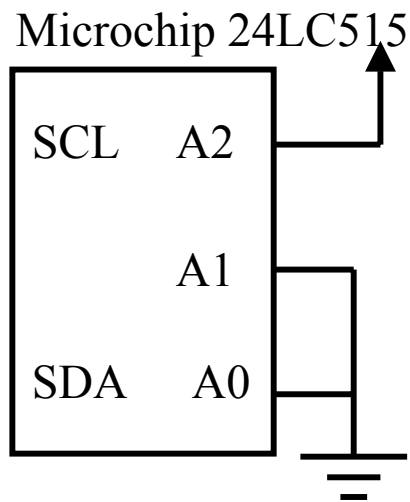
Reduces speed since data has to be sent serially, but now possible to add significant external storage to a low pin-count micro controller.

# I<sup>2</sup>C Device Addressing

Each I2C device has either a 7-bit or 10-bit device address.

We will use an I2C EEPROM and an I2C DAC (Digital-to-Analog Converter, MAX517) in lab. Both of these devices have a 7-bit address.

Upper four bits are assigned by device manufacturer and are hardcoded in the device. Lower three bits are used in different ways by manufacturer.



LC515 control byte (contains slave address):

7	6	5	4	3	2	1	0
1	0	1	0	B0	A1	A0	$\overline{\text{R/W}}$

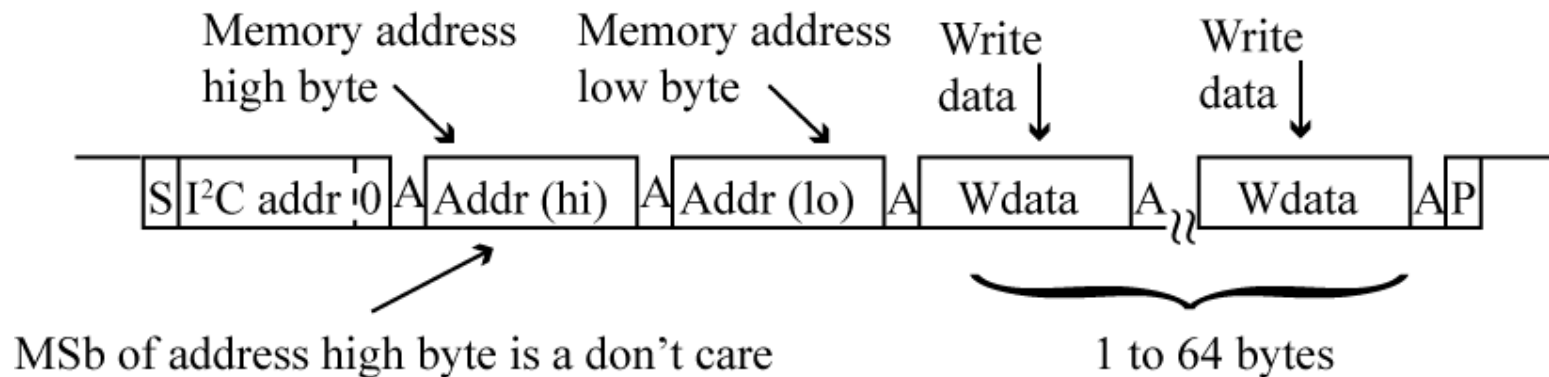
$\overline{\text{R/W}} = 1$   
for read, 0  
for write.

‘B0’ is block select (upper/lower 32K). A1, A0 are chip selects, four devices on one bus.

# Write Operation – can send up to 64 bytes

- Send up to 64 bytes, then perform write
  - Send starting address, followed by 64 bytes
  - After 64 bytes sent, wait 5 ms for write to complete
  - Much faster than individual writes

## Write Operation



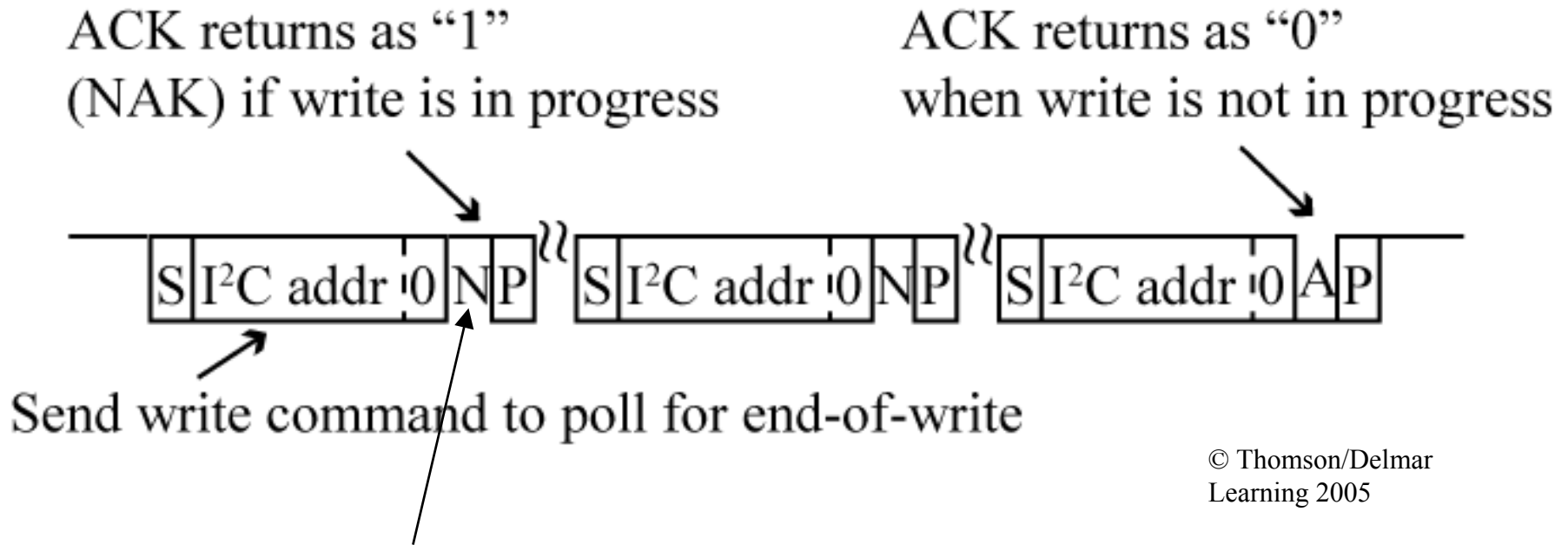
**Address should be on a page boundary when writing a block of 64 bytes.** For page size = 64 = 0x40, starting address should be a multiple of 64. A stop condition halts the write.

# Speed Comparison

- Assume a 400 Khz I<sup>2</sup>C bus, 2.5 us clock period (2.5 e-6)
- Writing one byte at a time:
  - 9 bit transmission = 2.5 us \* 9 = 22.5 us
  - 5 ms + 22.5 us\* 4 (control,addhi,addlo,data) =5.09 ms
  - For 64 bytes = 325 ms approximately, not counting software overhead.
- Writing 64 bytes at a time
  - 67 bytes total (control, addhi, addlo, data)
  - 5 ms + 67 \* 22.5 us = 6.5 ms!!!

# Polling for end-of-write

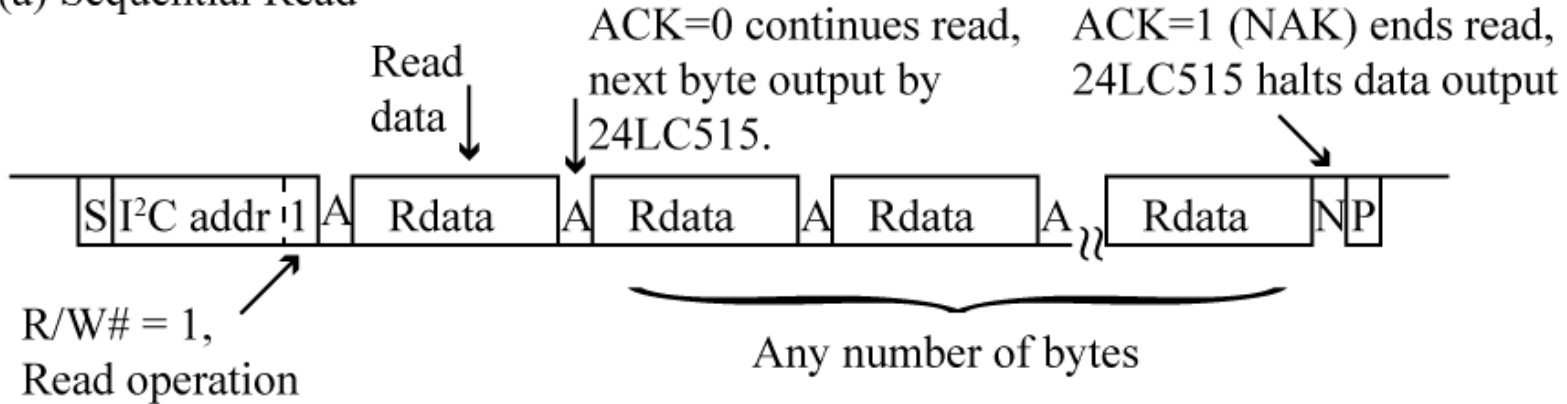
Timing on write is guaranteed to finish after 5 ms. But can end sooner; to determine if write finished use polling method.



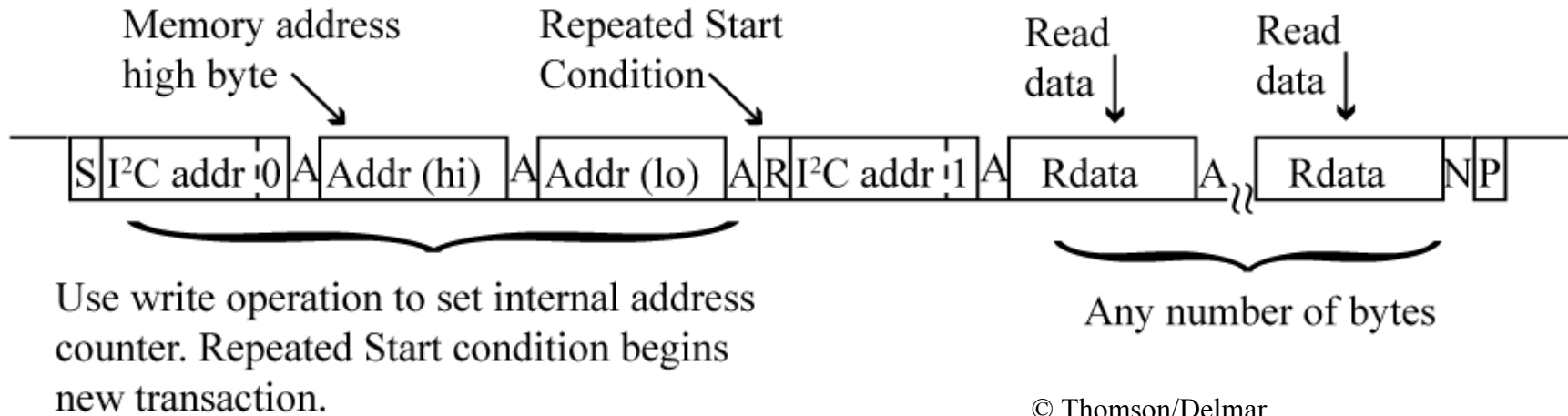
A NAK means device is still busy with last write.

# Read Operations

## (a) Sequential Read



## (b) Random Read



© Thomson/Delmar  
Learning 2005

# PIC18Fxx2 I<sup>2</sup>C Registers

- Synchronous Serial Port on PIC18Fxx2 implements I<sup>2</sup>C
- Registers are:
  - SSPCON – control register - we will always set this to 0x28 which enables I<sup>2</sup>C MASTER mode.
  - SSPCON1 – control register - used to initiate a START/STOP conditions, indicates if ACK has been received
  - SSPSTAT – status register – check this to see if byte finished transmitting, or byte has been received
  - SSPBUF – read/write to this register for data transfer
  - SSPADD – used to control clock rate

# I<sup>2</sup>C on the PIC18xx2

- Will always use master mode on the PIC18Fxx2
  - This means that the PICFxx2 will always initiate all I<sup>2</sup>C bus activity
- To set I<sup>2</sup>C clock rate, write 8-bit value to the SSPADD register
  - Clock rate =  $F_{osc}/(4 * (SSPADD+1))$
- I<sup>2</sup>C standard defines 100 KHz and 400 KHz but in reality just about any clock rate from DC to 400 KHz works

Clock Rate formula in SSPCON1 description, page 136 of datasheet (page 138 PDF page), section 15-4 of datasheet

# Lab #9: Read/Write to Serial EEPROM

- Lab #9 has you read/write to a Serial EEPROM via the I<sup>2</sup>C bus
- The files *i2cmsu.h*, *i2cmsu.c* define interface subroutines for the I2C bus
- This file *i2c\_memutil.c* has subroutines for random read/write, block read/write for the serial eeprom
- The file *i2cmemtst.c* tests uses the subroutines to read/write data to the serial EEPROM.

# i2cmsu.c Subroutines

- `i2c_idle()` – wait for idle condition on I2C bus
- `i2c_start()` – send a START and wait for START end
- `i2c_stop()` – send a STOP and wait for STOP end
- `i2c_ack(ackbit)` – send ackbit as acknowledge
- `i2c_put(byte)` – write byte to I2C, wait for ack, if ACK (=1) not returned then reset.
- `i2c_put_noerr(byte)` – write byte to I2C, wait for ack, return value of the ack bit that was returned
- `i2c_get(ackbit)` – get a byte from I2C bus and send ackbit as acknowledge
- `i2c_init(bitrate)` – initialize I2C mode, write bitrate to SSPADD register

# Watchdog Timer Use

The I<sup>2</sup>C subroutines assume the watchdog timer is enabled, so the statement:

```
asm("clrwdt");
```

is used in subroutines to clear the watchdog timer.

# i2c\_idle()

```
i2c_idle(){ // wait for idle condition
  unsigned char byte1;
  unsigned char byte2;
  asm("clrwdt");
  i2c_errstat = I2C_IDLE_ERR;
  do {
    // byte1 has R/W bit.
    byte1 = SSPSTAT & 0x04;
    byte2 = SSPCON2 & 0x1F;
  }while (byte1 | byte2);
  asm("clrwdt");
  i2c_errstat = 0;
}
```

R/W not in progress.

I<sup>2</sup>C interface is idle if R/W#, SEN, RSEN, PEN, RCEN, and ACKEN bits are all clear.

Check if lower 5 bits are all '0' indicating that Start, Stop, Acknowledge sequences are all idle.

# i2c\_start()/i2c\_stop()

```
i2c_start() {  
    i2c_idle();  
    i2c_errstat = I2C_START_ERR; ← Remember this function for error tracking  
    SEN = 1; // initiate start  
    while (SEN); // wait until start finished } Do START condition. If WDT  
    asm("clrwdt"); } expires, track error with i2c_errstat.  
    i2c_errstat = 0; ← Clear variable used for tracking function calls.  
}
```

---

```
i2c_stop() {  
    i2c_idle();  
    i2c_errstat = I2C_STOP_ERR;  
    PEN=1; // initiate stop, PEN=1  
    //wait until stop finished  
    while (PEN);  
    asm("clrwdt");  
    i2c_errstat = 0;  
}
```

} Perform Stop Condition

# i2c\_put()

```
unsigned char i2c_put(unsigned char byte ) {
    i2c_errstat = I2C_PUT_ERR;
    SSPIF = 0;      //clear interrupt flag
    SSPBUF = byte; // write byte
    while(!SSPIF); // wait for finish an ack
    i2c_errstat = 0;
    asm("clrwdt");
    if (ACKSTAT) {
        //no acknowledge returned, so reset
        i2c_errstat = I2C_MISSACK_ERR;
        asm("reset");
    }
    return(0);
}
```

} Initiate transmit by writing byte to SSPBUF register.  
SSPIF set when transmit is complete.

} If returned ACK bit is “1”, set error variable and do software reset.

If device does not responds with ack bit = ‘1’ (a NAK), then execute a software reset as this is usually an error condition. Use `i2c_put_noerr()` if want ack bit value returned with no reset.

# i2c\_get()

```
unsigned char i2c_get(unsigned char ackbit) {
    unsigned char byte;

    i2c_errstat = I2C_GET_ERR;
    RCEN = 1; //initiate read event
    while(RCEN); // wait until finished
    asm("clrwdt");
    while (!BF); //also check buffer full
    asm("clrwdt");
    byte = SSPBUF; // read data
    i2c_errstat = 0;
    i2c_ack(ackbit);
    return (byte);
}
```

} Configure for reception and wait for byte to be received.

} Read byte from SSPBUF and send acknowledgement

Send 'ackbit' as value of acknowledgement bit (1=NAK, 0 = ACK)

# i2c\_ack(unsigned char ackbit)

```
i2c_ack(unsigned char ackbit){  
    // send acknowledge  
    asm("clrwdt");  
    ACKDT = ackbit;  
    if (ackbit)    i2c_errstat = I2C_NAK_ERR;  
    else i2c_errstat = I2C_ACK_ERR;  
    //initiate acknowlege cycle  
    ACKEN = 1;  
    // wait until acknowledge cycle finished  
    while(ACKEN);  
    asm("clrwdt");  
    i2c_errstat = 0;  
}
```

Set ACK bit value

Start ACK cycle

Wait for finish of  
ACK cycle

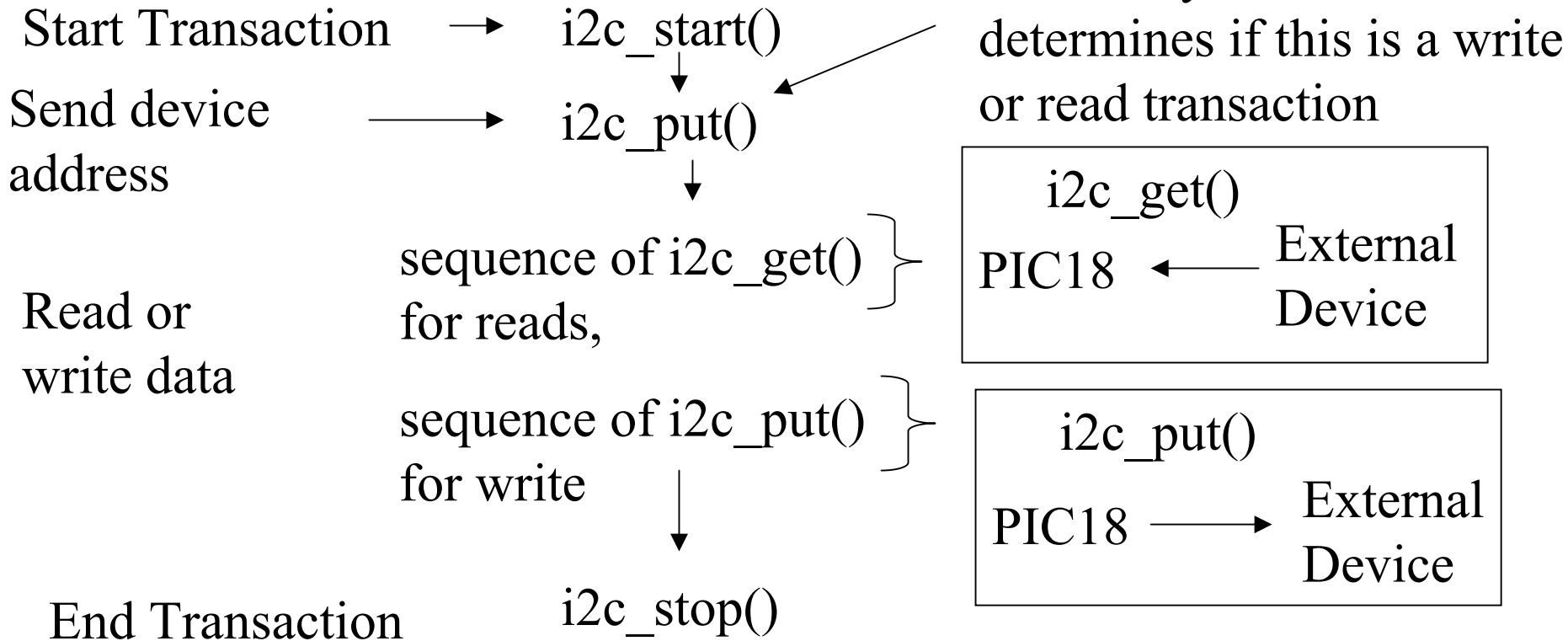
# i2c\_init()

```
i2c_init(char bitrate) {  
    // enable I2C Master Mode  
    SSPM3 = 1;  
    SSPM2 = 0;  
    SSPM1 = 0;  
    SSPM0 = 0;  
    SSPADD = bitrate; // set bus clk speed  
    SSPEN = 1;  
    bitset(TRISC,3);  
    bitset(TRISC,4); // SDA, SCL pins are inputs  
    SSPIF = 0; // clear SPIF bit  
    i2c_errstat = 0; // clear error status  
}
```

Initialize I<sup>2</sup>C module for master mode communication and set I<sup>2</sup>C bus speed.

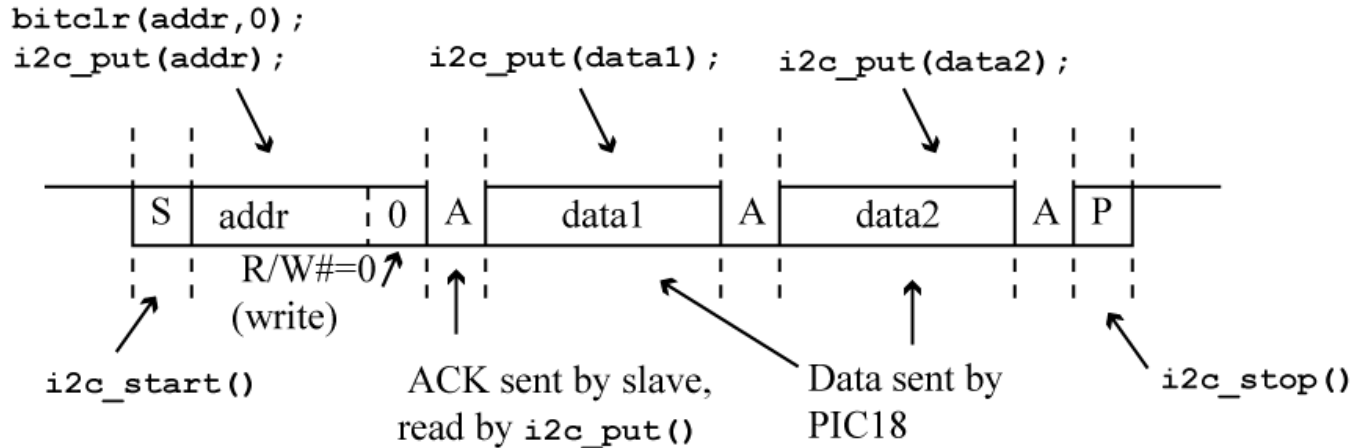
# An I<sup>2</sup>C Transaction

An I<sup>2</sup>C transaction is an exchange of data between the PIC18 and an external device over the I<sup>2</sup>C bus. All transactions use these calls:



# Write (master to slave)

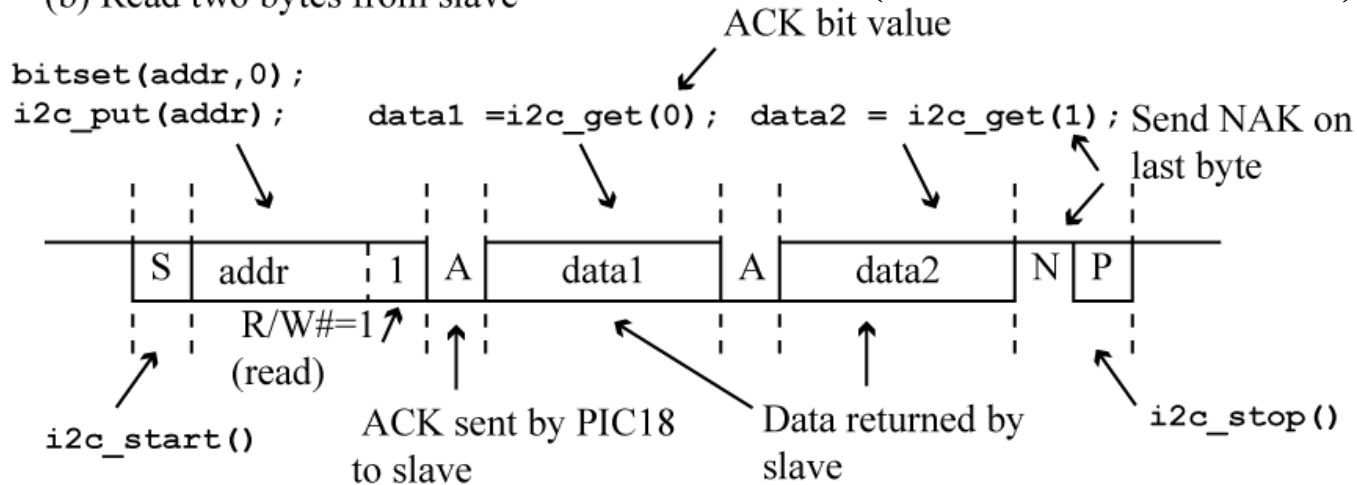
(a) Write two bytes to slave



Master initiates all transactions, read or write.

(b) Read two bytes from slave

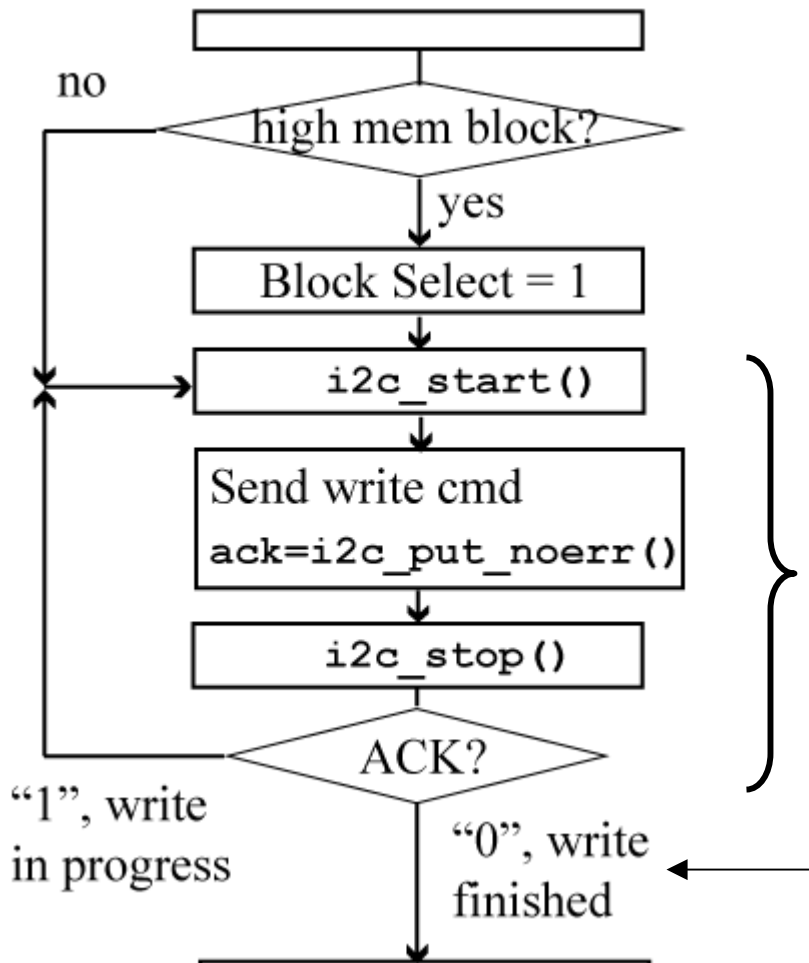
# Read (master from slave)



# 24LC515 EEPROM Utility Functions

- In file *i2c\_memutil.c*
- `i2c_memwrite(char i2caddr, unsigned int addr, volatile unsigned char *buf)`
  - write 64 bytes in *buf* to address *addr* to EEPROM with I2C address byte *i2caddr*
- `i2c_memread(char i2caddr, unsigned int addr, volatile unsigned char *buf)`
  - read 64 bytes from address *addr*, return bytes in *buf*. Use address I<sup>2</sup>C byte *i2caddr* to talk to EEPROM.

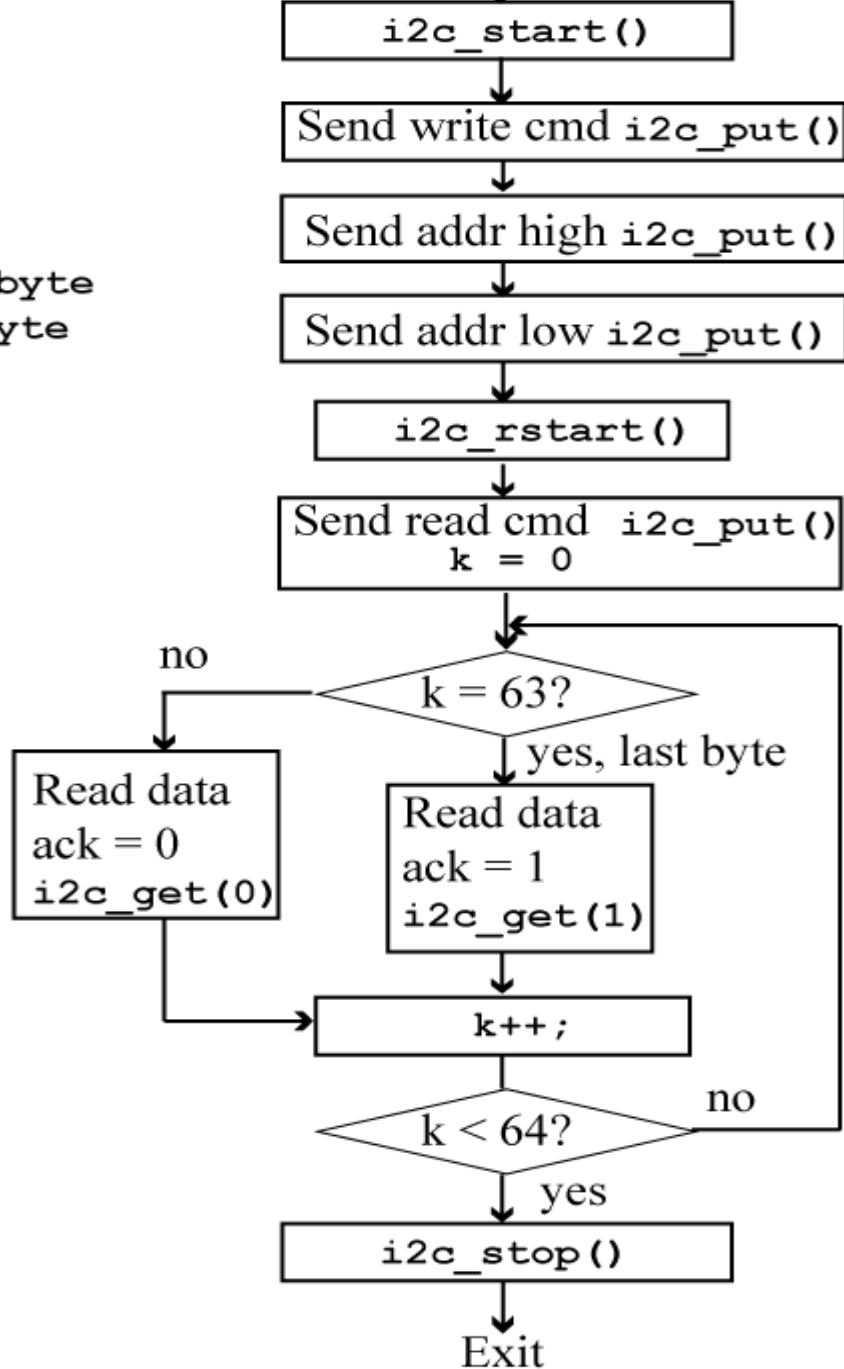
# i2c\_memread()



poll EEPROM to determine if last write is finished yet before doing read.

If returned ackbit = '0', then last write is finished.

byte  
byte



i2c\_memread() continued

Set address counter  
with write command

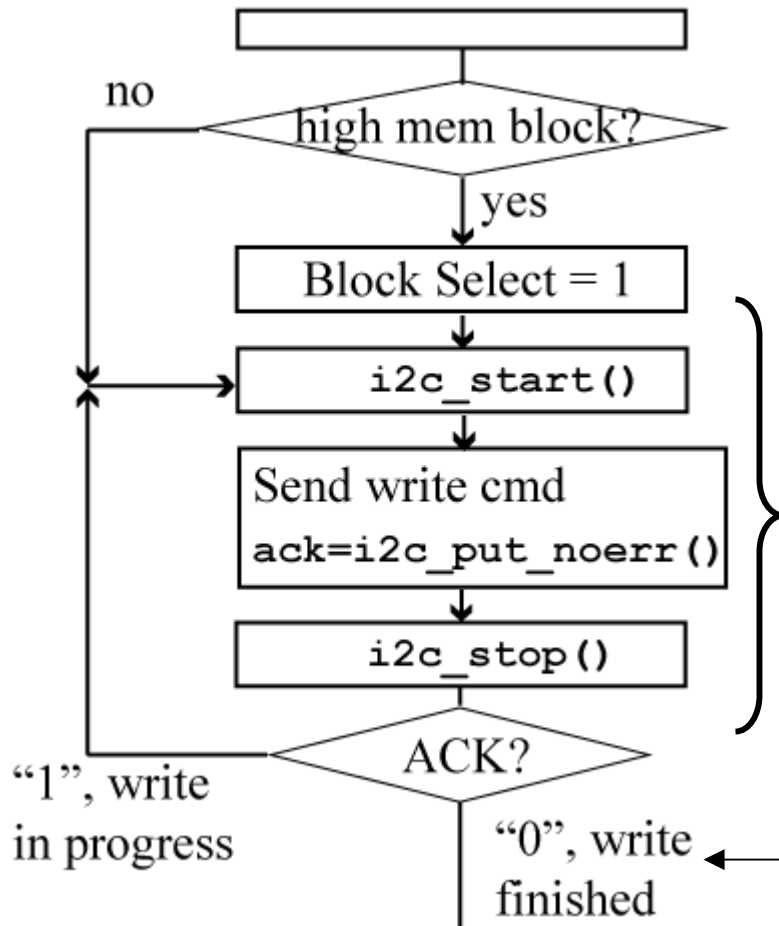
Send read command

read 64 bytes.

Return

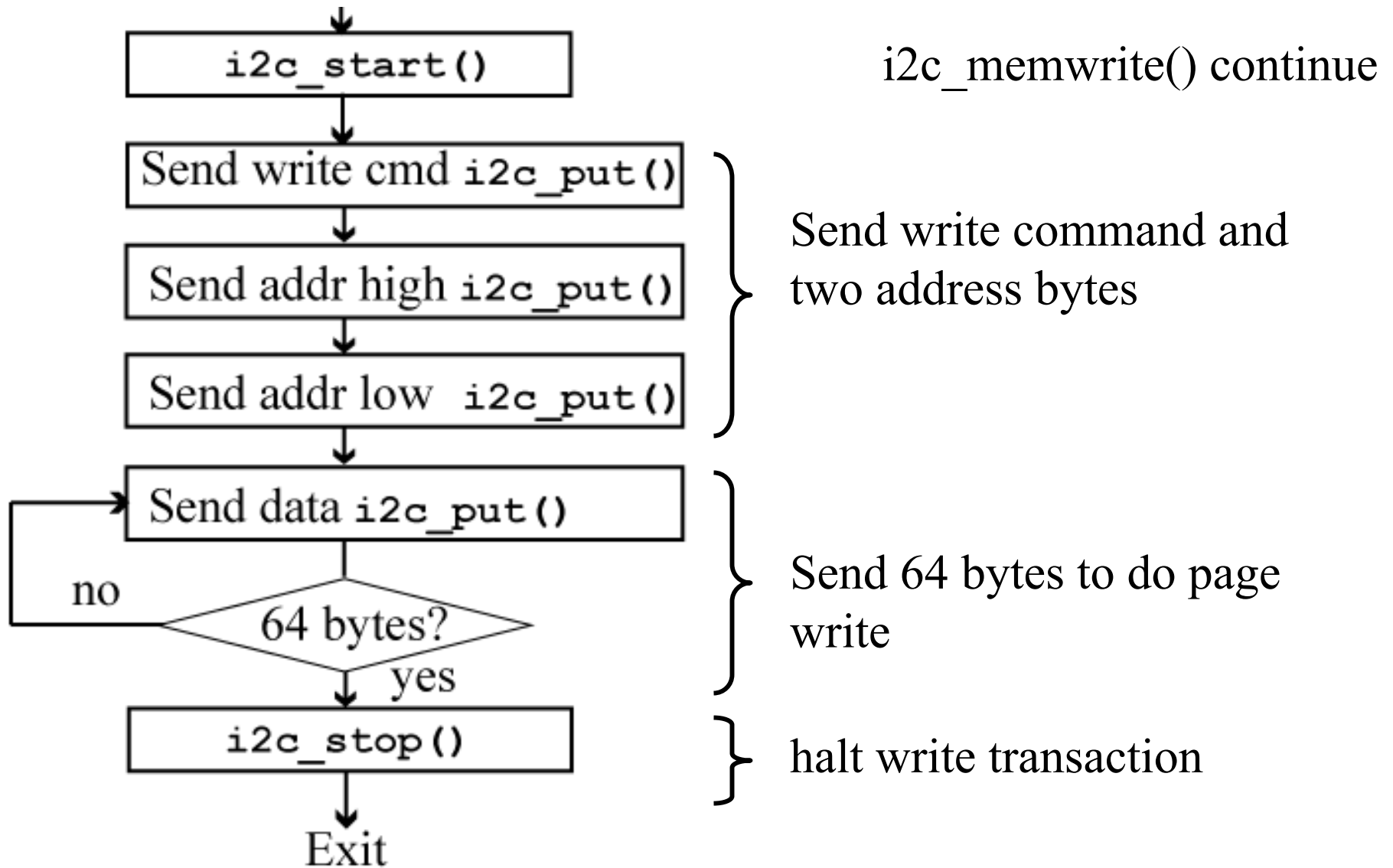
© Thomson/Delmar Learning 2005

# i2c\_memwrite()



poll EEPROM to determine if last write is finished yet before doing read.

If returned ackbit = '0', then last write is finished.



```
char membuf[BLKSIZE]; ← Storage for test strings
unsigned int memaddr;
```

```
main(void) {
    unsigned char mode,i;
    // 19200 in HSPLL mode, crystal = 7.3728 MHz
    serial_init(95,1); ← Initialize Serial Port
```

```
    if (!RI) {
        RI = 1;
        printf("Software reset!");pcrlf();
        if (i2c_errstat) i2c_print_err();
    }
    if (!TO) {
        printf("Watchdog timer reset has occurred.\n");
        pcrlf();
        if (i2c_errstat) i2c_print_err();
    }
```

} Software Reset, check if I<sup>2</sup>C function call was in progress

} Watchdog Timer Reset, check if I<sup>2</sup>C function call was in progress

```
i2c_init(72); ← Initialize I2C Port
```

```
pcrlf(); printf("I2C Mem Test Started"); pcrlf();
SWDTEN = 1; // enable watchdog timer
memaddr = 0;
printf ("Enter 'w' for write mode, anything else reads: ");
mode = getche(); pcrlf();
```

## EEPROM Test: i2cmemtst.c

## main loop of 'i2cmemtst'

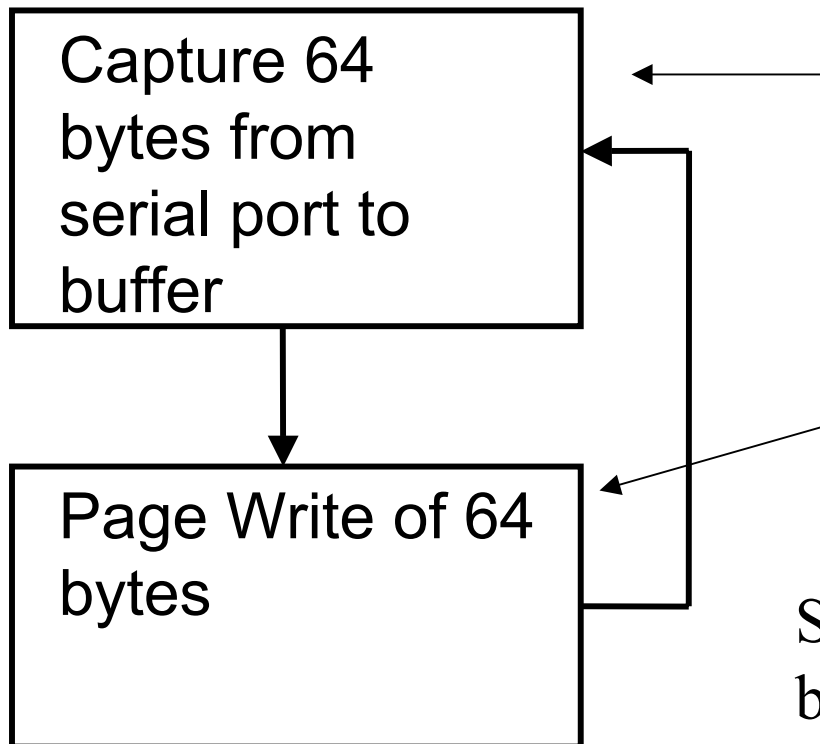
```
while(1) {
    if (mode == 'w') {
        printf("Enter %d chars.",BLKSIZE);pcrlf();
        for(i = 0;i< BLKSIZE;i++) {
            membuf[i] = getche();
        }
        pcrlf();printf("Doing Write");pcrlf();
        // write same string twice to
        //check Write Busy polling
        i2c_memwrite(EEPROM,memaddr,membuf);
        memaddr = memaddr +BLKSIZE;
        i2c_memwrite(EEPROM,memaddr,membuf);
        memaddr = memaddr +BLKSIZE;
    } else {
        // read 64 characters
        i2c_memread(EEPROM,memaddr,membuf);
        for(i = 0;i< BLKSIZE;i++) putchar(membuf[i]);
        pcrlf();
        printf("Any key continues read...");pcrlf();
        getch();
        memaddr = memaddr + BLKSIZE;
    }
}
```

Input 64 character string from console, write to EEPROM twice to check functionality of write-in-progress status check.

Read 64 characters from EEPROM and print to console

# Lab 9: I<sup>2</sup>C & Serial EEPROM

Goal: Capture streaming data from serial port, store to serial EEPROM



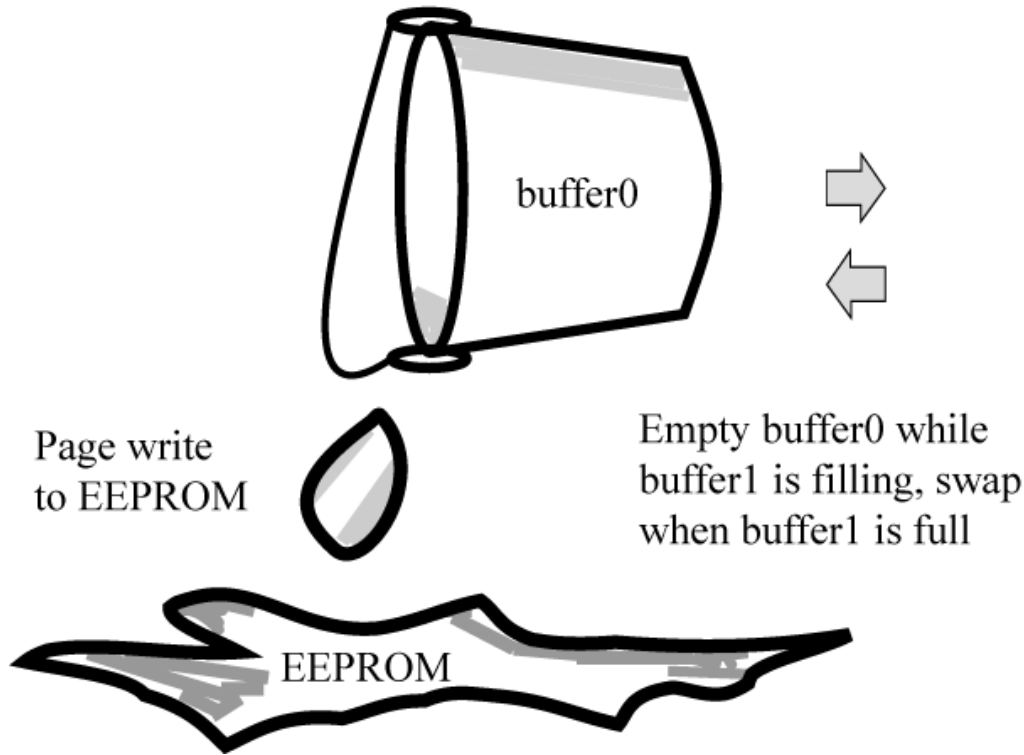
Interrupt service routine stores bytes in a buffer.

Problem: While writing bytes to serial EEPROM, more bytes are arriving!!!

Solution: Use two buffers! Second buffer captures data while first buffer data written to EEPROM.

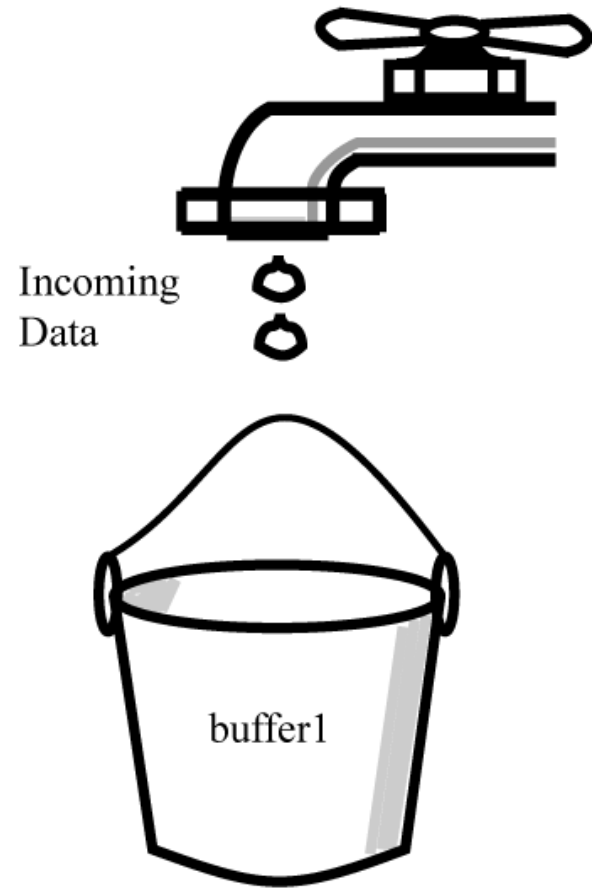
Interrupt Service Routine (background code) captures incoming data

Normal program flow (foreground code) does EEPROM writes.



When main() sees that buffer is full, tells ISR to swap buffers, then empties the full buffer.

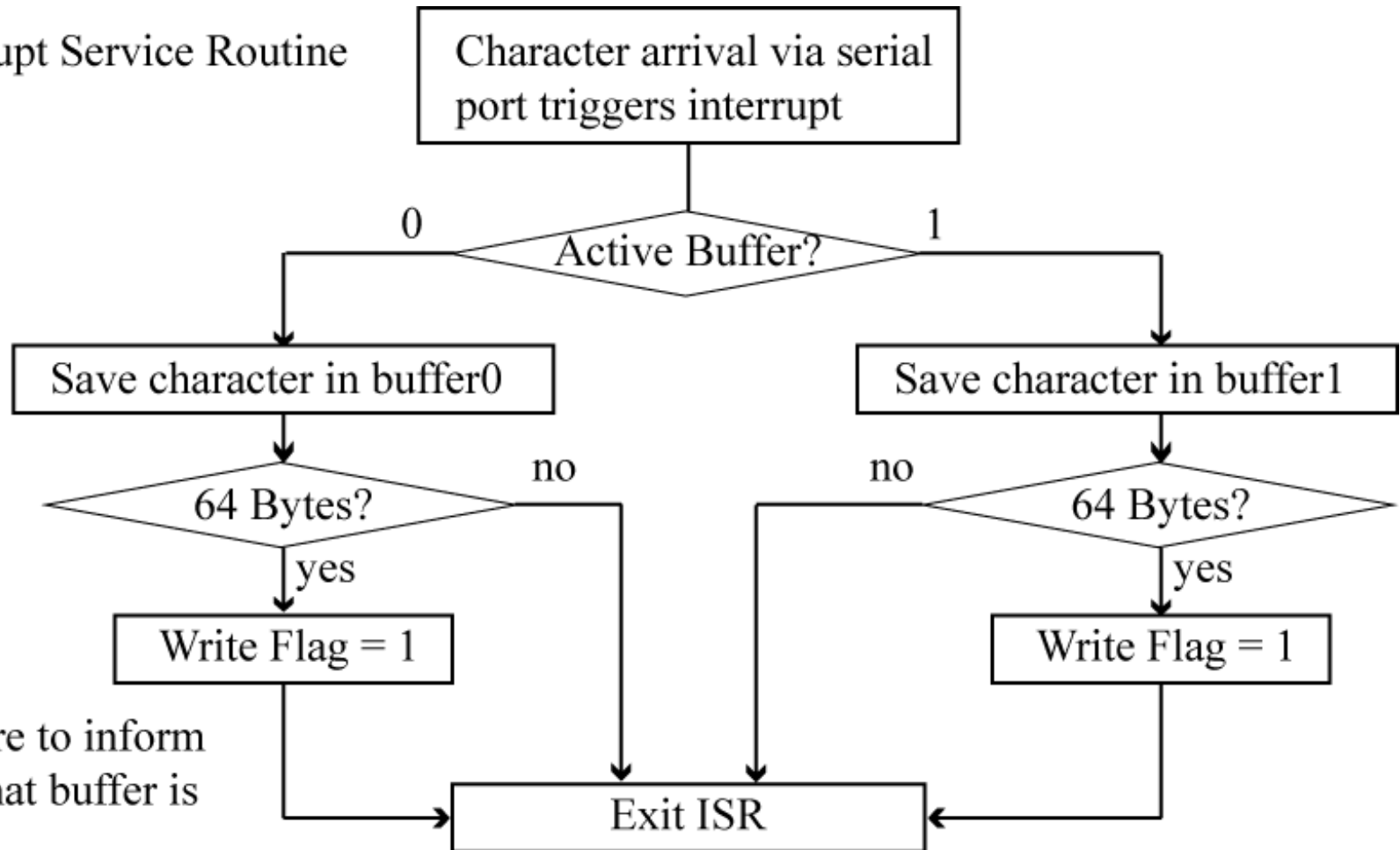
Empty buffer by writing contents to EEPROM, no longer need data after written to EEPROM.



Interrupt Service Routine (ISR) fills data buffer, sets flag when data buffer is full. ISR is invoked for each new input character.

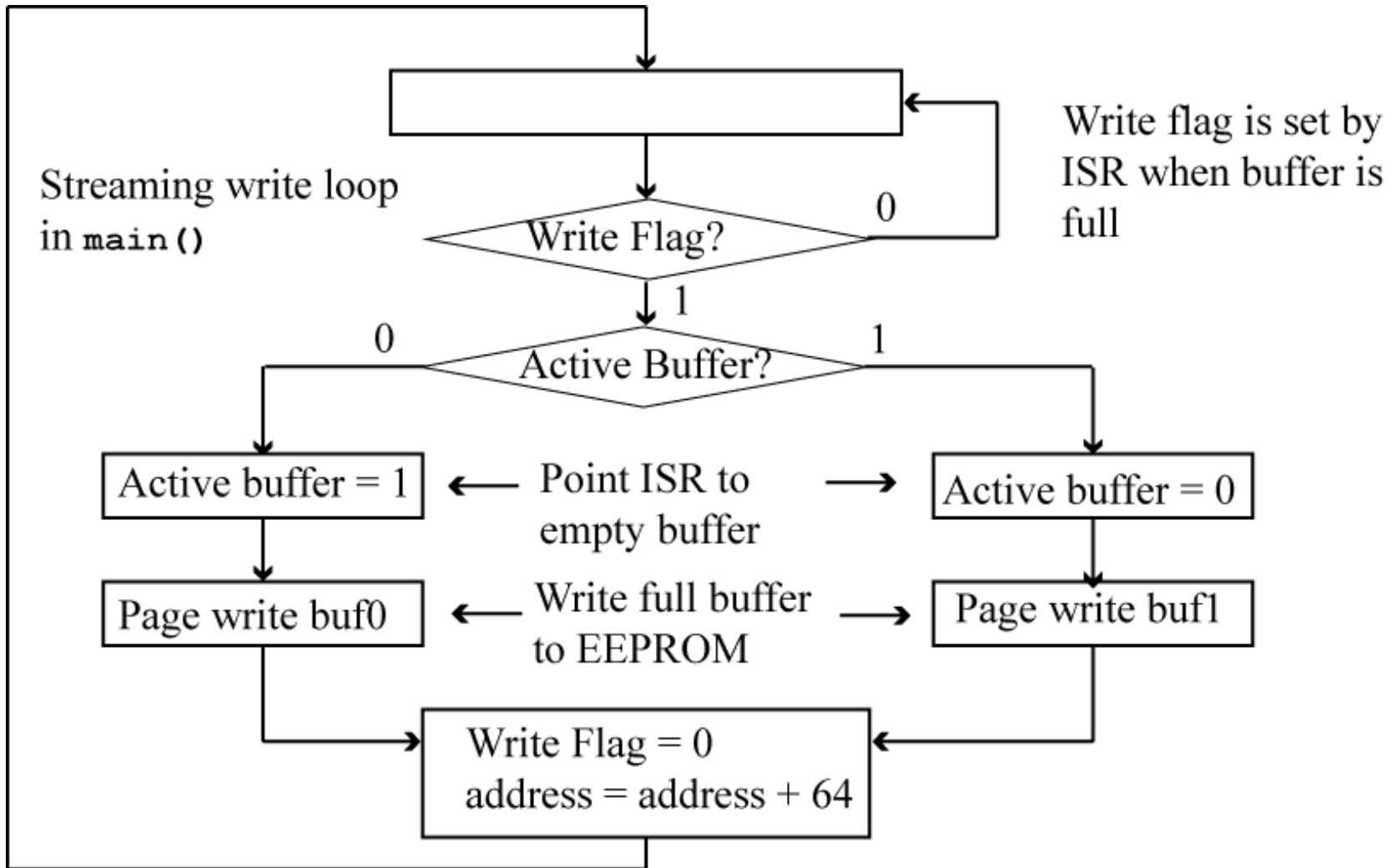
# ISR Flowchart

Interrupt Service Routine



Semaphore to inform `main()` that buffer is full.

# Streaming Write Loop: *main()*



# What do you have know?

- Serial Peripheral Interface (SPI) – how is this different from I<sup>2</sup>C?
- I<sup>2</sup>C Protocol
  - What is start, stop, ack conditions, purpose of each
  - How is data sent, received
  - Device addressing
- Serial EEPROM
  - Sequential, Random Read operations
  - Random, Block Write operations
- Using two buffers for performing streaming IO writes to EEPROM