SPI, I^2C Serial Interfaces

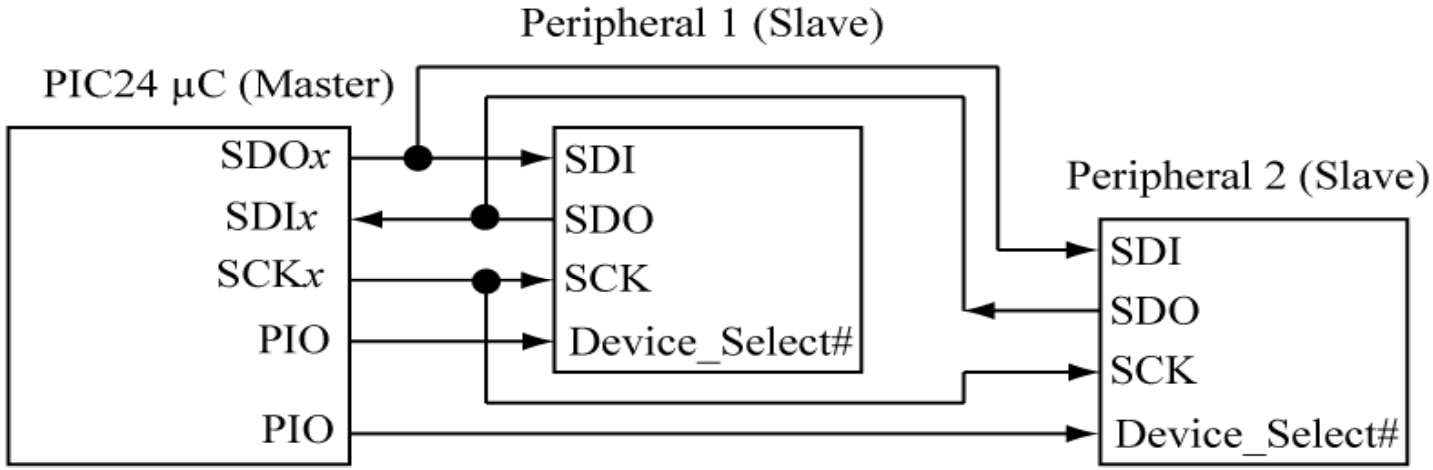
- The SPI and I^2C are two synchronous serial interfaces on the PIC24 μC

- Both are commonly used in industry

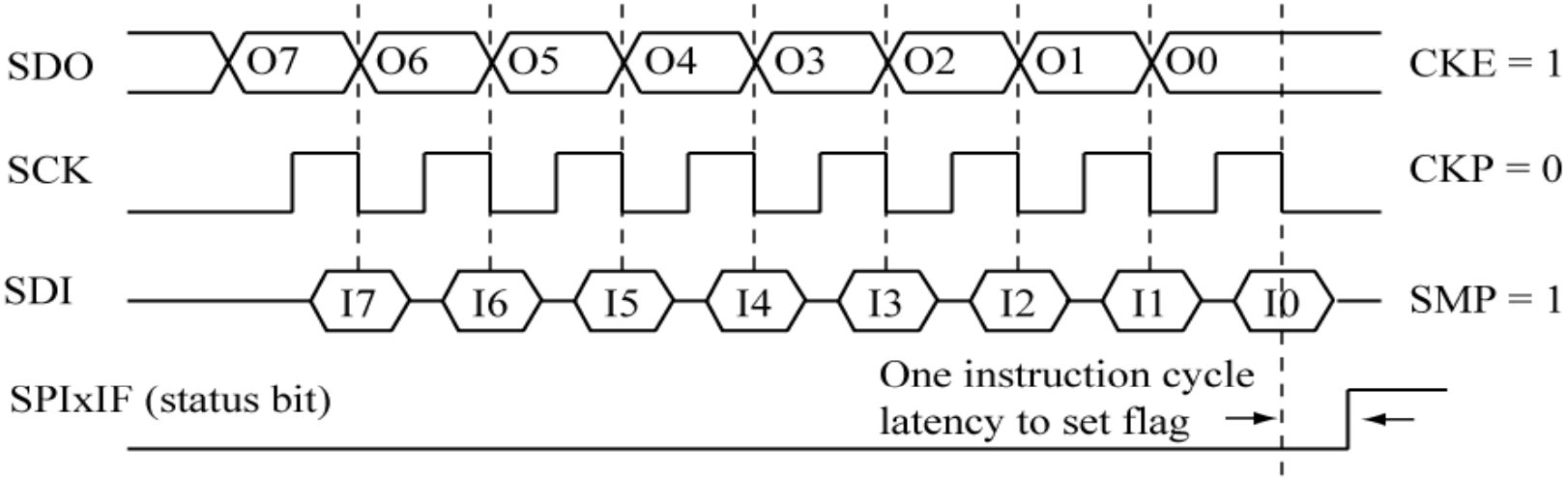
- The SPI port requires a minimum of three wires (and usually 4), and is technically duplex, even though most transfers are half-duplex. Its top speed on the PIC24 μC is 10 MHz.
    - Best for high-speed serial transfer
    - Very simple

- The I^2C port requires only two wires regardless of the number of peripherals, is half-duplex, and top speed is 1 MHz.
    - Best if you are trying to reduce external pin usage.

Serial Peripheral Interface (SPI)



Data is sent MSb first; received data is clocked in as transmitted data is clocked out. Every transmission is a duplex transmission because data is exchanged on SDO*x*/SDI*x*. *Device_Select#* must be low before transmission starts to select the Slave and must remain low for the duration of the transfer.



From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

SPIx Block Diagram for PIC24 µC

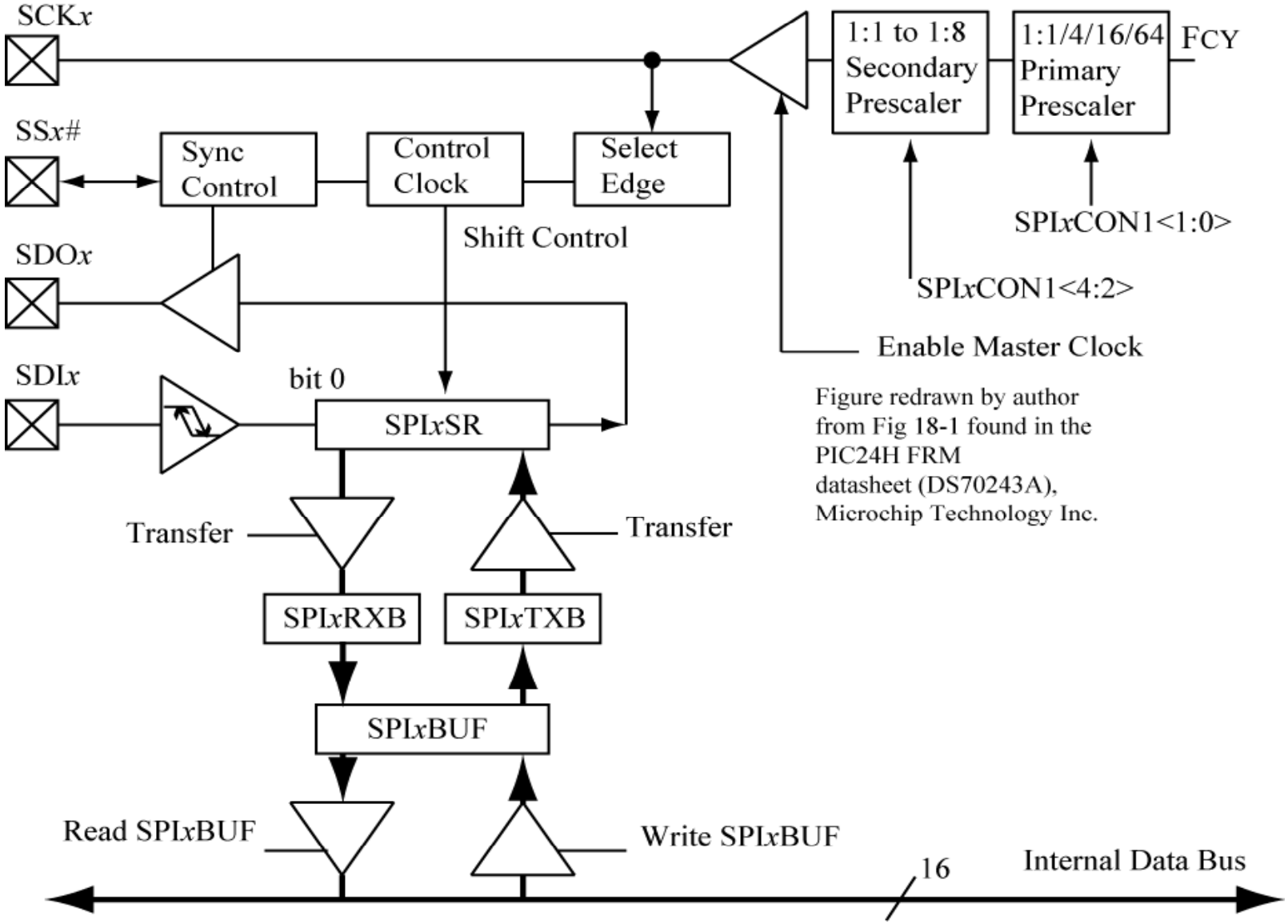
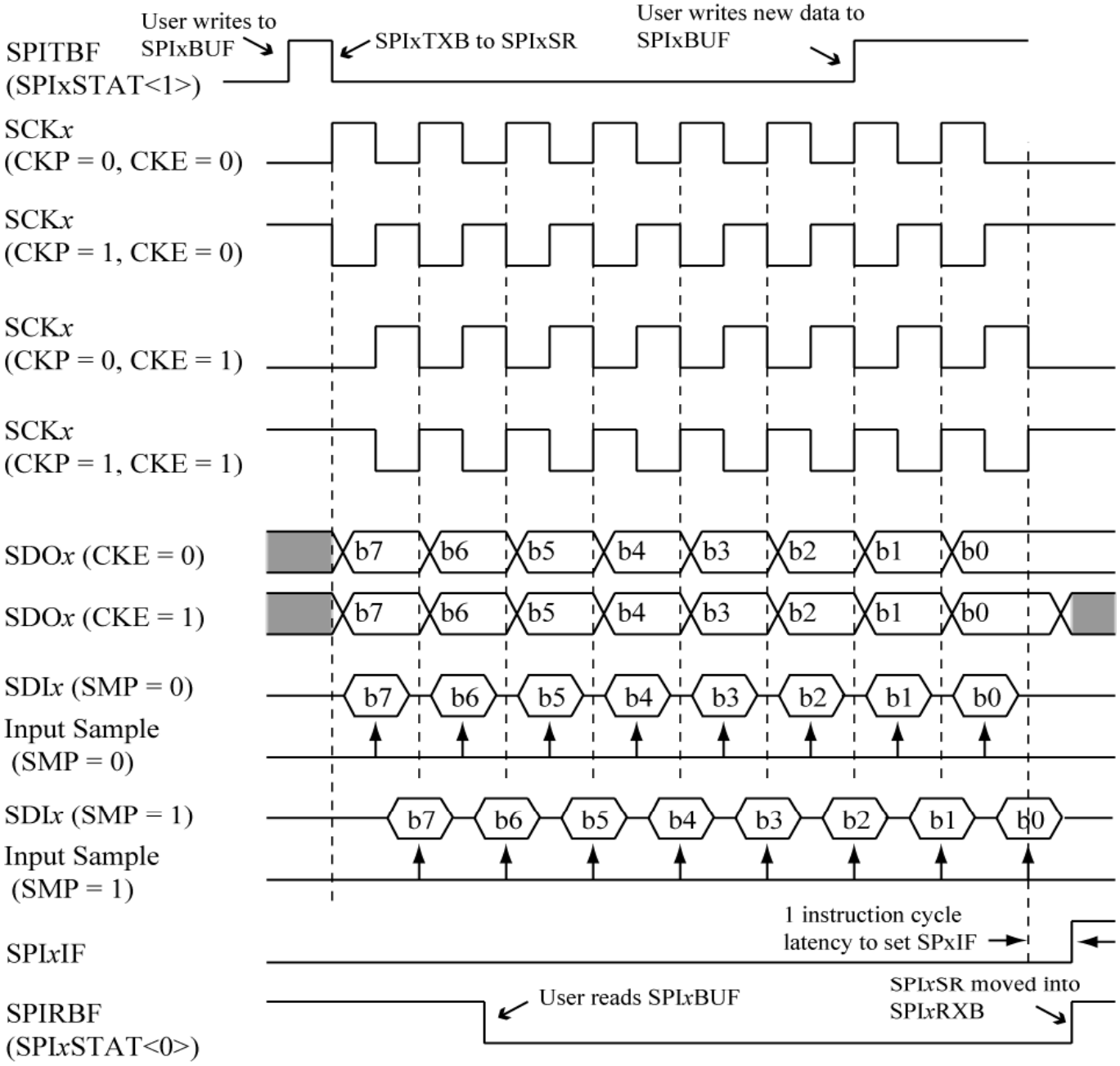


Figure redrawn by author from Fig 18-1 found in the PIC24H FRM datasheet (DS70243A), Microchip Technology Inc.

From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

SPI Transmission Formats



CKP – clock polarity.
CKE – controls which edge output data is transmitted on.

Which format is used depends on peripheral.

CKP =0, CKE = 1 seems common.

SPI *C* Functions

In *pic24_spi.c*, *pic24_spi.h*

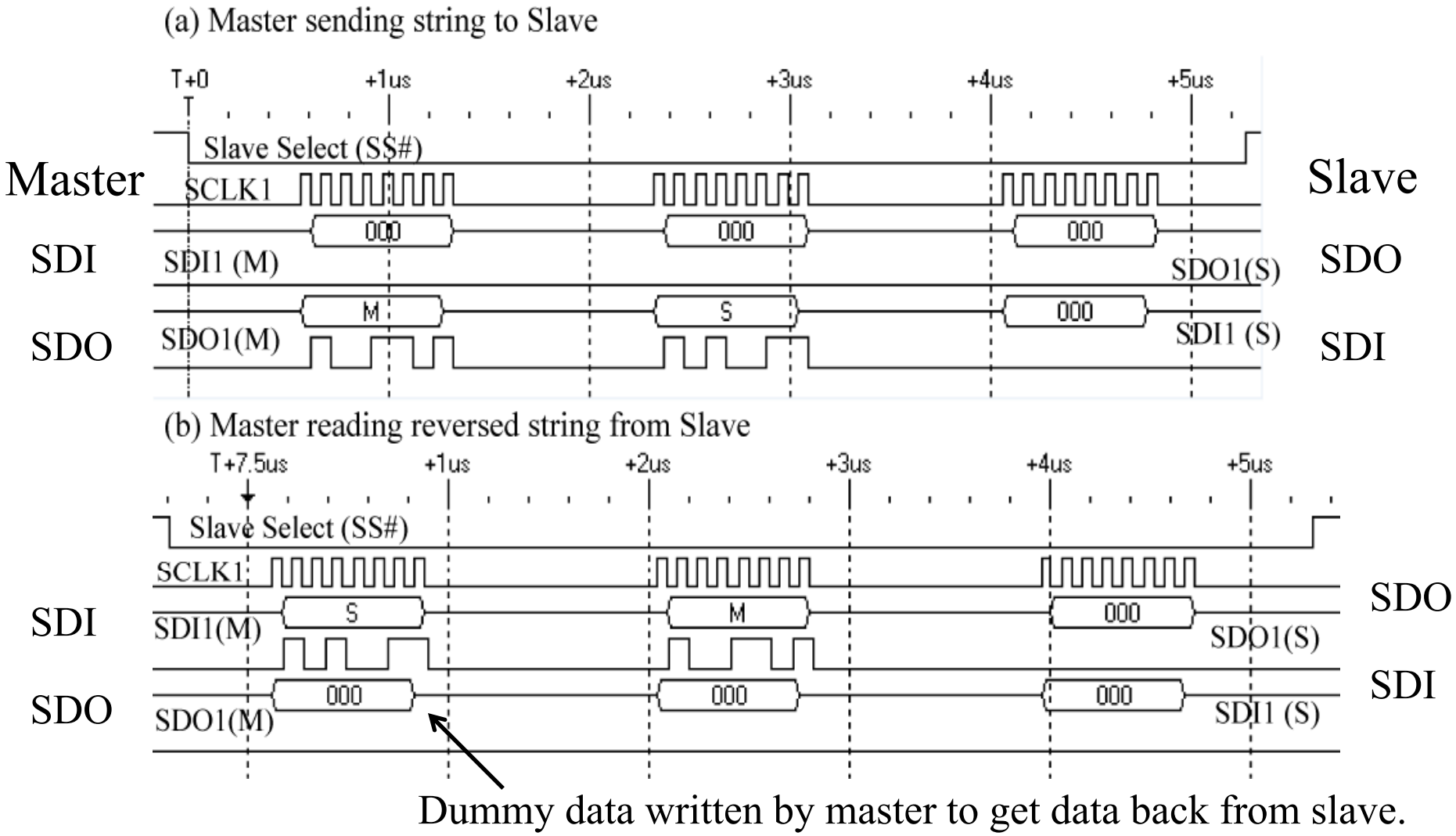
```
void checkRxErrorSPI1() {
 if (SPI1STATbits.SPIROV) {
   //clear the error
   SPI1STATbits.SPIROV = 0;
   reportError("SPI1 Receive Overflow\n");
 }
}
```

Only function needed besides configuration

```
uint16 ioMasterSPI1(uint16 u16_c) {
  checkRxErrorSPI1();
  _SPI1IF = 0;  //clear interrupt flag since we are about to
                // write new value
  SPI1BUF = u16_c;
  while (!_SPI1IF) { //wait for operation to complete
    doHeartbeat();
  }
  return SPI1BUF;  //return the shifted in value
}
```

Must ALWAYS read the input buffer or SPI overflow can occur!

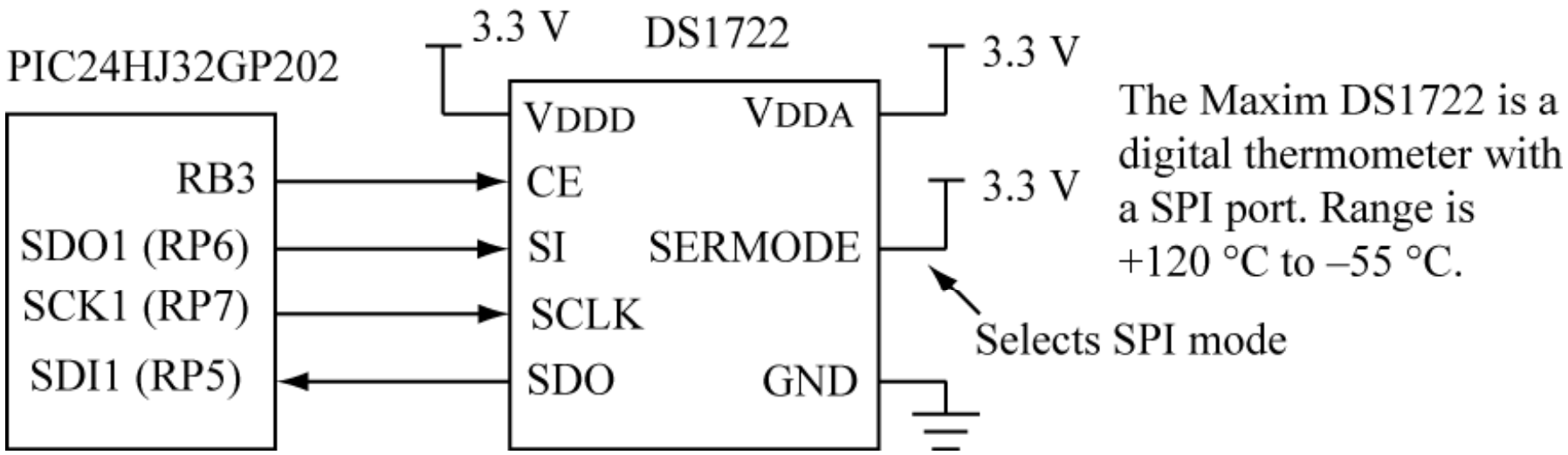
A SPI Transfer



To read data from a slave device, the master has to write data in order to get data back!

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

PIC24 μC Master to DS1722 Thermometer



We use RB3 from the PIC24 μC as the chip select for the DS1722. This chip select is high true.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

DS1722 Details

(a) Configuration byte

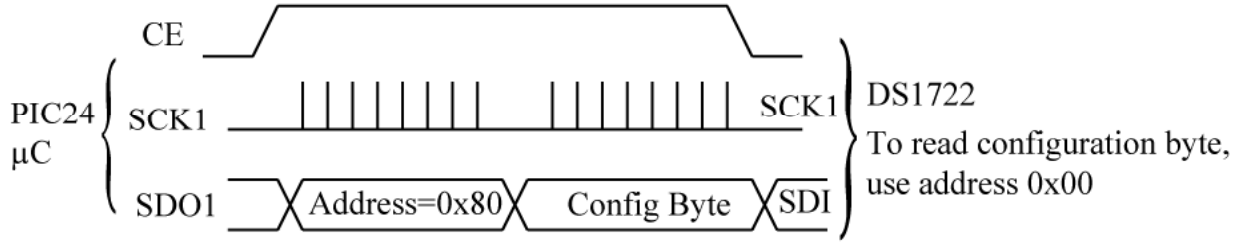
MSb								LSb
	1	1	1	1SHOT	R2	R1	R0	SD

SD: 0- continuous conversion, 1- complete current conversion, enter low power mode.

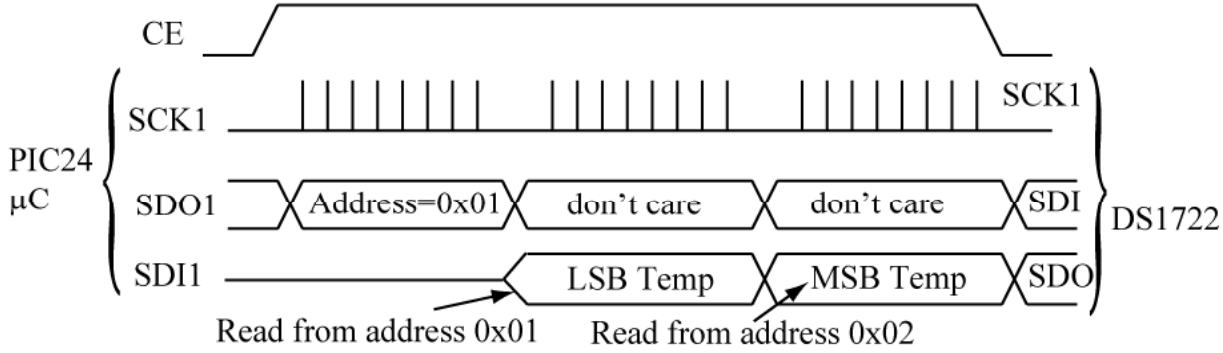
R2/R1/R0: 000 8-bit mode, 0.075s conversion time, 1.0° C resolution (8.0 signed fixed-point)
001 9-bit mode, 0.15s conversion time, 0.5° C resolution (8.1 signed fixed point)
010 10-bit mode, 0.3s conversion time, 0.25° C resolution (8.2 signed fixed point)
011 11-bit mode, 0.6s conversion time, 0.125° C resolution (8.3 signed fixed point)
1xx 12-bit mode, 1.2s conversion time, 0.0625° C resolution (8.4 signed fixed point)

1SHOT: when SD=1 writing a 1 to this bit starts conversion, is cleared when conversion finished.

(b) Single-byte transfer, write configuration



(c) Multi-byte transfer, read temperature



(d) Temperature data format is 8.4 two's complement fixed point (integer portion is MSByte, fractional is LSByte).

Celsius (float) = 16-bit temperature (int16) / 256

Write configuration byte to get the DS1722 started.
We will use continuous conversion mode.

Utility Functions for DS1722

```
#define CONFIG_SLAVE_ENABLE() CONFIG_RB3_AS_DIG_OUTPUT()
#define SLAVE_ENABLE()      _LATB3 = 1  //high true assertion
#define SLAVE_DISABLE()     _LATB3 = 0

void configSPI1(void) {
  //spi clock = 40MHz/1*4 = 40MHz/4 = 10MHz
  SPI1CON1 = SEC_PRESCAL_1_1 |      //1:1 secondary prescale
             PRI_PRESCAL_4_1 |      //4:1 primary prescale
             CLK_POL_ACTIVE_HIGH  | //clock active high (CKP = 0)
             SPI_CKE_OFF          | //out changes inactive to active (CKE=0)
             SPI_MODE8_ON         | //8-bit mode
             MASTER_ENABLE_ON;      //master mode
  //configure pins. Need SDO, SCK, SDI
  CONFIG_SDO1_TO_RP(6);       //use RP6 for SDO
  CONFIG_SCK1OUT_TO_RP(7);    //use RP7 for SCLK
  CONFIG_SDI1_TO_RP(5);       //use RP5 for SDI
  CONFIG_SLAVE_ENABLE();      //chip select for DS1722
  SLAVE_DISABLE();            //disable the chip select
  SPI1STATbits.SPIEN = 1;  //enable SPI mode
}
```

RB3 used for the DS1722 chip select.

10 MHz SPI clock

Clock can either be high or low true, but must use CKE=0.

RP6 used for SDO1, RP7 for SCK1, and RP5 for SDI.

Macros for SPI configuration are defined in *pic24_spi.h*

Utility Functions for DS1722 (cont.)

```
void writeConfigDS1722(uint8 u8_i) {
  SLAVE_ENABLE();          //assert chipselect
  ioMasterSPI1(0x80);      //config address
  ioMasterSPI1(u8_i);      //config value
  SLAVE_DISABLE();
}

int16 readTempDS1722() {
  uint16 u16_lo, u16_hi;
  SLAVE_ENABLE();          //assert chipselect
  ioMasterSPI1(0x01);      //LSB address
  u16_lo = ioMasterSPI1(0x00); //read LSByte
  u16_hi = ioMasterSPI1(0x00); //read MSByte
  SLAVE_DISABLE();
  return((u16_hi<<8) | u16_lo);
}
```

Writes to the DS1722 configuration register.

Reads 16-bit temperature value from DS1722.

Send dummy data to get data back.

Upper/lower bytes of temperature returned as single 16-bit value.

Testing the DS1722

(a) `main()` function.

```
int main (void) {
  int16 i16_temp;
  float  f_tempC,f_tempF;
  configBasic(HELLO_MSG);
  configSPI1();
  writeConfigDS1722(0xE8); //12-bit mode
  while (1) {
    DELAY_MS(1500);
    i16_temp = readTempDS1722();
    f_tempC = i16_temp;  //convert to floating point
    f_tempC = f_tempC/256;  //divide by precision
    f_tempF = f_tempC*9/5 + 32;
    printf("Temp is: 0x%0X, %4.4f (C), %4.4f (F)\n", i16_temp,
            (double) f_tempC, (double) f_tempF);
  }
}
```

Configure DS1722 for continuous conversion, 12-bit mode.

Use floating point and `printf` for convenience to print temperature value in Celsius and Fahrenheit.

(b) Sample Output

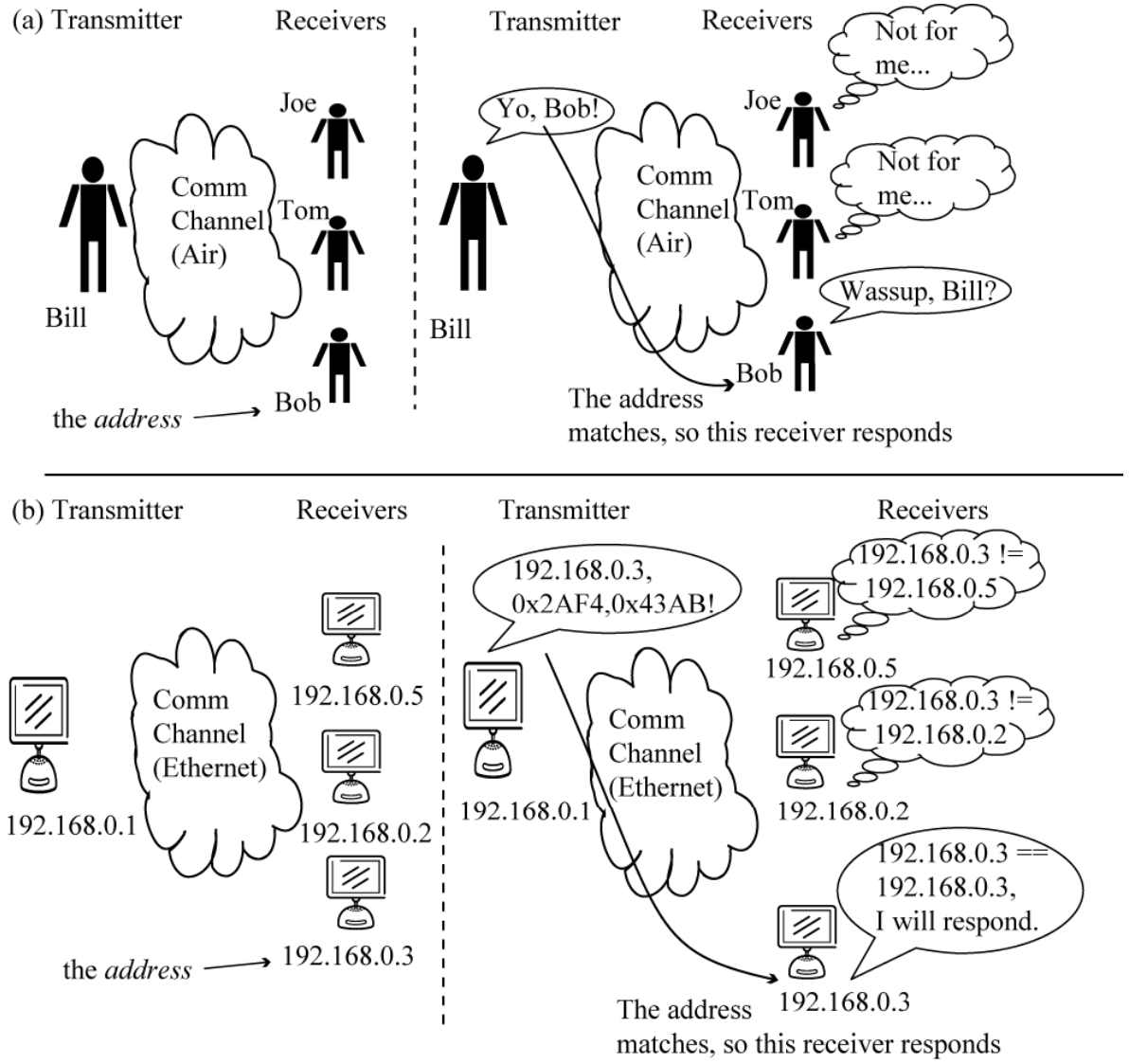
```
ds1722_spi_tempsense.c, built on Jun 27 2008 at 21:56:03
Temp is: 0x1BC0, 27.7500 (C), 81.9500 (F)
Temp is: 0x1BD0, 27.8125 (C), 82.0625 (F)
Temp is: 0x1BD0, 27.8125 (C), 82.0625 (F)
Temp is: 0x1C30, 28.1875 (C), 82.7375 (F)
Temp is: 0x1D70, 29.4375 (C), 84.9875 (F)
Temp is: 0x1DC0, 29.7500 (C), 85.5500 (F)
Temp is: 0x1E10, 30.0625 (C), 86.1125 (F)
Temp is: 0x1E30, 30.1875 (C), 86.3375 (F)
Temp is: 0x1D90, 29.5625 (C), 85.2125 (F)
Temp is: 0x1D30, 29.1875 (C), 84.5375 (F)
Temp is: 0x1CF0, 28.9375 (C), 84.0875 (F)
```

Finger placed on sensor to raise temperature.

Finger removed from sensor.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Bus Definition

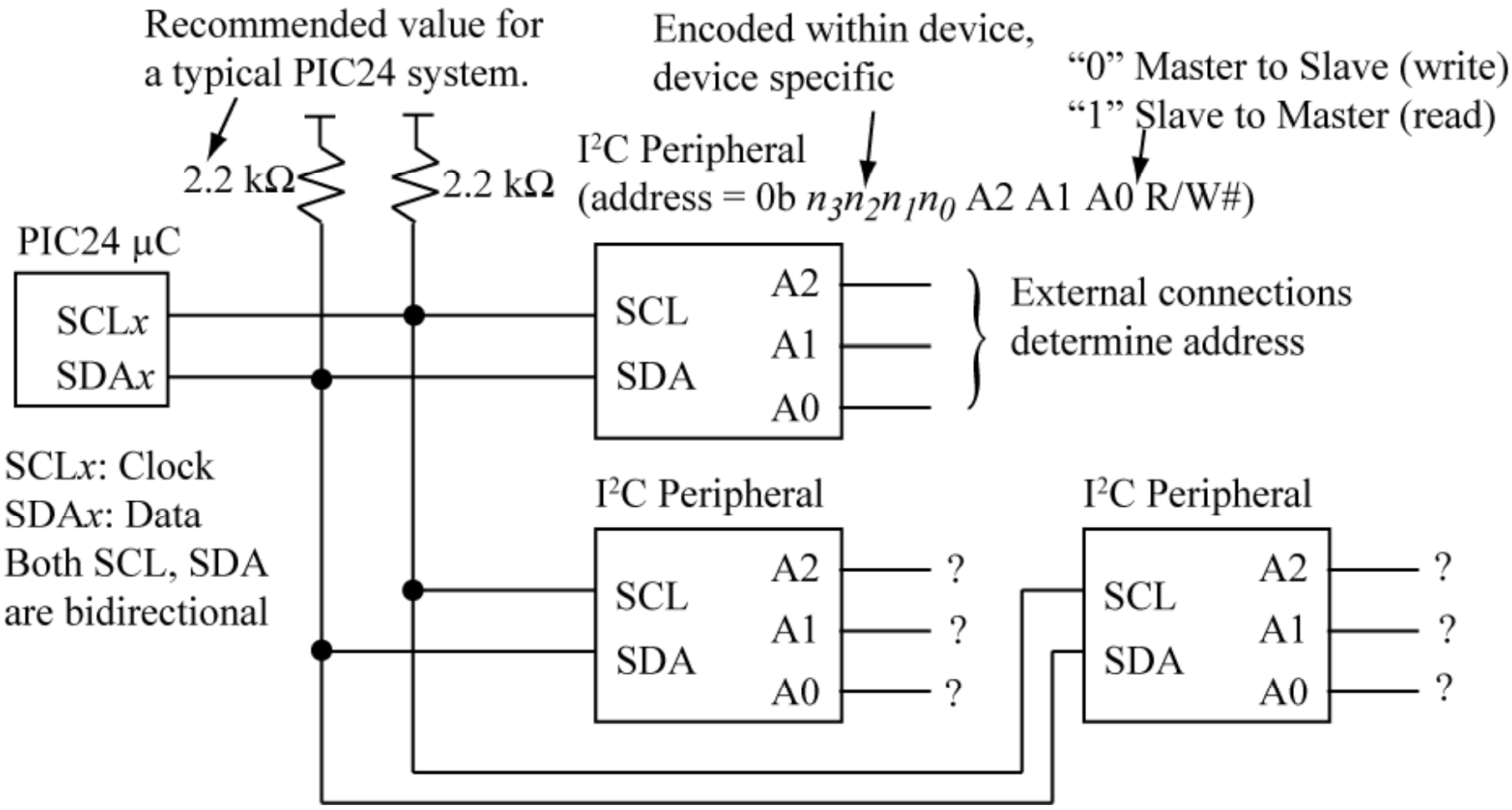


When a device on a bus talks, all hear what is said.

An *address* is used to specify what device the communication is intended for.

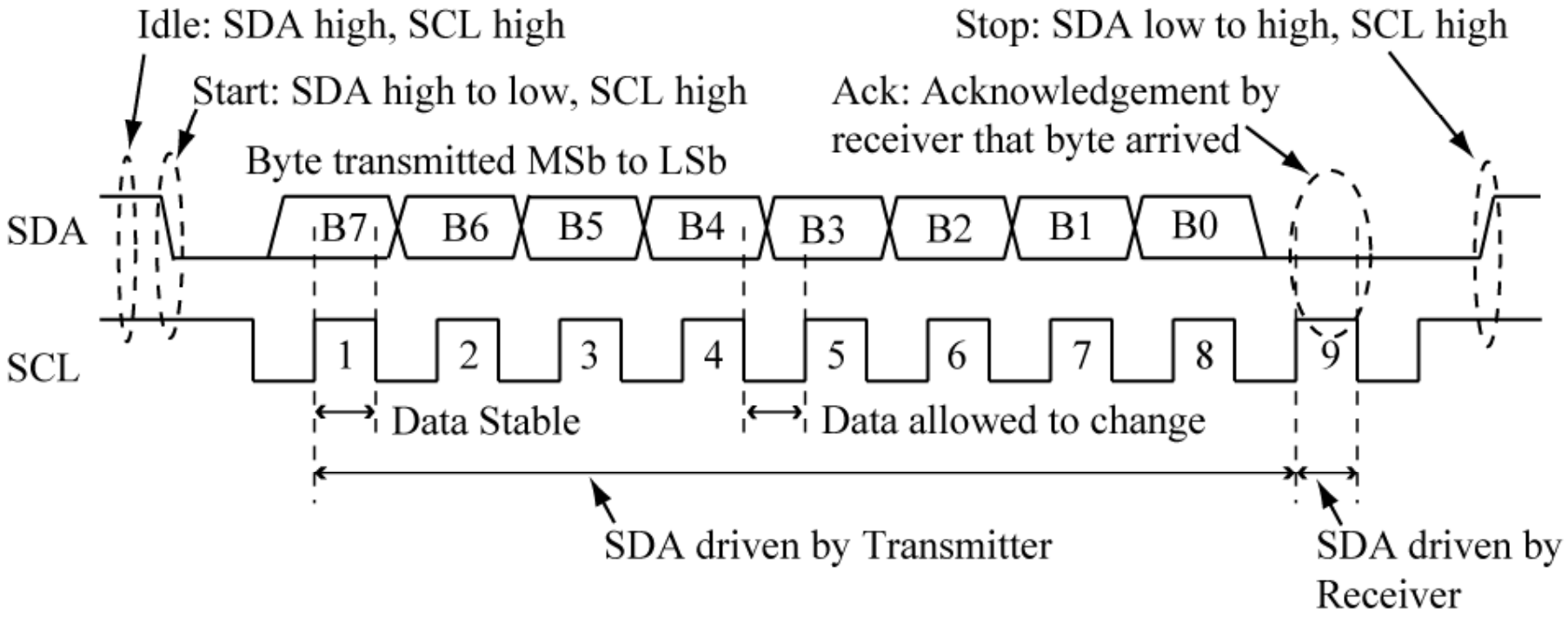
From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

Inter-Integrated Circuit (I^2C) Bus



From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

I²C Bus Signaling



Every byte transferred takes 9 bits because of acknowledgement bit.

I²C on the PIC24 µC

(a) I²C Module Registers

I²C Registers	Description
I2C*x*CON	Control Register
I2C*x*STAT	Status Register
I2C*x*BRG	Baud Rate Register
I2C*x*TRN	Transmit Register
I2C*x*RCV	Receive Register
I2C*x*MSK	Slave Mode Address Mask Register
I2C*x*ADD	Slave Mode Address Register

(b) Commonly used I²C Control and Status Bits

Bit Name	Register	Function
SEN	I2C*x*CON<0>	Set to begin Start sequence, cleared by HW
RSEN	I2C*x*CON<1>	Set to begin Repeated Start sequence, cleared by HW
PEN	I2C*x*CON<2>	Set to begin Stop condition, cleared by HW
RCEN	I2C*x*CON<3>	Set to enable receive, cleared in HW
ACKEN	I2C*x*CON<4>	Set to enable acknowledge sequence, cleared by HW
ACKDT	I2C*x*CON<5>	ACK bit to send; 1 for NAK, 0 for ACK.
I2CEN	I2C*x*CON<15>	Enable the I2C*x* module
RBF	I2C*x*STAT<1>	Set when I2C*x*RCV register is full, cleared by HW after read of I2C*x*RCV
SI2C*x*IF	Interrupt Flag Status Registers	Interrupt flag set on detection of address reception in Slave mode, reception of data, or request to transmit data

Support Functions – I²C Operations

(a) Support Functions for I²C Operations

I²C Support Functions (Operations)	**Description**
`void configI2C1(uint16 u16_FkHZ)`	Enables the I²C module for operation at `u16_FkHZ` kHz clock rate
`void startI2C1(void)`	Performs start operation
`void rstartI2C1(void)`	Performs repeated start operation
`void stopI2C1(void)`	Performs stop operation
`void putI2C1(uint8 u8_val)`	Transmits `u8_val`; software reset if NAK returned.
`uint8 putNoAckCheckI2C1(uint8 u8_val)`	Transmits `u8_val` and returns received acknowledge bit
`uint8 getI2C1(uint8 u8_ack2Send)`	Receive one byte and send `u8_ack2Send` as acknowledge bit

These are primitive operations.

Support Functions – I²C Transactions

(b) Support Functions for I²C Transactions

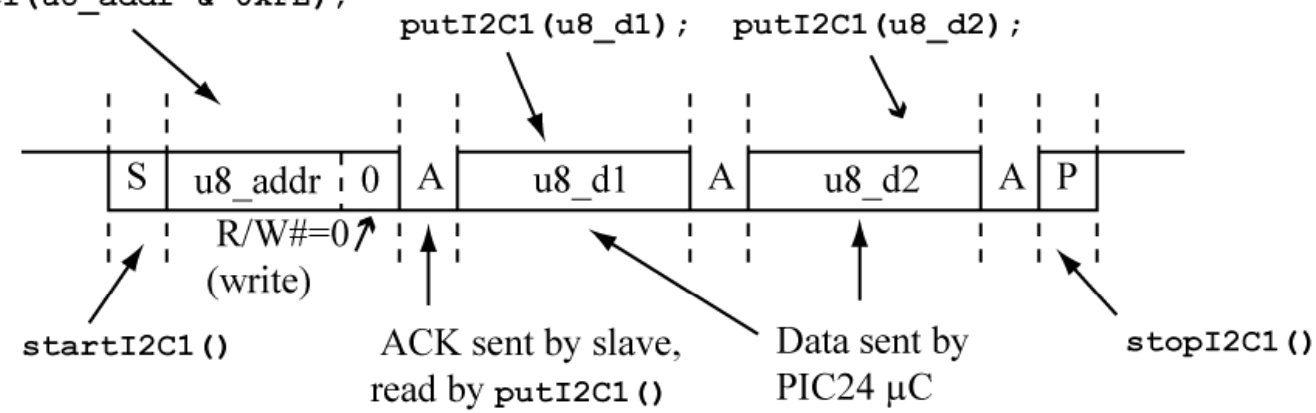
I²C Support Functions (Transactions)	Description
`void write1I2C1(uint8 u8_addr,uint8 u8_d1)`	Write 1 byte (`u8_d1`)
`void write2I2C1(uint8 u8_addr,uint8 u8_d1, uint8 u8_d2)`	Write 2 bytes (`u8_d1`)
`void writeNI2C1(uint8 u8_addr,uint8* pu8_data, uint16 u16_cnt)`	Write `u16_cnt` bytes in buffer `pu8_data`
`void read1I2C1 (uint8 u8_addr,uint8* pu8_d1)`	Read 1 byte; return in `*pu8_d1`
`void read2I2C1 (uint8 u8_addr,uint8* pu8_d1, uint8* pu8_d2)`	Read 2 bytes; return in `*pu8_d1`, `*pu8_d2`
`void readNI2C1 (uint8 u8_addr,uint8* pu8_data, uint16 u16_cnt)`	Read `u16_cnt` bytes; return in `*pu8_data`

These are use the primitive operations to read/write 1 or more bytes to a slave.

I²C Read/Write Transactions

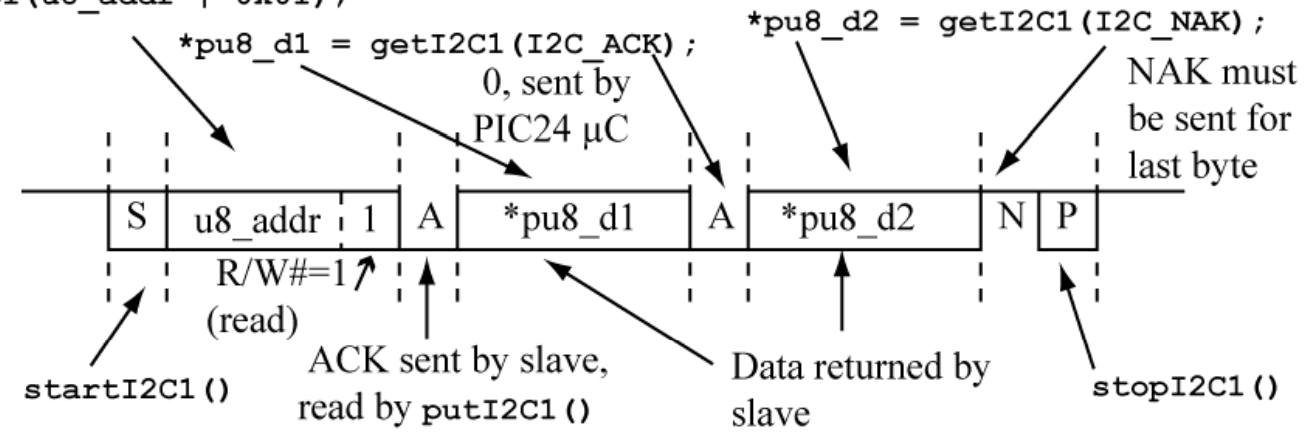
(a) Write two bytes to slave:

```
write2I2C1(uint8 u8_addr,uint8 u8_d1, uint8 u8_d2)
putI2C1(u8_addr & 0xFE);
```



(b) Read two bytes from slave:

```
read2I2C1(uint8 u8_addr, uint8* pu8_d1, uint8* pu8_d2)
putI2C1(u8_addr | 0x01);
```



Example primitive function

```
void configI2C1(uint16 u16_FkHZ) {
  uint32 u32_temp;

  u32_temp = (FCY/1000L)/((uint32) u16_FkHZ);
  u32_temp = u32_temp - FCY/10000000L - 1;
  I2C1BRG = u32_temp;
  I2C1CONbits.I2CEN = 1;
}

void startI2C1(void) {
  uint8 u8_wdtState;

  sz_lastTimeoutError = "I2C Start";
  u8_wdtState = _SWDTEN;  //save WDT state
  _SWDTEN = 1; //enable WDT
  I2C1CONbits.SEN = 1;    // initiate start
  // wait until start finished
  while (I2C1CONbits.SEN);
  _SWDTEN = u8_wdtState;  //restore WDT
  sz_lastTimeoutError = NULL;
}
```

Compute I2C1BRG value for operation at `u16_FkHZ` kHz.

Enable I^2C module

Functions `stopI2C1()`, `rstartI2C1()` are similar but use bits `PEN`, `RSEN` respectively.

Initiate start condition and wait for finish.

Transactions

(a) Write Transactions

```
#define I2C_WADDR(x)  (x & 0xFE)

void write1I2C1(uint8 u8_add.
uint8 u8_d1){
 startI2C1();
 putI2C1(I2C_WADDR(u8_addr));
 putI2C1(u8_d1);
 stopI2C1();
}

void write2I2C1(uint8 u8_addr,
uint8 u8_d1, uint8 u8_d2){
 startI2C1();
 putI2C1(I2C_WADDR(u8_addr));
 putI2C1(u8_d1);
 putI2C1(u8_d2);
 stopI2C1();
}
void writeNI2C1(uint8 u8_addr,
uint8* pu8_data,
uint16 u16_cnt){
 uint16 u16_i;
 startI2C1();
 putI2C1(I2C_WADDR(u8_addr));
 for (u16_i=0; u16_i < u16_cnt;){
  putI2C1(*pu8_data);
  pu8_data++;u16_i++;
 }
 stopI2C1();
}
```

LSb must be 0 for write.

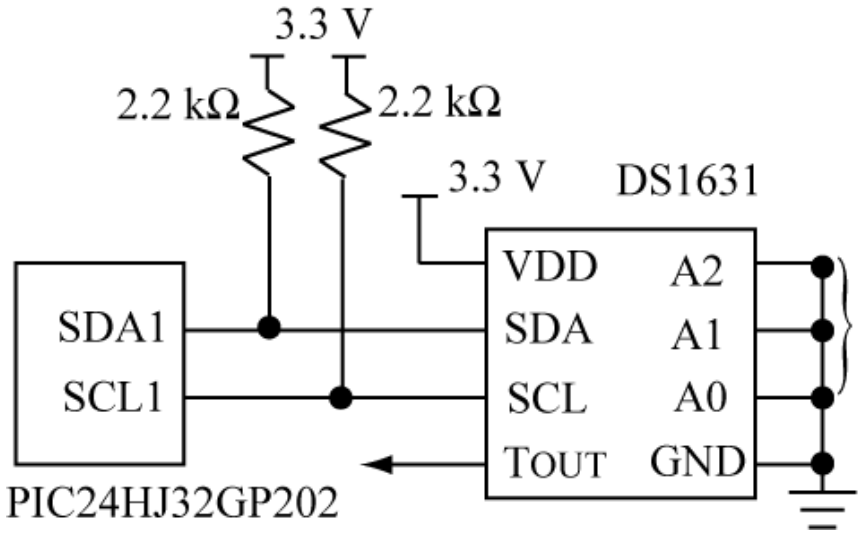
(b) Read Transactions

```
#define I2C_RADDR(x)  (x | 0x01)

void read1I2C1 (uint8 u8_addr,
  uint8* pu8_d1) {
 startI2C1();
 putI2C1(I2C_RADDR(u8_addr));
 *pu8_d1 = getI2C1(I2C_NAK);
 stopI2C1();
}

void read2I2C1 (uint8 u8_addr,
  uint8* pu8_d1, uint8* pu8_d2) {
 startI2C1();
 putI2C1(I2C_RADDR(u8_addr));
 *pu8_d1 = getI2C1(I2C_ACK);
 *pu8_d2 = getI2C1(I2C_NAK);
 stopI2C1();
}

void readNI2C1 (uint8 u8_addr,
  uint8* pu8_data, uint16 u16_cnt) {
 uint16 u16_i;
 startI2C1();
 putI2C1(I2C_RADDR(u8_addr));
 for (u16_i=0; u16_i < u16_cnt;){
  if (u16_i != u16_cnt-1)
        *pu8_data = getI2C1(I2C_ACK);
   else *pu8_data = getI2C1(I2C_NAK);
  pu8_data++; u16_i++;
 }
 stopI2C1();
}
```

LSb must be 1 for read.

Used for block data Transfers.

PIC24 μC Master to DS1631 Thermometer



Similar to the DS1722 but does not have as many precision options. Also has a thermostat function – Two internal registers named TH, TL used for that. When temp > TH, the TOUT output goes high. When Temp falls below TL, the TOUT output goes back low.

TH, TL are stored in non-volatile memory.

DS1631 Details

Address byte format for DS1631 Temperature Sensor

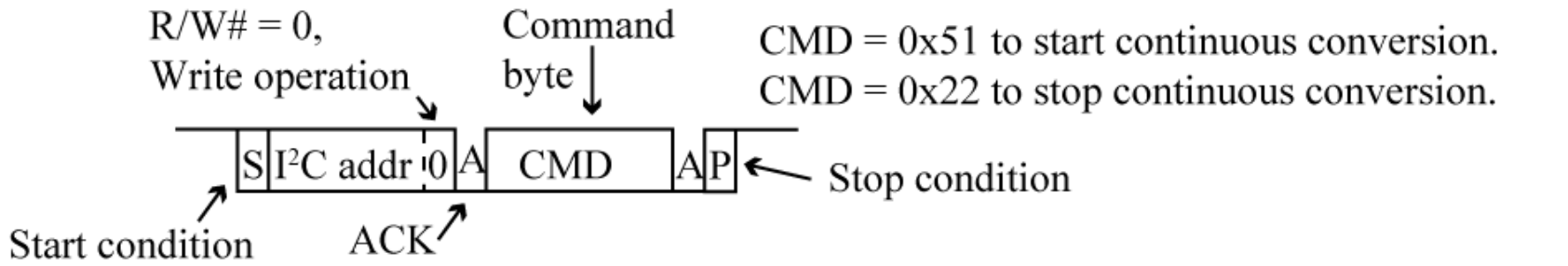
7	6	5	4	3	2	1	0
1	0	0	1	A2	A1	A0	R/W#

9 (8.1), 10 (8.2), 11 (8.3), 12-bit (8.4) temperature in Celsius, fixed-point two's complement format.

	MSByte	LSByte
	T[11:4] of temp. value	T[3:0]
–27.3125 ºC = 0xE4B0 (12-bit mode)	1 1 1 0 0 1 0 0	1 0 1 1 0 0 0 0

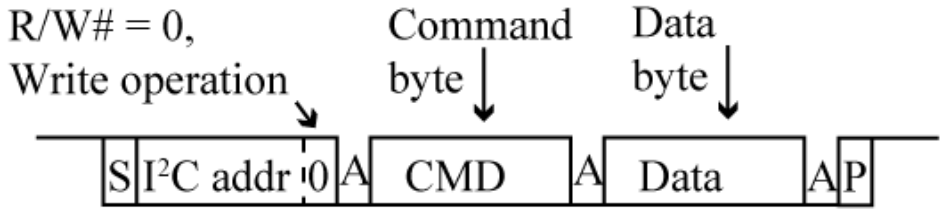
0xE4B0 = –6992 = –6992/256 = –27.3125 ºC

(a) Standalone command



(b) 8-bit Write

CMD = 0xAC accesses configuration register.

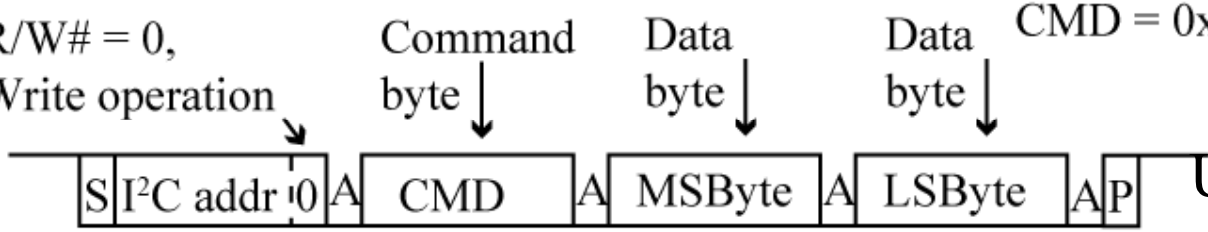


Used to configure the device

DS1631 Details

(c) 16-bit Write

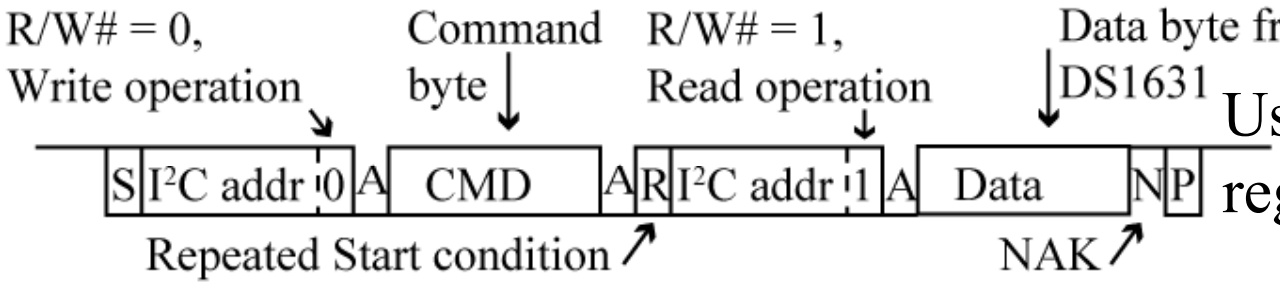
CMD = 0xA1 Temp. High Trigger.
CMD = 0xA2 Temp. Low Trigger.



Used to set TH, TL regs.

(d) 8-bit Read

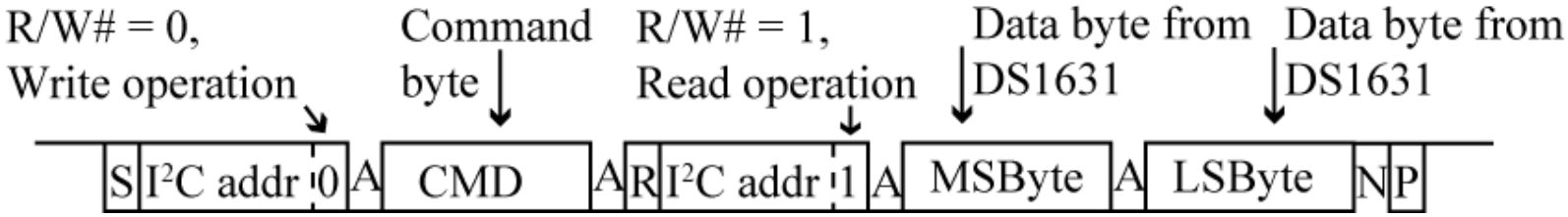
CMD = 0xAC accesses configuration register.



Used to read the config. register

(e) 16-bit Read

CMD = 0xAA accesses 16-bit temperature register.



Used to read the 9-bit temperature value.

DS1631 Configuration Register

	7	6	5	4	3	2	1	0
CONFIG Register	DONE	THF	TLF	NVB	1	0	POL	1SHOT

DONE	Conversion Done Flag: "1" when conversion is complete, "0" when conversion is in progress.
THF	Temperature High Flag: "1" if temperature has exceeded the TH register value since power-on; reset on power down, write to CONFIG register, or software power-on-reset command.
TLF	Temperature Low Flag: "1" if temperature has dropped below the TL register value since power-on; reset on power down or write to CONFIG register, or software power-on-reset command.
NVB	Nonvolatile Memory Busy flag: "1" when write to nonvolatile memory is in progress, "0" otherwise.
R1:R0	Resolution selection bits, 00: 9-bit (93.75 ms), 10-bit (187.5 ms), 11-bit (375 ms), 12-bit (750 ms)
POL	Polarity bit: "1" TOUT is active high, "0" TOUT is active low. Stored in NVM.
1SHOT	One-Shot Mode: "1" DS1631 only performs conversions upon receiving a Start Conversion command; "0" the DS1621 performs continuous conversions. Store in NVM.

After configuring for continuous conversion, must sent the *Start* command (0xEE) to start conversions.

Support Functions

```
#define DS1631ADDR 0x90    //DS1631 address with all pins tied low
#define ACCESS_CONFIG 0xAC
#define START_CONVERT 0x51
#define READ_TEMP 0xAA
```

DS1631 address = 0b 1001 A2 A1 A0 R/W
0x90 = 0b 1001 0 0 0 ?

```
void writeConfigDS1631(uint8 u8_i) {
  write2I2C1(DS1631ADDR, ACCESS_CONFIG, u8_i);
}
```

} Implements 8-bit write command.

```
void startConversionDS1631(){
  write1I2C1(DS1631ADDR, START_CONVERT);
}
```

} Implements standalone command.

```
int16 readTempDS1631() {
  uint8 u8_lo, u8_hi;
  int16 i16_temp;
  write1I2C1(DS1631ADDR, READ_TEMP);
  read2I2C1 (DS1631ADDR, &u8_hi, &u8_lo);
  i16_temp = u8_hi;
  return ((i16_temp<<8)|u8_lo);
}
```

} Implements 16-bit read command.

These use the ‘transaction’ functions to communicate with the DS1621

Testing the DS1631

```
int main (void) {
  int16 i16_temp;
  float  f_tempC,f_tempF;
  configBasic(HELLO_MSG);
  configI2C1(400);                //configure I2C for 400 kHz
  writeConfigDS1631(0x0C);        //continuous conversion, 12-bit mode
  startConversionDS1631();        //start conversions
  while (1) {
    DELAY_MS(750);
    i16_temp = readTempDS1631();
    f_tempC = i16_temp;  //convert to floating point
    f_tempC = f_tempC/256;  //divide by precision
    f_tempF = f_tempC*9/5 + 32;
    printf("Temp is: 0x%0X, %4.4f (C), %4.4f (F)\n",
           i16_temp, (double) f_tempC, (double) f_tempF);
  }
}
```

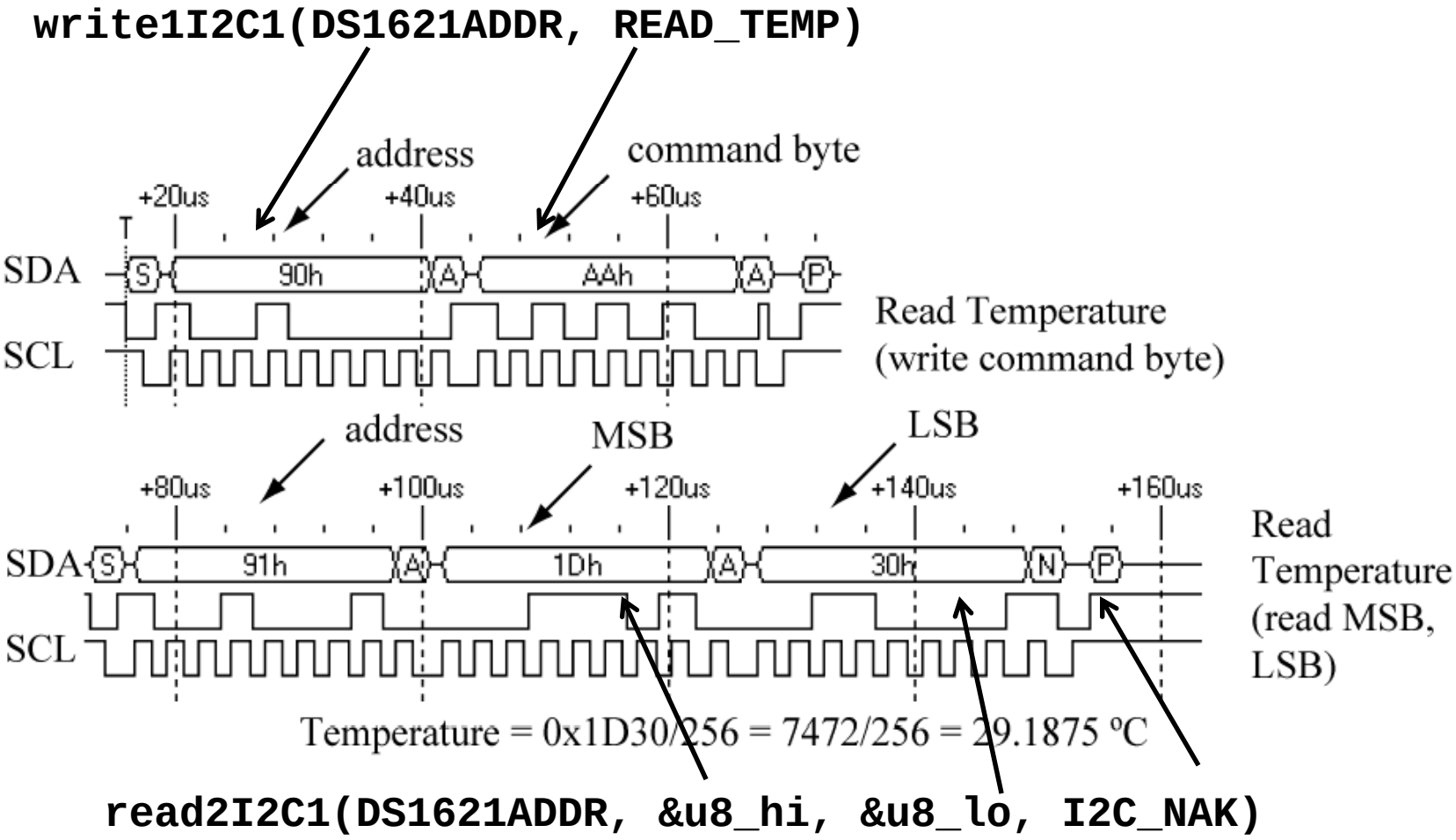
Configure I²C bus for 400 kHz.

Configure for continuous conversions.

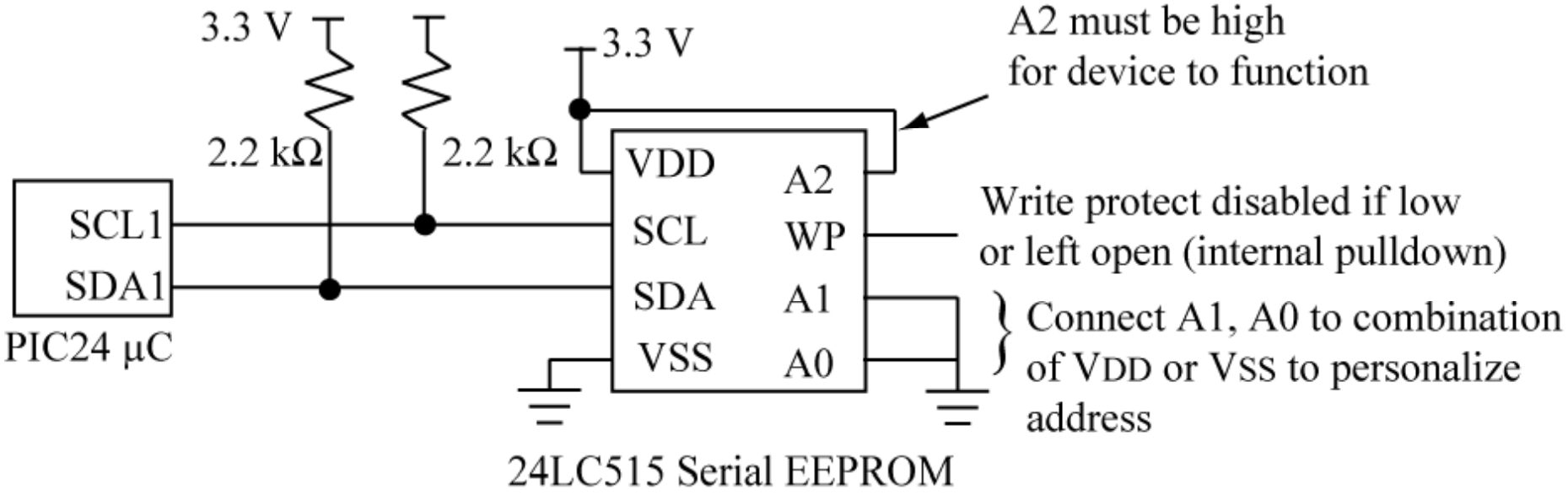
Read temperature and print result as hex, Celsius and Fahrenheit.

`While(1){}` loop is basically the same as used for DS1722.

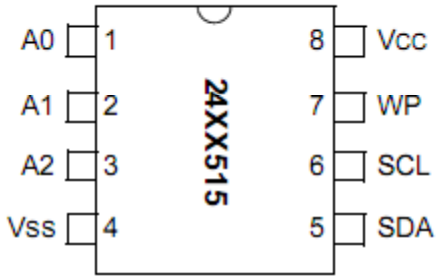
I²C Bus Activity when Reading DS1631 Temp



PIC24 µC Master to 24LC515 Serial EEPROM



EEPROM is 64 Ki x 8 , internally arranged as two separate 32 Ki x 8 memories.



NOTE: The diagram above is a logical layout, not the physical pinout, shown on the right.

24LC515 I²C Address Format, Write Operation

(a) Address byte format for 24LC515 serial EEPROM

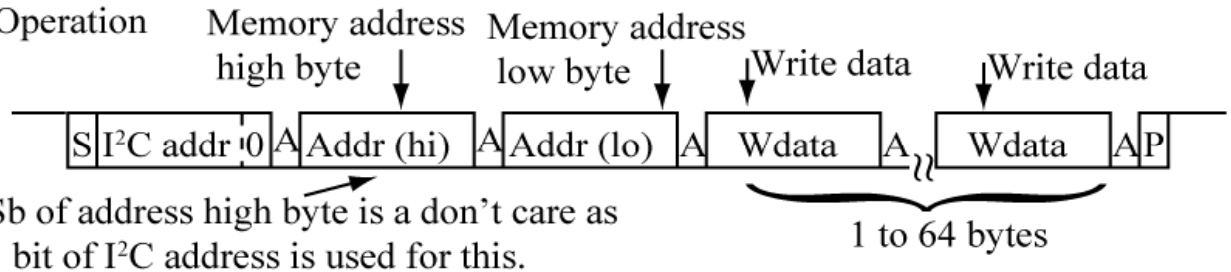
7	6	5	4	3	2	1	0
1	0	1	0	B	A1	A0	R/W#

B : Memory block select, if "0" then operation is to low memory block (0x0000-0x7FFF), if "1" then operation is to high memory block (0x8000-0xFFFF)

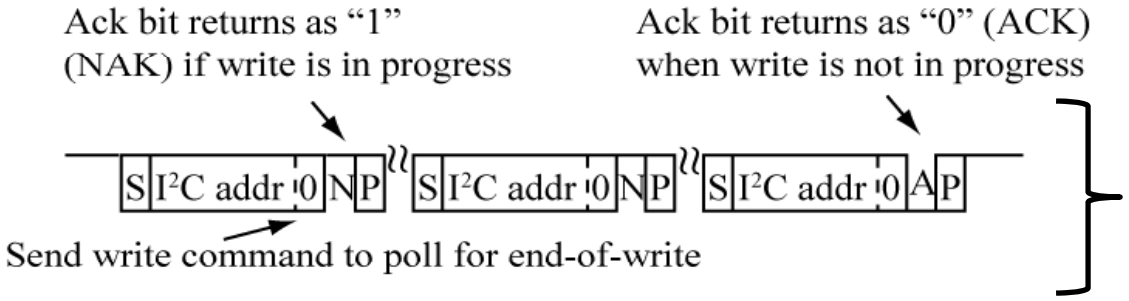
A1, A0: Used to personalized address, up to four LC515 EEPROMs can be on bus.

R/W#: "1" if read operation, "0" if write operation

(b) Write Operation



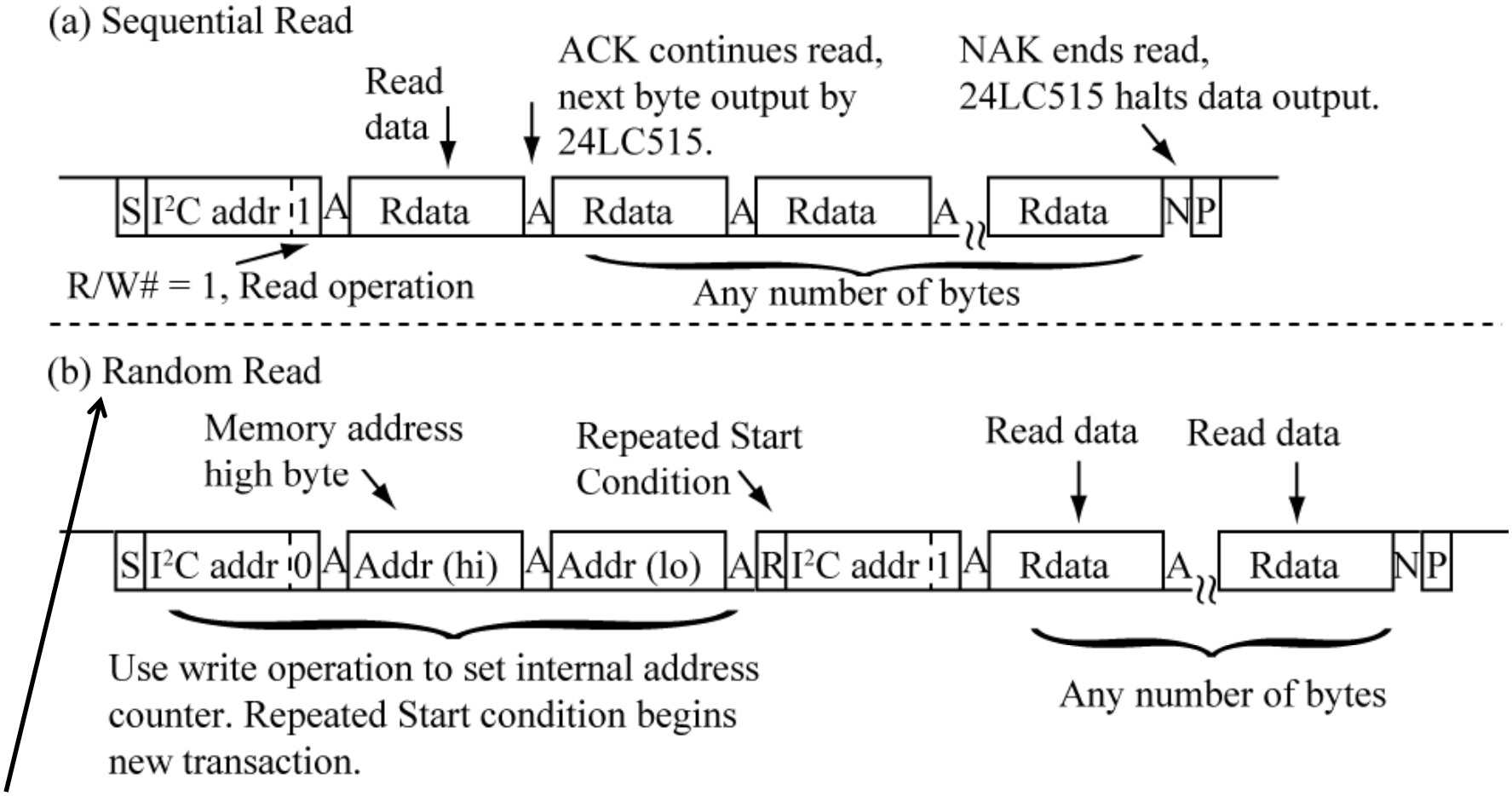
(c) End-of-Write Polling



Most efficient to write 64 bytes at a time as write takes 3 – 5 ms to complete.

Poll device to see when the write is finished.

24LC515 I²C Read Operation



We will do this to read the device, and always will read 64 bytes at a time.

Support Functions

```
#define EEPROM 0xA0    //LC515 address assuming both address pins tied low.
#define BLKSIZE 64
```

24LC151 address = 0b 1010 B A1 A0 R/W

0xA0 = 0b 1010 0 0 0 ?

Assume lower 32 Ki block.

```
void waitForWriteCompletion(uint8 u8_i2cAddr){
 uint8 u8_ack, u8_savedSWDTEN;
  u8_savedSWDTEN = _SWDTEN;
  _SWDTEN = 1;
  u8_i2cAddr = I2C_WADDR(u8_i2cAddr);  //write transaction, so R/W# = 0;
  do {
    startI2C1();
    u8_ack = putNoAckCheckI2C1(u8_i2cAddr);
    stopI2C1();
  } while (u8_ack == I2C_NAK);
  _SWDTEN = u8_savedSWDTEN;  //restore WDT to original state
}
```

Enable WDT to escape infinite loop, assumes WDT timeout is greater than EEPROM write time.

Send I²C address with R/W=0, check the ack bit that comes back.

Keep looping until get an ACK back.

Poll device to see when the write is finished.

Write 64 bytes in `*pu8_buf` to EEPROM starting at address `u16_MemAddr`

```
void memWriteLC515(uint8 u8_i2cAddr, uint16 u16_MemAddr, uint8 *pu8_buf){
  uint8 u8_AddrLo, u8_AddrHi;
  u8_AddrLo = u16_MemAddr & 0x00FF;
  u8_AddrHi = (u16_MemAddr >> 8);
  pu8_buf[0] = u8_AddrHi;
  pu8_buf[1] = u8_AddrLo;
  if (u16_MemAddr & 0x8000) {
    // if MSB set , set block select bit
    u8_i2cAddr = u8_i2cAddr | 0x08;
  }
  waitForWriteCompletion(u8_i2cAddr);
  writeNI2C1(u8_i2cAddr,pu8_buf,BLKSIZE+2);
}
```

Get the high, low bytes of the memory address.

First two bytes of `pu8_buf` are reserved for the EEPROM address.

Set the "B" bit of the I²C memory address if writing upper 32 Ki block.

Wait for last write to finish.

I²C block write transaction.

Write 64 bytes in **`*pu_buf`** to device

Support Functions

Read 64 bytes into `*pu8_buf` from EEPROM starting at address `u16_MemAddr`

```
void memReadLC515(uint8 u8_i2cAddr, uint16 u16_MemAddr, uint8 *pu8_buf){
  uint8 u8_AddrLo, u8_AddrHi;
  u8_AddrLo = u16_MemAddr & 0x00FF;
  u8_AddrHi = (u16_MemAddr >> 8);
  if (u16_MemAddr & 0x8000) {
    // if MSB set , set block select bit
    u8_i2cAddr = u8_i2cAddr | 0x08;
  }
  waitForWriteCompletion(u8_i2cAddr);
  //set address counter
  write2I2C1(u8_i2cAddr,u8_AddrHi, u8_AddrLo);
  //read data
  readNI2C1(u8_i2cAddr,pu8_buf, BLKSIZE);
}
```

Get the high, low bytes of the memory address.

Set the "B" bit of the I²C memory address if reading upper 32 Ki block.

Wait for last write to finish.

Set EEPROM's internal address counter.

I²C block read transaction.

Read 64 bytes from device, return in **`*pu_buf`**

Testing the 24LC515

```
int main (void) {
  uint8 au8_buf[64+2];  //2 extra bytes for address
  uint16 u16_MemAddr;
  uint8 u8_Mode;

  configBasic(HELLO_MSG);
  configI2C1(400);             //configure I2C for 400 KHz
  outString("\nEnter 'w' for write mode, anything else reads: ");
  u8_Mode = inCharEcho();
  outString("\n");
  u16_MemAddr = 0;     //start at location 0 in memory
  while (1) {
    uint8 u8_i;
    if (u8_Mode == 'w') {
      outString("Enter 64 chars.\n");
      //first two buffer locations reserved for starting address
      for (u8_i = 2;u8_i< 64+2;u8_i++) {
        au8_buf[u8_i] = inCharEcho();
      }
      outString("\nDoing Write\n");
      // write same string twice to check Write Busy polling
      memWriteLC515(EEPROM,u16_MemAddr, au8_buf); // do write
      u16_MemAddr = u16_MemAddr +64;
      memWriteLC515(EEPROM,u16_MemAddr,au8_buf); // do write
      u16_MemAddr = u16_MemAddr +64;
    } else {
      memReadLC515(EEPROM,u16_MemAddr,au8_buf); // do read
      for (u8_i = 0;u8_i< 64;u8_i++) outChar(au8_buf[u8_i]);
      outString("\nAny key continues read...\n");
      inChar();
      u16_MemAddr = u16_MemAddr + 64;
    }
  }
}
```

Get 64 bytes from the console.

Echo 64 bytes to the console.

In write mode, read 64 characters from the console, write to the 24LC515

In read mode, read 64 characters from the memory, write to the console.

Test Output

```
Reset cause: Power-on.
Device ID = 0x00000F1D (PIC24HJ32GP202), revision 0x00003002 (A3)
Fast RC Osc with PLL

i2c_serialeepromtest.c, built on Jun 28 2008 at 19:21:32

Enter 'w' for write mode, anything else reads: w
Enter 64 chars.
A person who graduates today and stops learning tommorrow is
Doing Write
Enter 64 chars.
uneducated the day after. Life long learning is very important.
Doing Write
Enter 64 chars.

Reset cause: MCLR assertion.
Device ID = 0x00000F1D (PIC24HJ32GP202), revision 0x00003002 (A3)
Fast RC Osc with PLL

i2c_serialeepromtest.c, built on Jun 28 2008 at 19:21:32

Enter 'w' for write mode, anything else reads: r
A person who graduates today and stops learning tommorrow is
Any key continues read...
A person who graduates today and stops learning tommorrow is
Any key continues read...
uneducated the day after. Life long learning is very important.
Any key continues read...
uneducated the day after. Life long learning is very important.
Any key continues read...
```

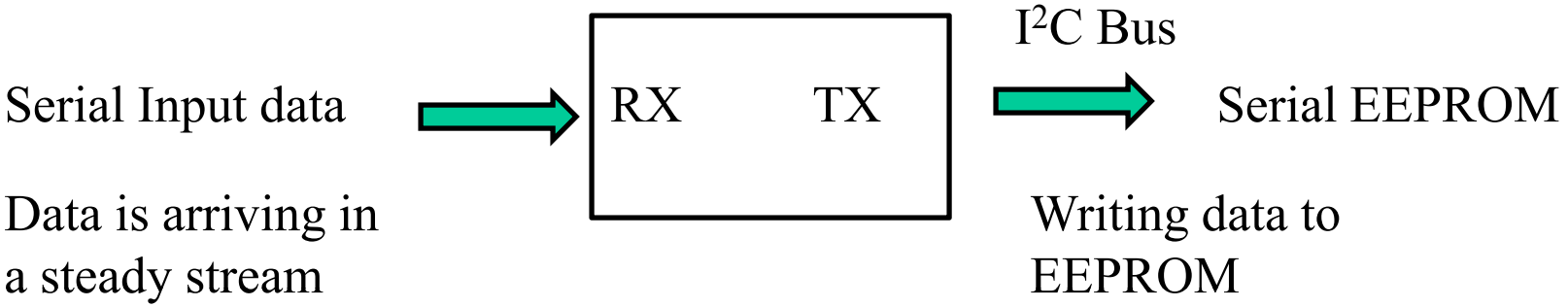
Two strings entered; each string saved twice to EEPROM.

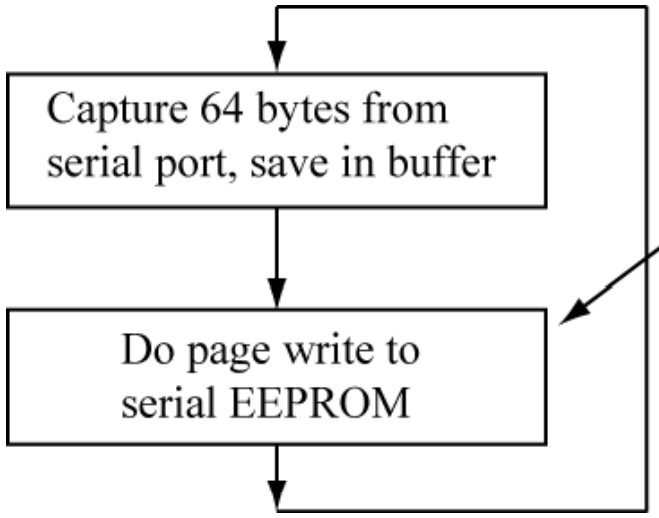
Reset button pressed.

Strings read back from EEPROM.

I²C Lab Assignment

Read streaming data from serial port, write to EEPROM





Interrupt service routine stores arriving bytes in buffer.

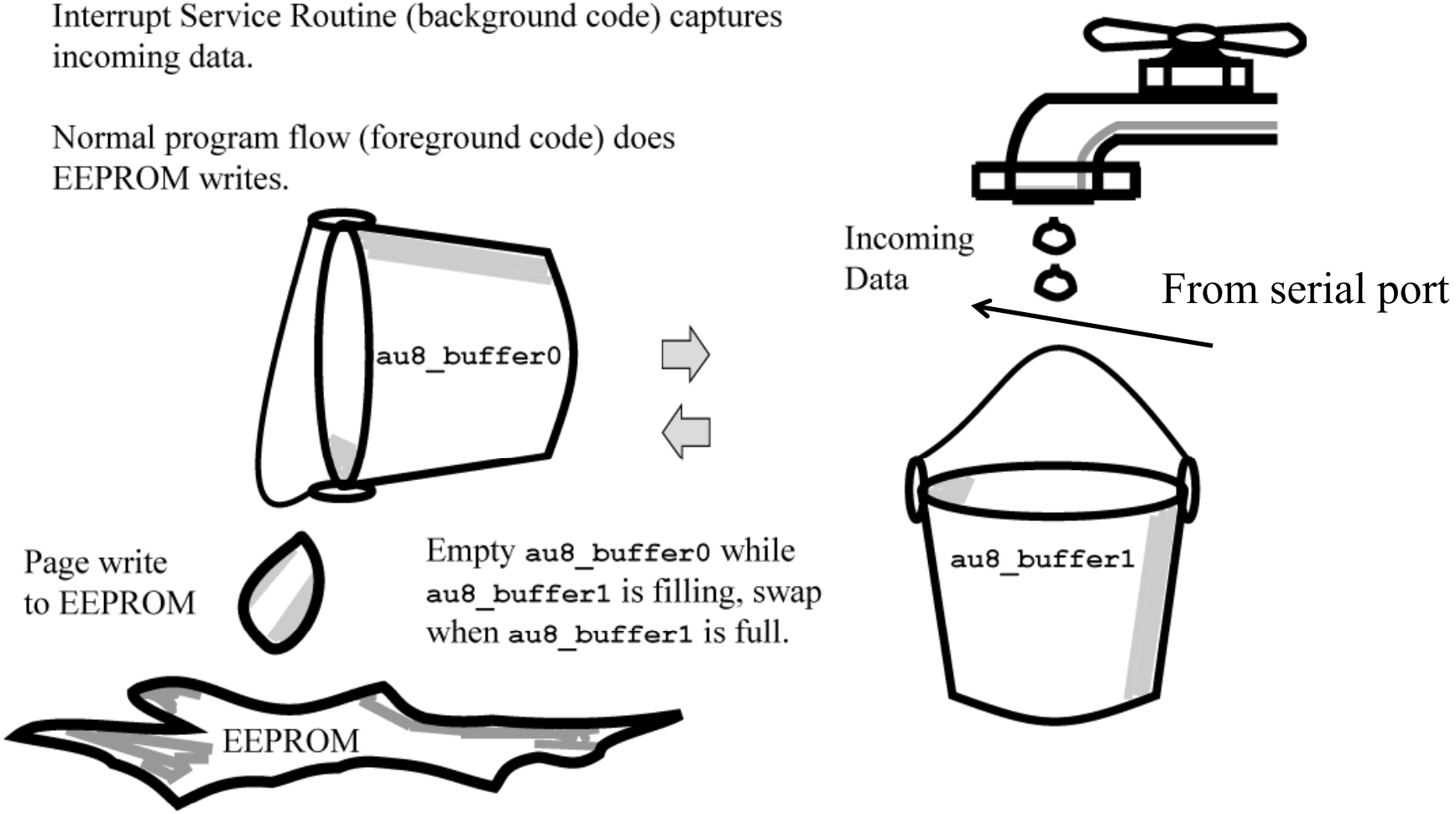
Problem: While writing bytes to serial EEPROM, more bytes are arriving! Where are they placed?

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

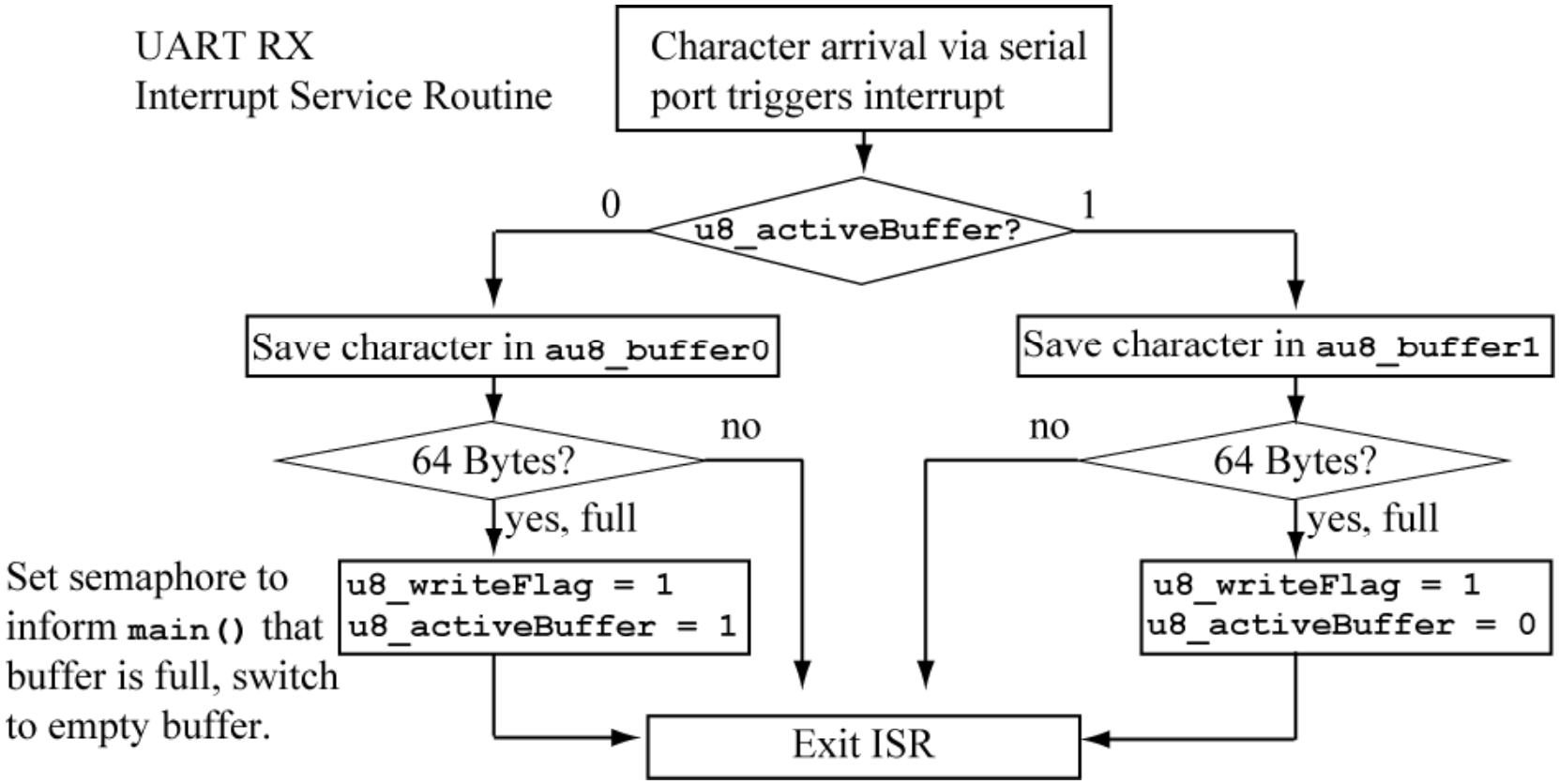
Double buffering for Streaming Data

Interrupt Service Routine (background code) captures incoming data.

Normal program flow (foreground code) does EEPROM writes.

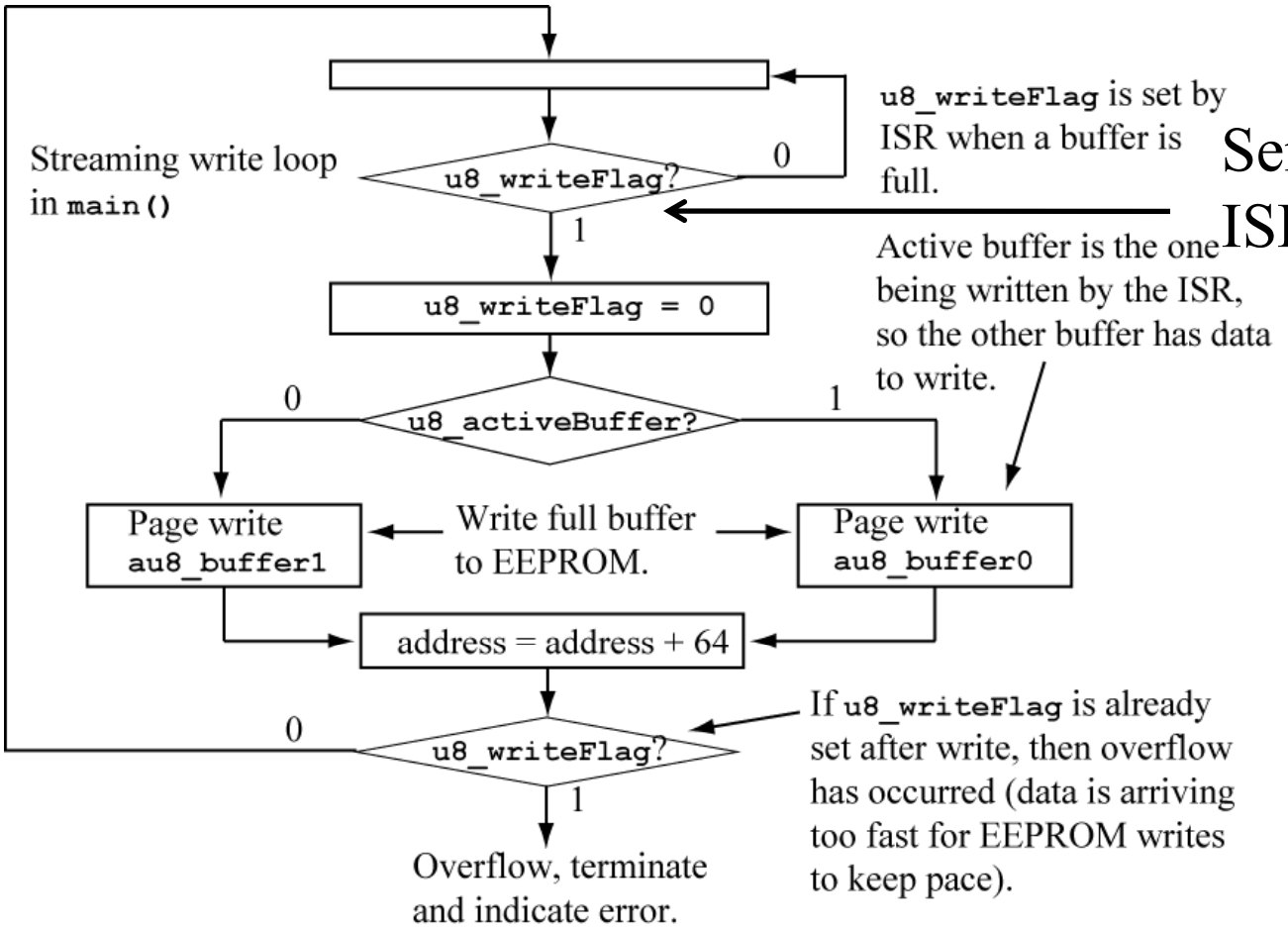


UART Receive ISR



This is a flowchart of the UART Receive ISR you must write. This is NOT a software FIFO, you are just placing input data in one of two separate buffers.

`main()` Code



Semaphore set by ISR.

This is a flowchart of the **`while(1){}`** loop in **`main()`** that you must write. The ISR places data into the buffer, and then the **`while(1){}`** loop writes it to EEPROM.

What do you have to know?

- SPI port operation
- SPI slave to PIC24 master communication, hookup
- DS1722 operation
- I2C Bus operation
- I2C primitive function operation and usage
- I2C Transaction function operation and usage
- DS1621 operation
- 24LC515 EEPROM Operation