

The I²C Bus

The General Bus Structure

- On embedded boards, at least one bus interconnects the other major components in the system

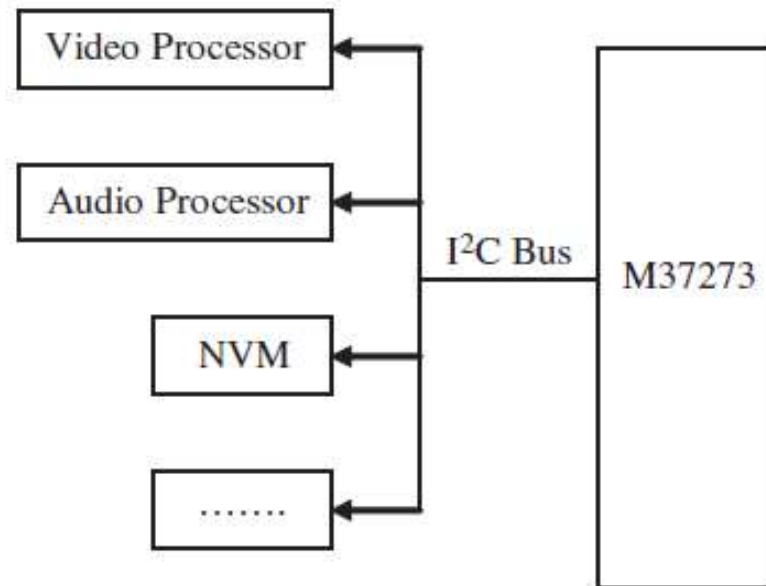
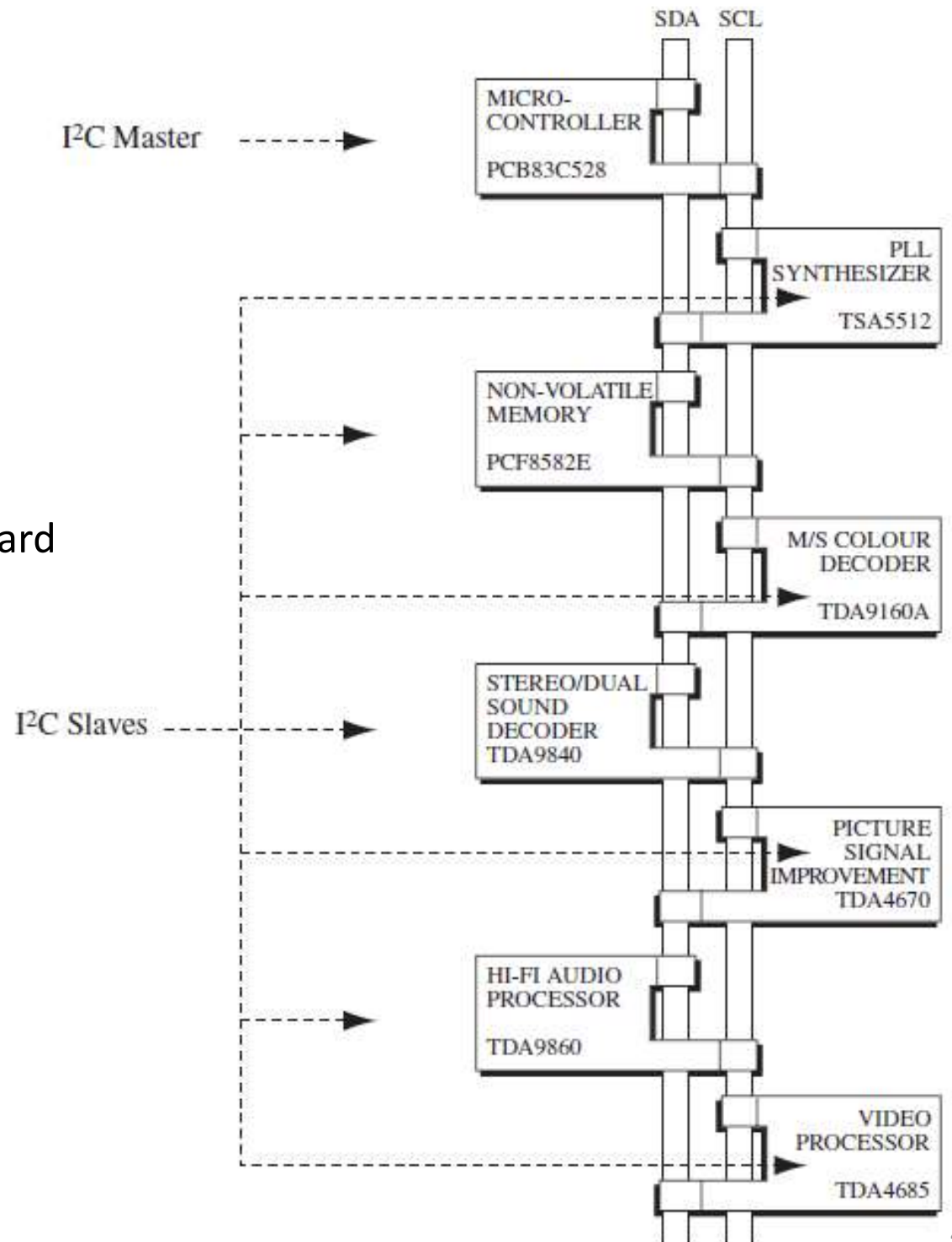


Figure 1: General bus structure

I²C Bus Example – communication lines

- The I²C (**Inter IC**) bus interconnects processors that have incorporated an I²C on-chip interface, allowing direct communication between these processors over the bus.
- A master/slave relationship between these processors exists at all times, with the master acting as a master transmitter or master receiver.
- As shown in Figure 2, the I²C bus is a two-wire bus with
 - one serial **data line** (**SDA**) and
 - one serial **clock line** (**SCL**).

Figure 2:
Sample analog TV board



I²C Bus Example – SCL Line

- The processors connected via I²C are each addressable by a unique address that is part of the data stream transmitted between devices.
- The I²C master initiates data transfer and generates the clock signals to permit the transfer.
- Basically, the SCL just cycles between HIGH and LOW (see Figure 3).

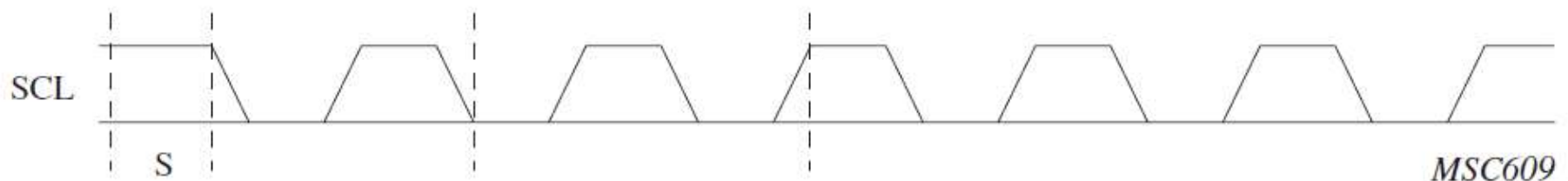


Figure 3: SCL cycles

I²C Bus Example – Start and Stop Conditions

- The master then uses the SDA line (as SCL is cycling) to transmit data to a slave.
- A session is started and terminated as shown in Figure 4, where a “START” is initiated when the master pulls the SDA port (pin) LOW while the SCL signal is HIGH, whereas a “STOP” condition is initiated when the master pulls the SDA port HIGH when SCL is HIGH.

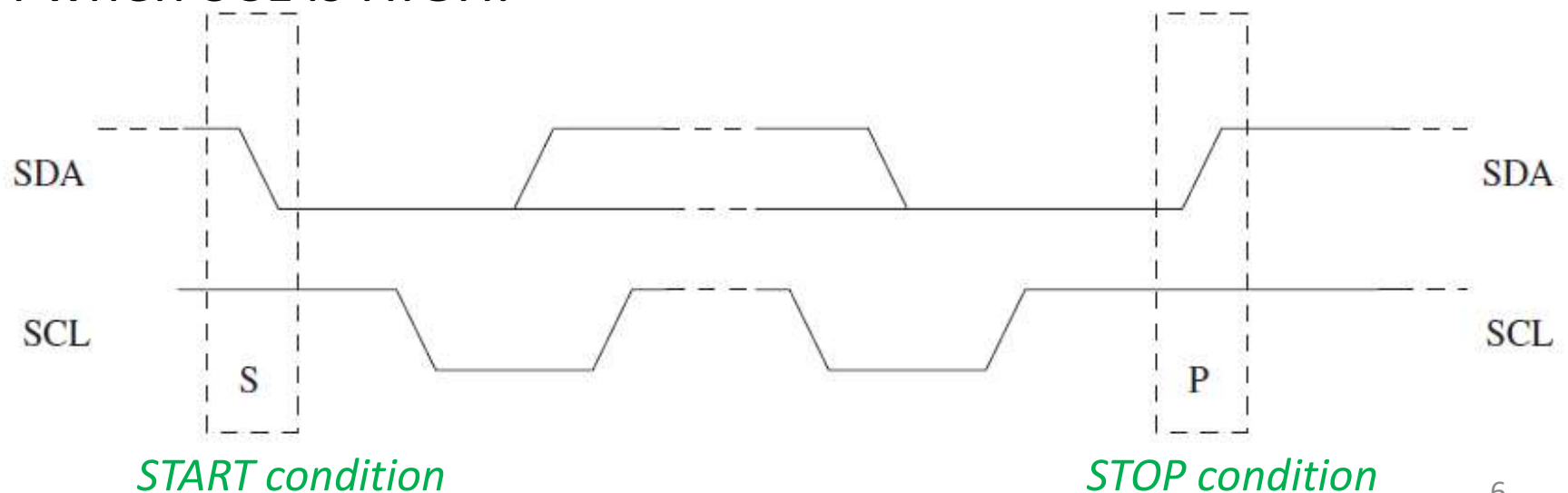


Figure 4: I2C START and STOP conditions

I2C Features

- With regard to the transmission of data, the I²C bus is a serial, 8-bit bus.
- This means that, while there is no limit on the number of bytes that can be transmitted in a session, only one byte (8 bits) of data will be moved at any one time, 1 bit at a time (serially).
- How this translates into using the SDA and SCL signals is that a data bit is “read” whenever the SCL signal moves from HIGH to LOW, edge to edge.
- If the SDA signal is HIGH at the point of an edge, then the data bit is read as a “1”. If the SDA signal is LOW, the data bit read is a “0”.

A Sample Byte Transfer

- An example of byte “00000001” transfer is shown in Figure 5

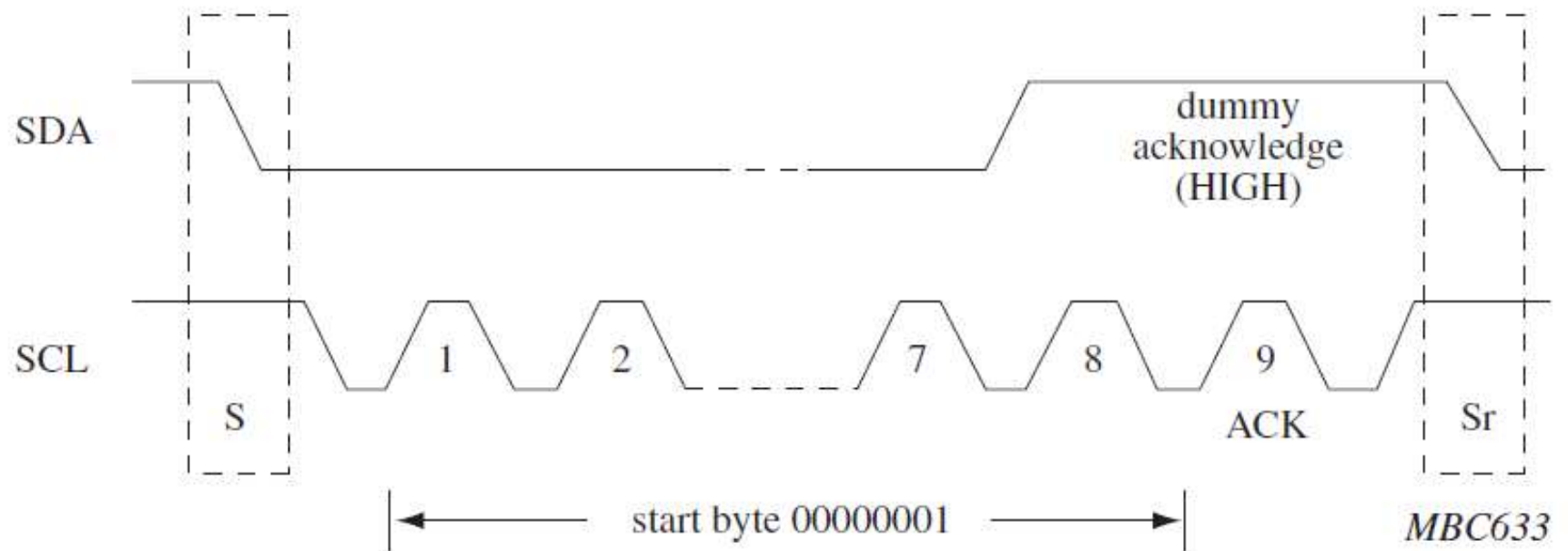


Figure 5: I²C data transfer example

A Complete Transfer Session

- Figure 6 shows an example of a complete transfer session.

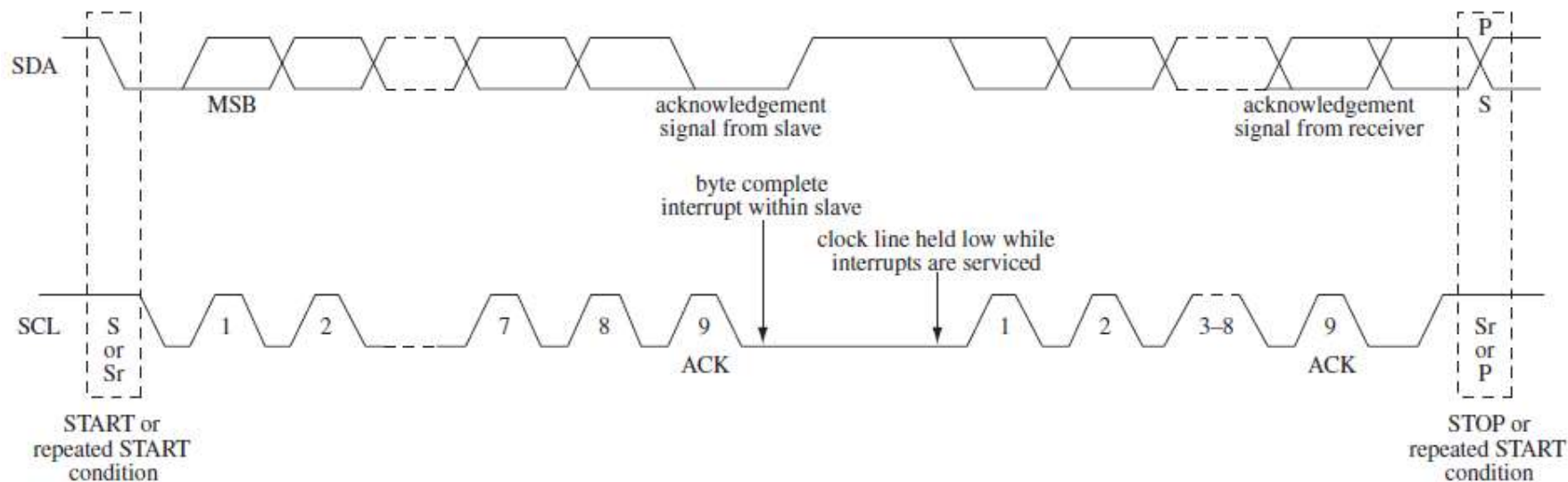


Figure 6: I2C complete transfer diagram

AN541 - Using a PIC16 as a Smart I²C Terminal

AN541

The I²C Bus

- Bi-directional communication (and in a full, multi-master system, collision detection, and clock synchronization) is facilitated through the use of a “wire-and” (i.e., active-low, passive-high) connection.
 - The standard mode I2C bus supports SCL clock rates up to 100 kHz.
 - The fast-mode I2C bus supports clock rates up to 400 kHz.
 - This application will support the 100 kHz (standard-mode) clock rate.
- Each device has a unique **seven bit address**, which the master uses to access each individual slave device.
- During normal communication, the SDA line is only permitted to change while the SCL line is low, thus providing two violation conditions (Figure 7) which are used to signal a start condition (SDA drops while SCL is high) and a stop condition (SDA rises while SCL is high), which frame a message.

The I²C Bus

- Each byte of a transfer is 9-bits long (see timing chart in the program listing).
 - The talker sends 8 data bits followed by a '1' bit.
 - The listener acknowledges the receipt of the byte and gives permission to send the next byte by inserting a '0' bit over the trailing '1'.
 - The listener may indicate "not ready for data" by leaving the acknowledge bit as a '1'.
- The clock is generated by the master only.
- The slave device must respond to the master within the timing specifications of the I2C definition otherwise the master would be required to operate in slow mode, which most software implementations of I2C masters do not actually support.
- The specified (standard-mode) t_{CL} is 4.7 μs , and t_{CH} is only 4 μs , so it would be extremely difficult to achieve the timing of a hardware slave device with a conventional microcontroller.

I²C timing

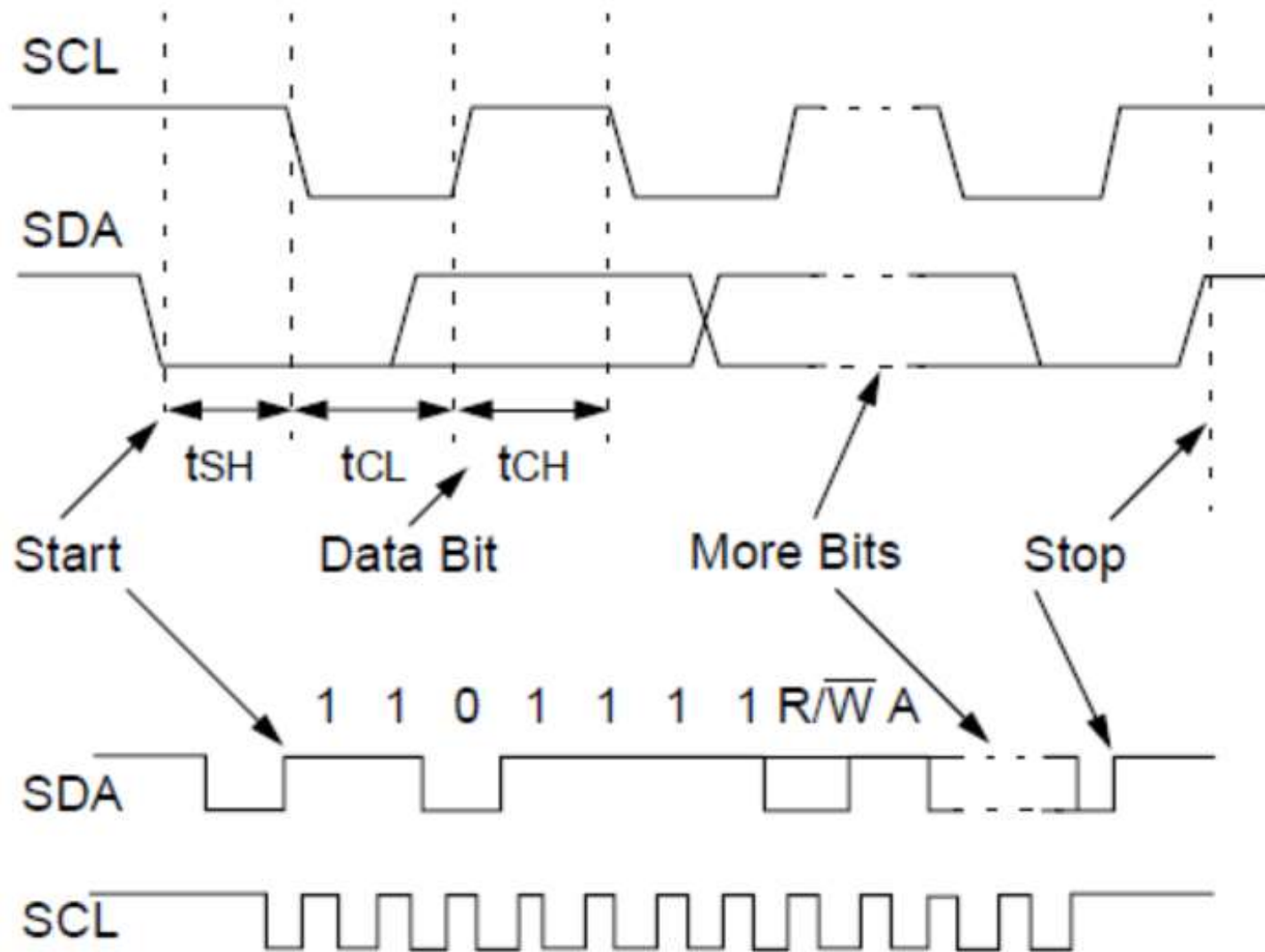


Figure 7: I2C Timing

Message Format

- A message is always initiated by the master, and begins with a start condition, followed by a slave address (7 MSbs) and direction bit (LSb = '1' for READ, '0' for WRITE).
 - The addressed slave must acknowledge this byte if it is ready to communicate any data.
 - If the slave fails to respond, the master should initiate a stop condition and retry.
- If the direction bit is '0' the next byte is considered the sub-address (this is an extension to I2C used by most multi-register devices).
 - The sub-address selects which "register" or "function" subsequent read or write operations will affect.
 - Any additional bytes will be received and stored in consecutive locations until a stop is sent.
 - If the slave is unable to process more data, it could terminate transfer by not acknowledging the last byte.

Message Format

- If the direction bit is '1', the slave will transfer successive bytes to the master (the master holds the line at '1'), while the master acknowledges each byte with a '0' in the ninth bit.
 - The master can terminate the transfer by not acknowledging the last byte, while the slave can stop the transfer by generating a stop condition.
- The start address of a read operation is set by sending a write request with a sub-address only (no data bytes).

Schematic of I²C Connections

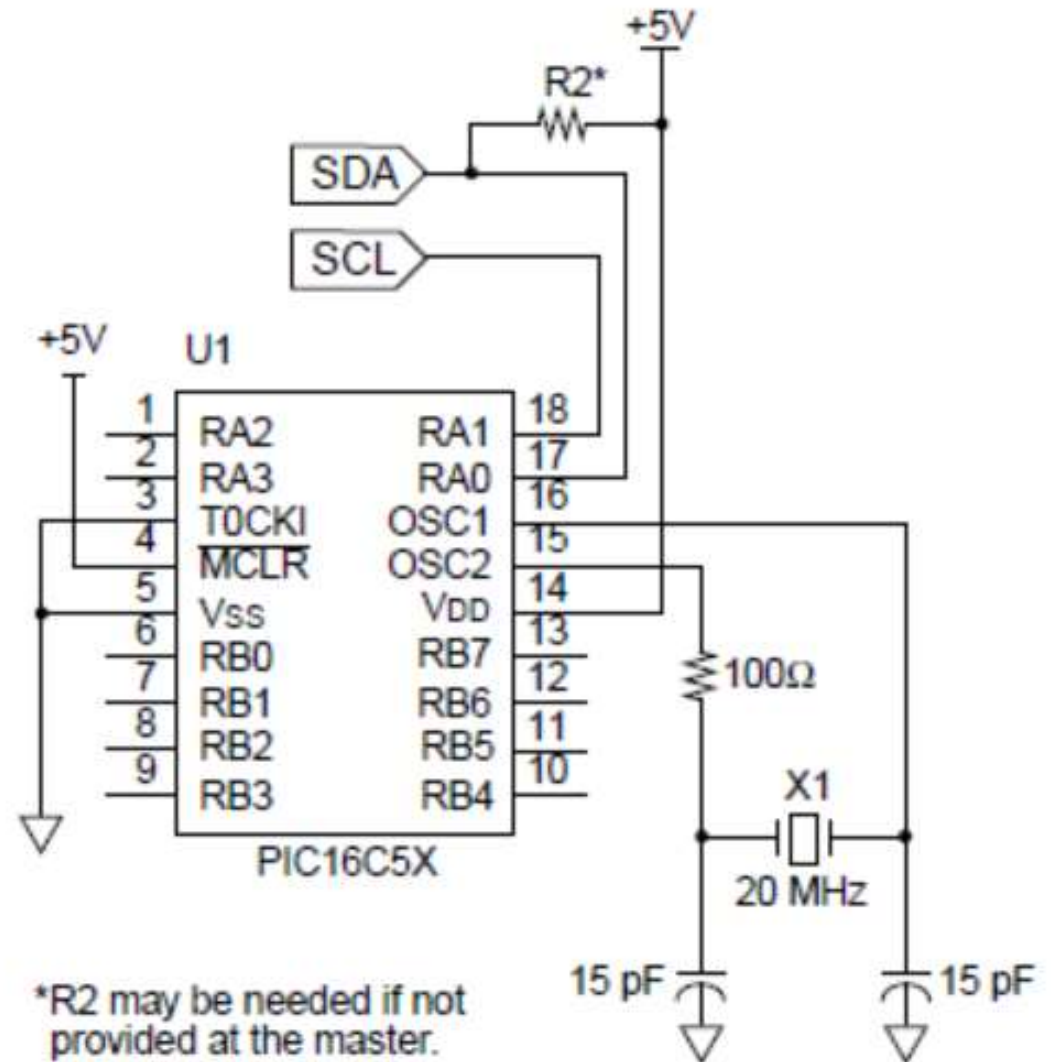


Figure 8: Schematics

I²C Device Flowchart

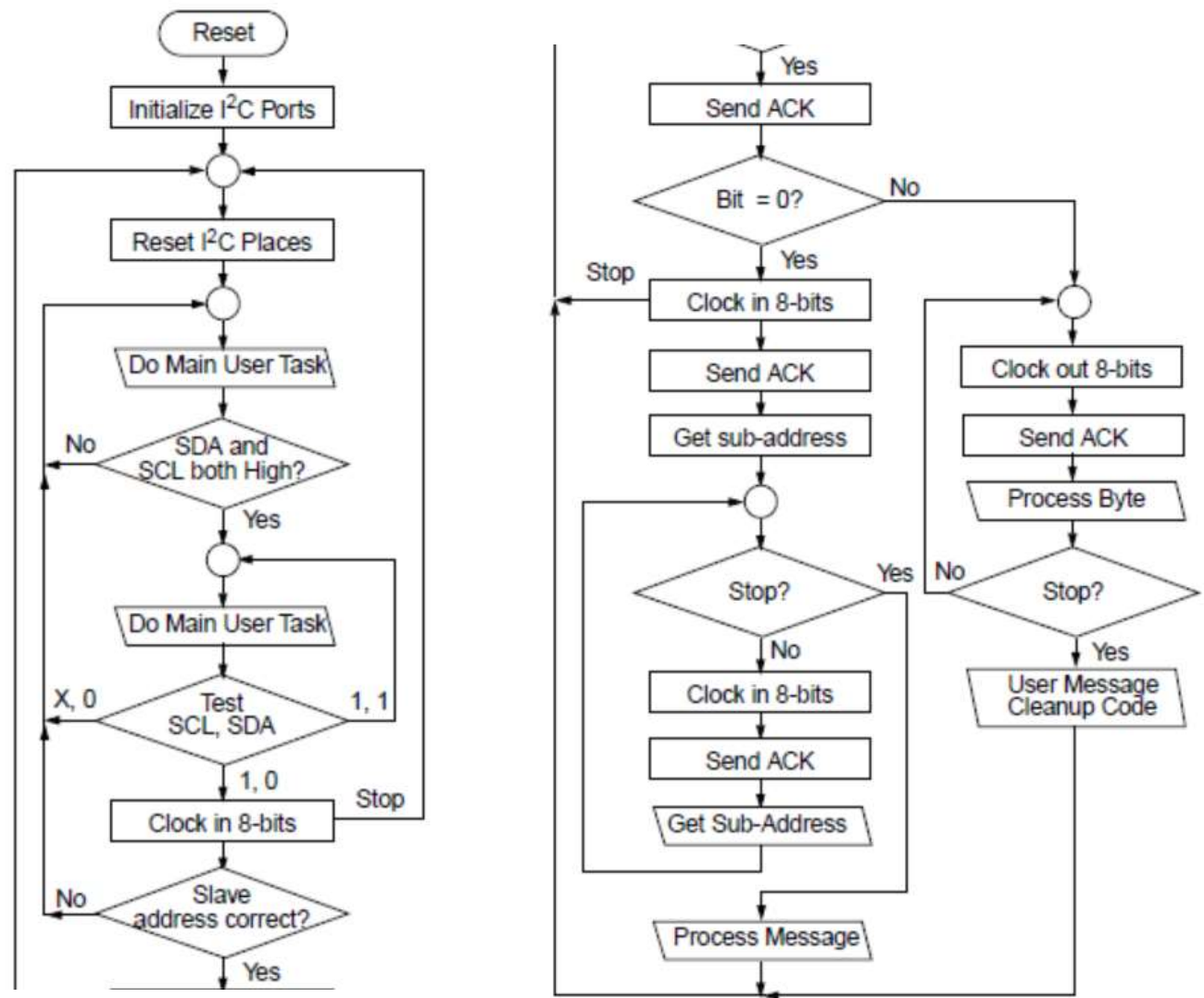


Figure 9: I2C Implementation Flowchart

Implementation

- The chip will respond to slave address "DEVICE_ADDRESS", which by default is D6₁₆ (D7₁₆ for read).
 - This address was chosen because it is the fourth optional address of a Philips PCF8573 clock /calender or a TDA8443 tippel video switch (unlikely that a product would contain four of those).
- The connections to the device are shown in Figure 8.
 - The use of RA0 for data input is required. Data is shifted directly out of the port. The code could be modified to make it port independent, but the loss of efficiency may hinder some real-time applications.
- This application emulates an I2C device with 8 registers, accessed as sub-addresses 1 through 8 (module 7), plus a data channel (0).
- The example code returns an ID string when the data channel is accessed.
- When bytes are written to sub-addresses other than 0, they are stored in I2CR0-I2CR7 (I2CR0 gets data written to sub-address 8).

Implementation

- When the initial subaddress is 0, the flag B_ID is set. This is used to indicate access to a special channel. In this case, the data channel is used to return an ID message, or output data to PORTB, however the natural extension would be to use this as a data I/O channel.
- To make the basic device routines easily adaptable to a variety of uses, **macros** are used to implement the application specific code.
 - This allows the developer the option of using subroutine calls, or in-line code to avoid the 4 clock cycle overhead and use of the precious stack.

Macros

Macro	User Code Function
USER_MAIN	Code to execute in the main loop while not in a message. If this code takes too long, tSH of 4 ms will be violated (Figure 1). The slave will simply miss the address, not acknowledge, and the master will retry.
USER_Q	This would be quick user code to implement real-time processes. In most applications, this macro would be empty. If used, this routine should be kept under 4 ms if possible.
USER_MSG	This would be user code to process a message. It is inserted after a message is successfully received.
USER_RECV	This would be user code to process a received byte. It allows the user to add extra code to implement special purpose sub-addresses such as FIFOs.
USER_XMIT	This would be user code to prepare an output byte. In the default routine, it traps sub-address 0 and calls the ID string function.

References:

1. I2C Bus Specification, Philips Corporation, December 1988.
2. The I2C bus and how to use it (including specification), Signetics/Philips Semiconductors, January 1992.
3. Fenger, Carl, "The Inter-Integrated Circuit (I2C) Serial Bus: Theory and Practical Consideration", Application Note 168, Philips Components, December 1988.
4. "24C16 16K CMOS Serial Electrically Erasable PROM", Microchip Data Book (1992).

I2C Reloaded

AN515

Introduction

- Unlike the 3-wire bus Serial EEPROMs, the 24CXX/ 85CXX communicate with any microcontroller only by a serial data I/O line (SDA) and a serial clock (SCL).
 - Chip select is not required.
 - Data transfer may be initiated only when the bus is not busy.
 - During such transfer, the data line (SDA) must remain stable whenever the clock line (SCL) is high.
 - Changes in the data line while the clock line is high are interpreted as a START or STOP condition.

Transfer Format

- After the START condition, a slave address is sent.
 - This address is 7-bits long, the eighth bit is a data direction bit.
- (R/W - a logical '0' indicates a transmission WRITE, a logical '1' represents a request for data READ.
- A data transfer is always terminated by a STOP condition generated by the master controller.
 - However, if a master still wishes to communicate on the bus, it can generate another START condition and address another slave without first generating a STOP condition.
 - Various combinations of read/write formats are then possible within such transfer.

A Simple Hardware Connection

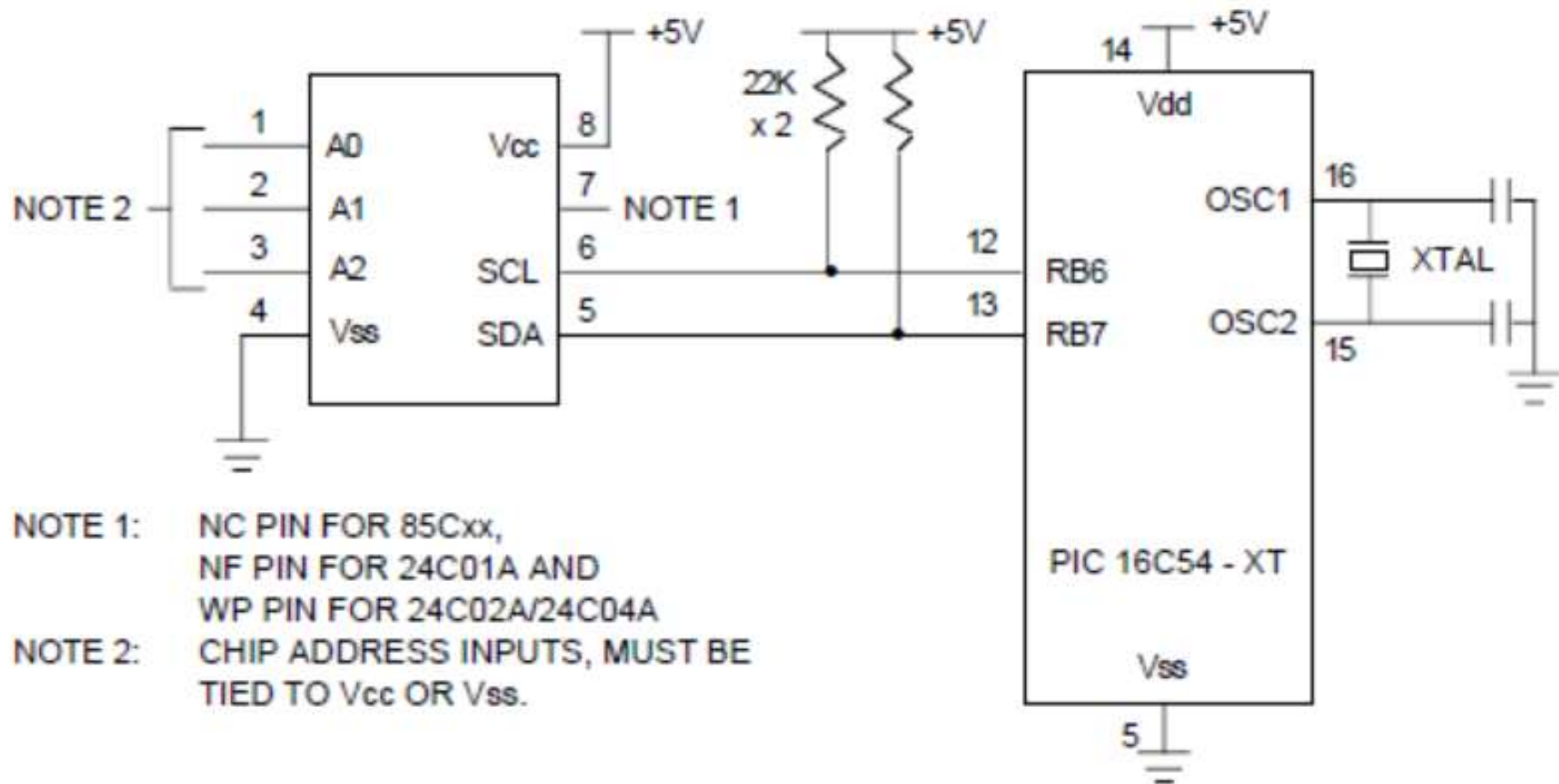


Figure 11: I²C Hardware Connection

Detail

- A simple hardware connection is illustrated in Figure 11.
- A maximum of eight 24C01A/ 24C02A/85C72/85C82's, or four 24C04A/85C92's can be addressed by a microcontroller on the same two wire bus without additional interfaces.
 - Each device is identified by its Chip Address and will only respond to the correct slave address.
 - Figures 13 and 14 describe how the bit stream is set up for READ and WRITE mode in the microcomputer programming software prior to sending it on the two wire serial bus.
 - The stop condition, after the write sequence, starts the internal self-timed write cycle which may last up to 6 milliseconds (.7 ms per byte).
 - Acknowledge signal should be monitored during this period.

Setting The Internal Word Address

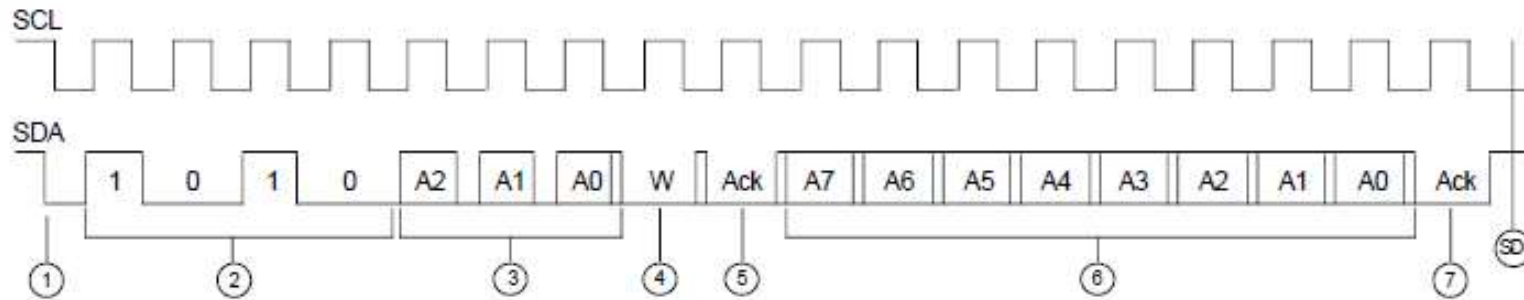


Figure 12 - Setting The Internal Word Address Of The 24cxx/85cxx

1. Start Condition
2. 4 Bit Device Code 1010 For 24Cxx/85Cxx
3. Device Address A0=PA, High Order Address Bit A8 For 24C04a/85C92
4. 0 = Write To Address Register
5. EEPROM Drives Bus Low With Acknowledgement (Timeout If No Response)
6. Address For Data, A7-A0 (MSb-LSb)
7. Bus Low For Acknowledgement

Byte Write Sequence

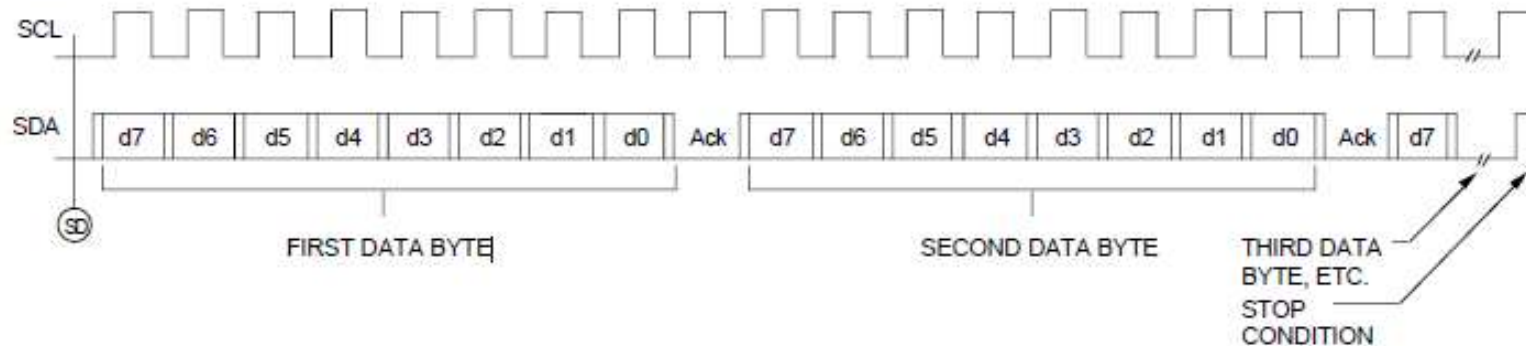


Figure 13 - Byte **Write** Sequence

Read Mode Sequence

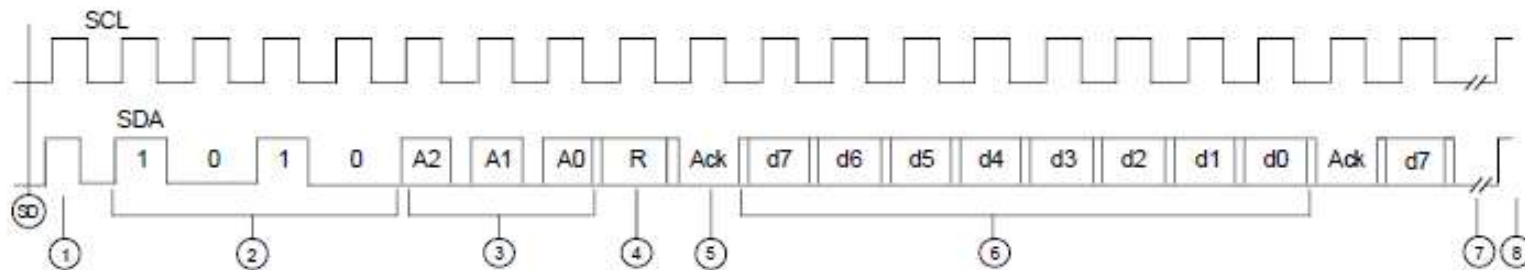


Figure 14 - Read Mode Sequence

1. Start Condition
2. 4 Bit Device Code 1010 For 24cxx/85cxx
3. Device Address A2, A1, A0. A0 = PA, High Order Address Bit A8 For 24C04a/85C92
4. 1 = Read Mode
5. Acknowledge From EEPROM
6. First Data Byte
7. Second Data Byte, Etc.
8. Stop Condition

Read Up To 128 Bytes (24C01A, 85C72)
Read Up To 256 Bytes (24C02A/04A, 85C82/92)

←

→

↺

microchip.wikidot.com

Apps

★


Bookmarks

⚙

Setări

📄

Retete

 **MICROCHIP** Developer Help

Search This Site

Q Search

Site updated 11 hours ago

2669 active pages

🏠 Home

📖 Training

🔧 Development Tools


⚙ Functions

📁 Projects

📦 Products

🛒 Store

❓ Help


MICROCHIP

Search This Site

Q Search

⚙

What's New?

[USB Flash Drive Audio Player Tutorial: Step 6](#)

26 Oct 2016, 23:03

[USB Flash Drive Audio Player Tutorial: Step 5](#)

26 Oct 2016, 23:02

[USB Flash Drive Audio Player Tutorial: Step 4](#)

26 Oct 2016, 23:01

💡

Tips & Tricks

[Finding the Cause of Reset](#)

14 Sep 2016, 17:33

[Keeping Unused Functions](#)

21 Jul 2016, 20:54

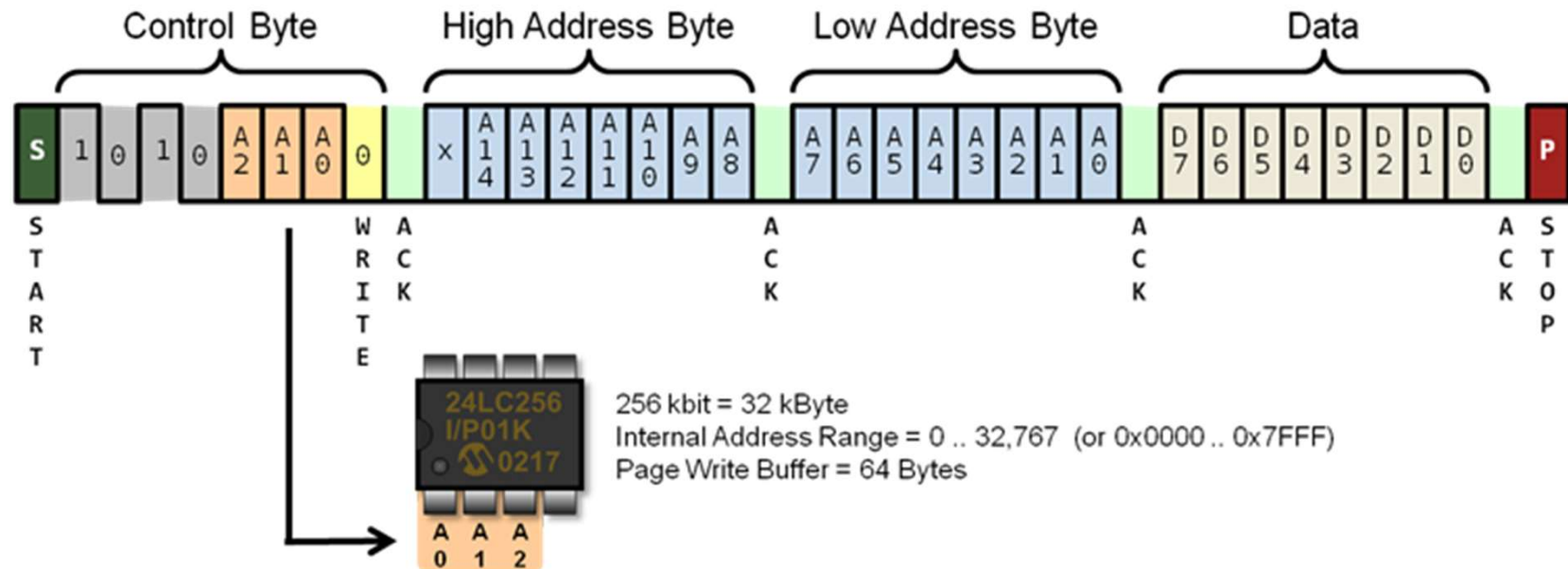
[Using High-Endurance Flash Memory](#)

31 May 2016, 17:12

I²C Chip Select

- The I²C bus selects a destination device by its address.
 - In some cases, a device might have a fixed address, meaning only one device of that type can appear on the bus.
 - However in most cases, a device's address is determined by its hardware configuration.

I²C Chip Select



- In the image above, a 24LC256 serial EEPROM is depicted showing its three chip select pins which correspond to the three address bits in the I²C signal.
- The address the device will respond to depends on which pins are connected to the positive supply and which pins are connected to ground.

I²C Start and Stop Conditions

- Ordinarily, data on the I²C SDA line is only allowed to change states when the SCL (clock) line is **low**.
 - This is because transitions on the SDA line while the SCL line is **high** have special meanings (see next slide).

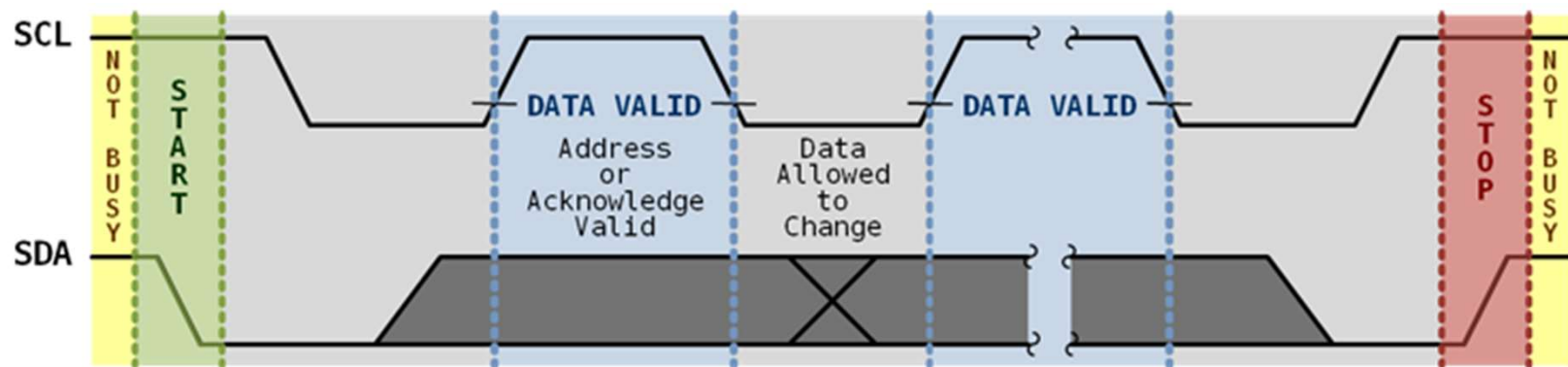
I²C Start and Stop Conditions

- **Start Condition**

- A start condition is defined as a transition from high to low on the SDA line while the SCL line is **high**.

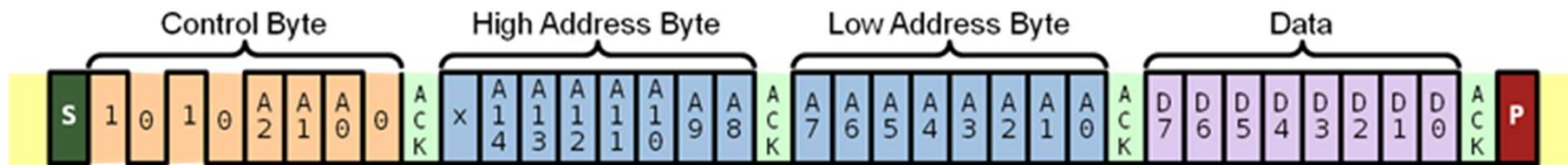
- **Stop Condition**

- A stop condition is defined as a transition from low to high on the SDA line while the SCL line is **high**.



I²C Byte Write

- An I²C byte write is used to write a byte of data to a specific address.



EEPROM Byte Write Procedure (I2C)

1. Check for bus idle
2. Send start condition, wait for it to complete
3. Write control byte (with device address)
4. Check for bus idle
5. Write high byte of memory address
6. Check for bus idle
7. Write low byte of memory address
8. Check for bus idle
9. Write byte of data
10. Check for bus idle
11. Send stop condition, wait for it to complete
12. Wait for write cycle to complete (ACK from EEPROM)

EEPROM Byte Write Procedure (I2C)

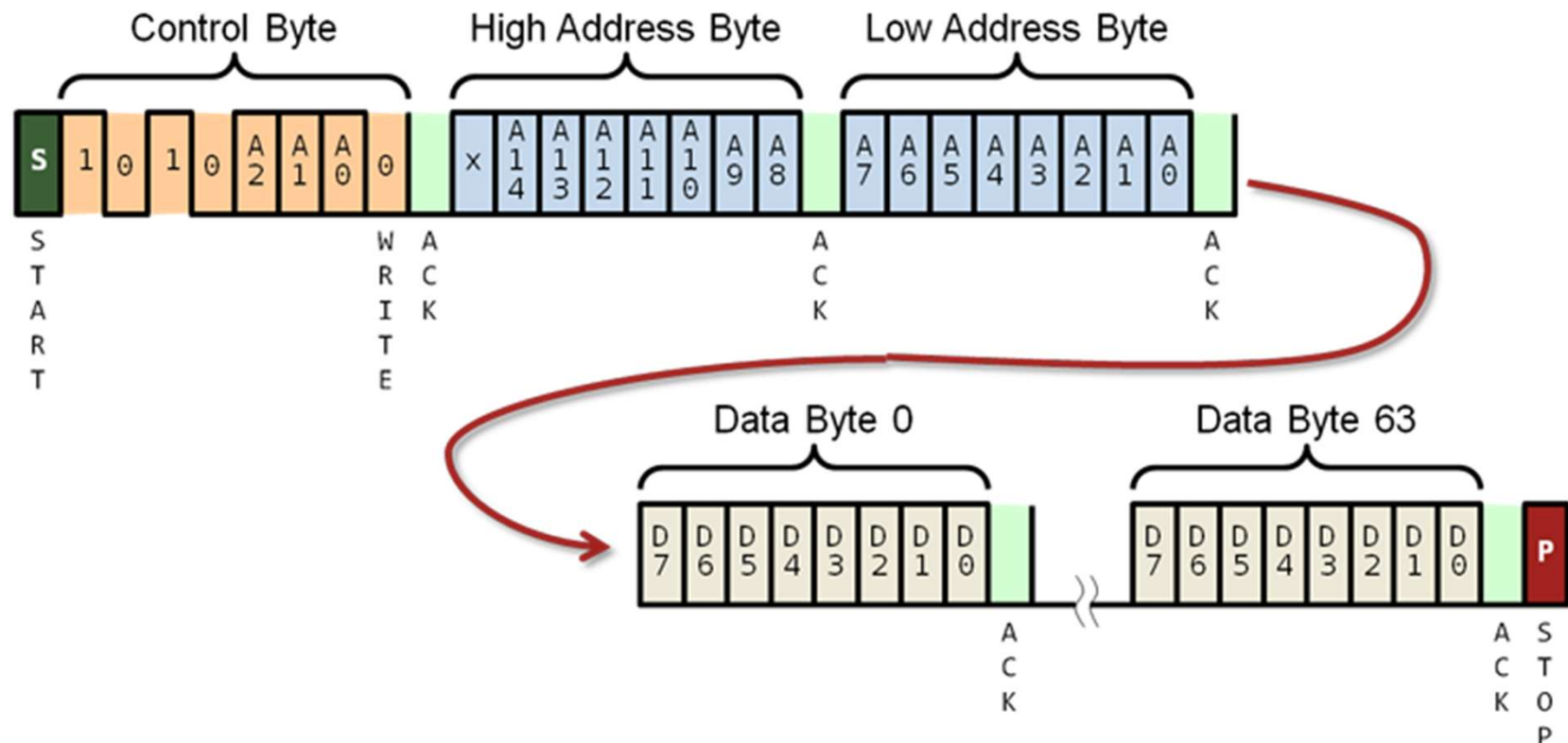
IdleI2C();	// Ensure module is idle
StartI2C();	// Initiate START condition
while (SSPCON2bits.SEN);	// Wait until START condition is over
WriteI2C(0xA0);	// Write 1 byte - R/W bit should be 0
IdleI2C();	// Ensure module is idle
WriteI2C(HighAddress);	// Write high address byte to EEPROM
IdleI2C();	// Ensure module is idle
WriteI2C(LowAddress);	// Write low address byte to EEPROM
IdleI2C();	// Ensure module is idle
WriteI2C(Data);	// Write data byte to EEPROM
IdleI2C();	// Ensure module is idle
StopI2C();	// Send STOP condition
while (SSPCON2bits.PEN);	// Wait until STOP condition is over
while (EEAckPolling(0xA0));	// Wait for write cycle to complete

EEPROM Byte Write C Code PIC18 (I2C)

```
unsigned char HDByteWriteI2C( unsigned char ControlByte, unsigned char HighAdd,
unsigned char LowAdd, unsigned char data )
{
    IdleI2C();                // ensure module is idle
    StartI2C();               // initiate START condition
    while ( SSPCON2bits.SEN ); // wait until start condition is over
    WriteI2C( ControlByte );  // write 1 byte - R/W bit should be 0
    IdleI2C();                // ensure module is idle
    WriteI2C( HighAdd );      // write address byte to EEPROM
    IdleI2C();                // ensure module is idle
    WriteI2C( LowAdd );       // write address byte to EEPROM
    IdleI2C();                // ensure module is idle
    WriteI2C ( data );        // Write data byte to EEPROM
    IdleI2C();                // ensure module is idle
    StopI2C();                // send STOP condition
    while ( SSPCON2bits.PEN ); // wait until stop condition is over
    while (EEAckPolling(ControlByte)); //Wait for write cycle to complete
    return ( 0 );             // return with no error
}
```

I²C Page Write

- An I²C page write is used to write a stream of 64 bytes to the device
 - some devices might have different page sizes.

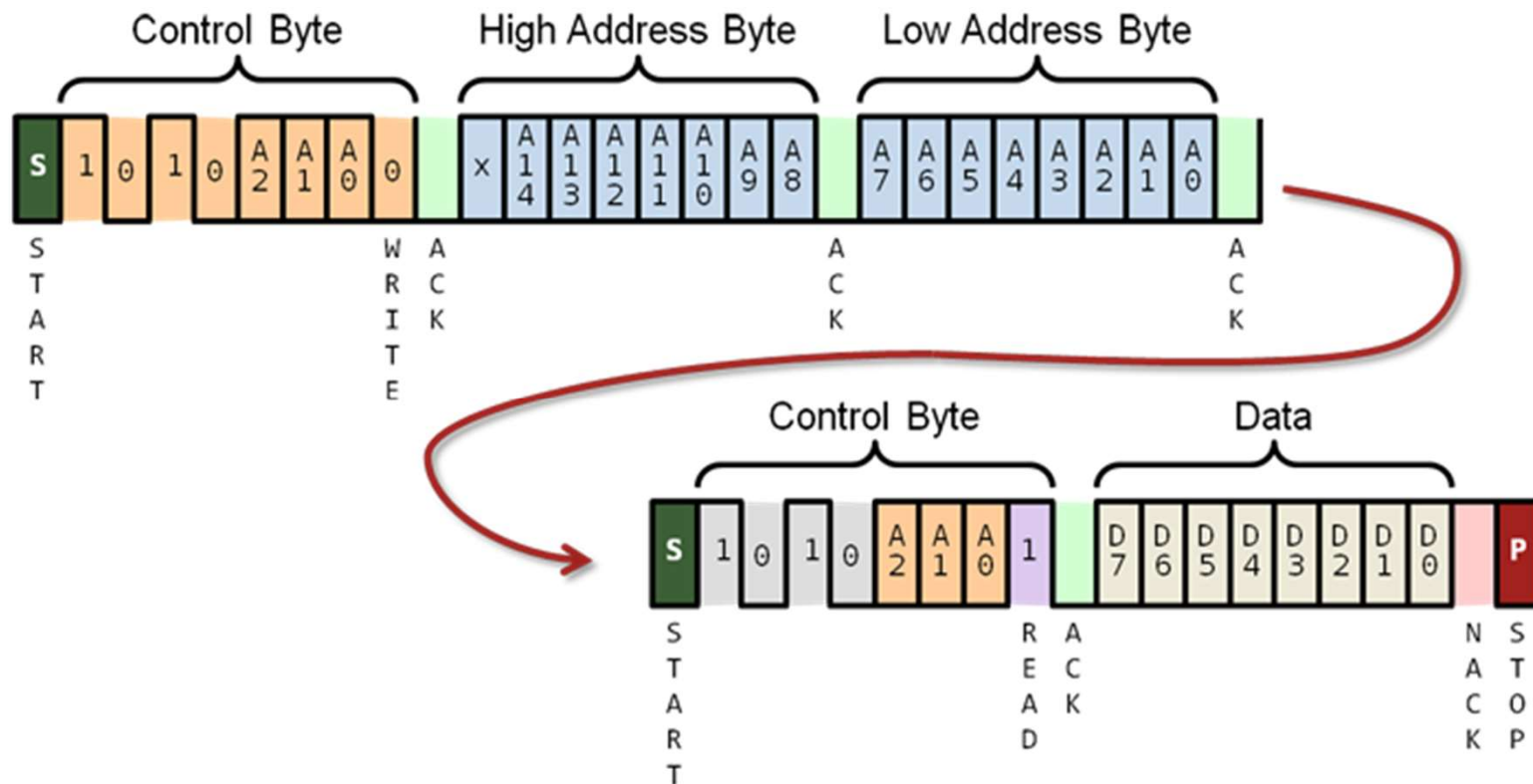


EEPROM Page Write Procedure (I2C)

1. Check for bus idle
2. Send start condition, wait for it to complete
3. Write control byte (with device address)
4. Check for bus idle
5. Write high byte of memory address
6. Check for bus idle
7. Write low byte of memory address
8. Check for bus idle
9. Write byte of data
10. Check for bus idle - goto 9 until 64 bytes written
11. Send stop condition, wait for it to complete
12. Wait for write cycle to complete (ACK from EEPROM)

I²C Random Read

- An I2C random read can be used
 - to read a single byte of data from a specific address,
 - or it can be used to move the address pointer for use in a sequential read.



EEPROM Byte Write Procedure (I2C)

1. Check for bus idle
2. Send start condition, wait for it to complete
3. Write control byte (with device address)
4. Check for bus idle
5. Write high byte of memory address
6. Check for bus idle
7. Write low byte of memory address
8. Check for bus idle
9. Send restart condition - wait for it to complete
10. Write control byte
11. Check for bus idle
12. Read **one** byte
13. Send NACK condition - wait for ACK sequence to complete
14. Send stop condition, wait for it to complete
15. Wait for write cycle to complete (ACK from EEPROM)

Example Code (PIC18)

- Specific I²C function names and parameters may differ depending on your target device, compiler and/or peripheral library.
- The first half is the same as for a byte write while the second half is like a current address read, but with a "Repeated Start" so the bus doesn't go idle.

I²C Random Read

IdleI2C();	// Ensure module is idle
StartI2C();	// Initiate START condition
while (SSPCON2bits.SEN);	// Wait until START condition is over
WriteI2C(0xA0);	// Write 1 byte - R/W bit should be 0
IdleI2C();	// Ensure module is idle
WriteI2C(HighAddress);	// Write high address byte to EEPROM
IdleI2C();	// Ensure module is idle
WriteI2C(LowAddress);	// Write low address byte to EEPROM
IdleI2C();	// Ensure module is idle
RetartI2C();	// Initiate RESTART condition
while (SSPCON2bits.RSEN);	// Wait until RESTART condition is over
WriteI2C(0xA1);	// Write 1 byte - R/W bit should be 1 for read
IdleI2C();	// Ensure module is idle
getSI2C(&dataOut, 1);	// Read in one byte
NotAckI2C();	// Send NACK condition
while (SSPCON2bits.ACKEN);	// Wait until ACK sequence is over
StopI2C();	// Send STOP condition
while (SSPCON2bits.PEN);	// Wait until STOP condition is over

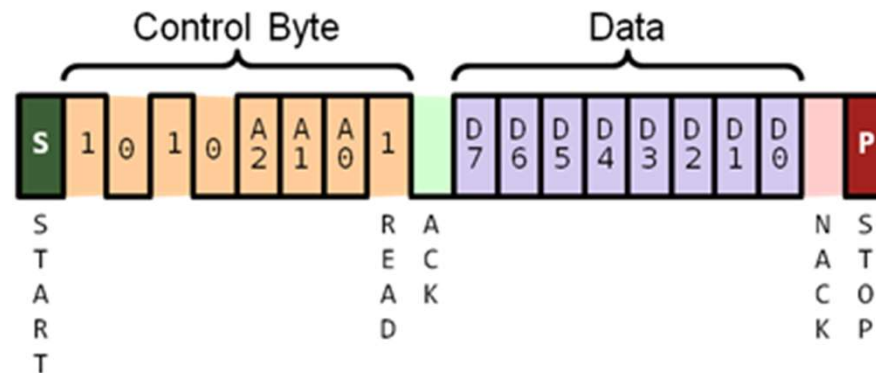
EEPROM Byte Random Read

C Code PIC18 (I2C)

```
unsigned char HDByteReadI2C( unsigned char ControlByte, unsigned char HighAdd, unsigned char
LowAdd, unsigned char *data, unsigned char length )
{
    IdleI2C();           // ensure module is idle
    StartI2C();          // initiate START condition
    while ( SSPCON2bits.SEN ); // wait until start condition is over
    WriteI2C( ControlByte ); // write 1 byte
    IdleI2C();           // ensure module is idle
    WriteI2C( HighAdd ); // WRITE word address to EEPROM
    IdleI2C();           // ensure module is idle
    while ( SSPCON2bits.RSEN ); // wait until re-start condition is over
    WriteI2C( LowAdd ); // WRITE word address to EEPROM
    IdleI2C();           // ensure module is idle
    RestartI2C();        // generate I2C bus restart condition
    while ( SSPCON2bits.RSEN ); // wait until re-start condition is over
    WriteI2C( ControlByte | 0x01 ); // WRITE 1 byte - R/W bit should be 1 for read
    IdleI2C();           // ensure module is idle
    getsI2C( data, length ); // read in multiple bytes
    NotAckI2C();         // send not ACK condition
    while ( SSPCON2bits.ACKEN ); // wait until ACK sequence is over
    StopI2C();           // send STOP condition
    while ( SSPCON2bits.PEN ); // wait until stop condition is over
    return ( 0 );        // return with no error
}
```

I²C Current Address Read

- I²C current address reads are used to read one byte from the current address.
- The current address will be one beyond the address that was last acted upon.



EEPROM Current Address Read Procedure

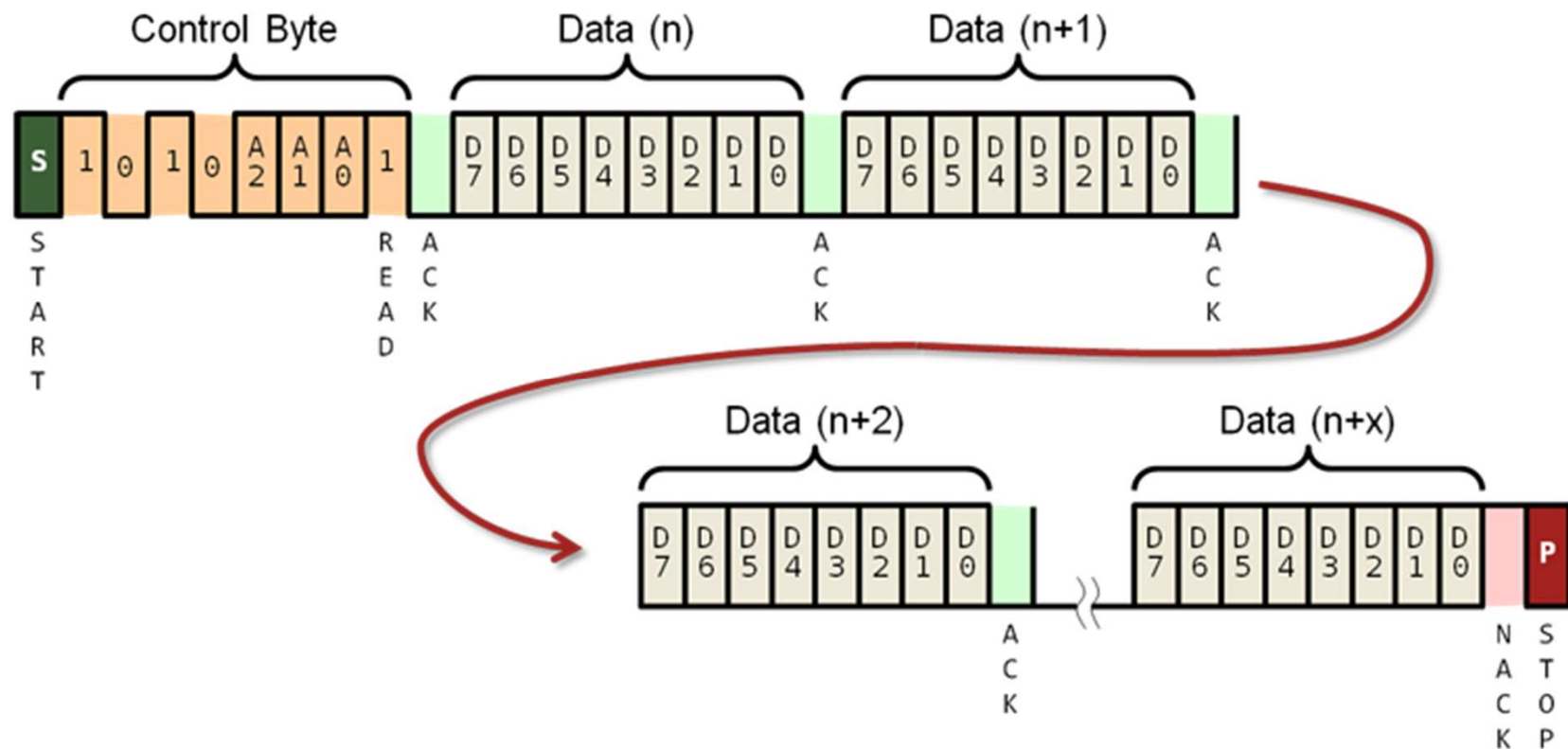
1. Send start condition - wait for it to complete
2. Send control byte
3. Check for bus idle
4. Read in **one** byte
5. Send NACK, wait until ACK sequence complete
6. Send stop condition - wait for it to complete

Example Code (PIC18)

StartI2C(); while (SSPCON2bits.SEN);	// Initiate START condition // Wait until START condition is over
WriteI2C(0xA1);	// Write 1 byte - R/W bit should be 1 for read
IdleI2C();	// Ensure module is idle
getI2C(&dataOut, 1);	// Read in one byte
NotAckI2C(); while (SSPCON2bits.ACKEN);	// Send NACK condition // Wait until ACK sequence is over
StopI2C(); while (SSPCON2bits.PEN);	// Send STOP condition // Wait until STOP condition is over

I²C Sequential Read

- An I²C sequential read begins reading at whatever the current address is inside the device.
 - You simply keep reading in bytes (interspersed with idle checks) until you've read enough.
- To **stop** the reading, send a NACK and then a STOP condition.



EEPROM Sequential Read Procedure

1. Check for bus idle
2. Send start condition, wait for it to complete
3. Write control byte (with device address)
4. Check for bus idle
5. Read one byte - goto 4 until you've had enough data
6. Send NACK condition - wait for ACK sequence to complete
7. Send stop condition, wait for it to complete

Proteus

- Microchip AN245 - Interfacing The MCP23016 I/O Expander With The PIC16F877A
 - VSM for PIC Micro -> AppNotes -> AN245
 - VSM for PIC Micro -> PIC16 -> MCP23016 & PIC16F630
 - VSM for PIC Micro -> PIC18 -> MCP3421 & PIC18
- Advanced Simulation
 - VSM for PIC 16 -> PIC Serial memory Example
 - I2C Memory Test (with waveforms):