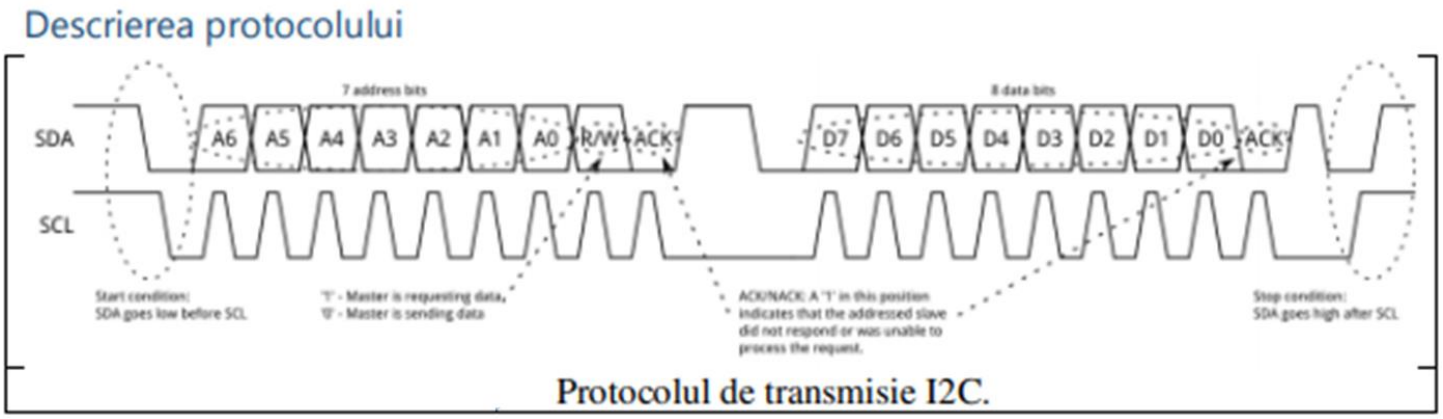
Interfața **Inter-Integrated Circuit (IIC sau I2C)**

Interfața I2C sau IIC (Inter-Integrated Circuit) este un tip de magistrală (se mai numește și protocol) pentru transmisie de date sub formă serială *master-slave*, fiind creat pentru a permite mai multor circuite integrate "slave" sa comunice cu unul sau mai mute cipuri "master". Spre deosebire de UART, I2C permite comunicarea între mai mult de două dispozitive. Unul dintre dispozitive va fi master, iar el va comunica cu un slave, dar ulterior rolurile se pot schimba.
A fost inițial dezvoltată de către firma Philips (în prezent NXP) în anul 1982, ulterior specificația suferind mai multe modificări (au apărut diferite versiuni):
vers. 1.0 (1992): permite două tipuri de transmisie: Normal mode (rata de transfer: 100 kbps) respectiv Fast mode (rata de transfer: 400 kbps) precum și adresarea pe 10 biți (max 1024 dispozitive)
vers. 2.0 (1998): s-a adăugat o nouă rată superioară de transfer (High speed mode) de 3,4 Mbps
vers. 2.1 (ianuarie 2000) îmbunătățiri aduse versiunii anterioare
vers. 3.0 (2007) s-a inclus o nouă rată de transfer numita Fast mode plus de 1 Mbps
vers. 4.0 (2012) rata de transfer Ultra-Fast mode de 5 Mbps

Acest tip de comunicare a fost intenționat pentru a fi folosit doar pe distante mici de comunicare si asemenea protocolului UART sau RS232 are nevoie doar de 2 fire de semnal pentru a trimite/ primii informații:
SCL - Serial Clock Line (linie de ceas seriala) - semnalul de ceas
SDA - Serial Data Line (linie de date seriala) - semnalul de date
Semnalul de ceas este întotdeauna generat de bus masterul curent.
(Unele componente slave vor forța ceasul la nivelul low uneori pentru a sugera masterului sa introducă o întârziere (delay) în transmiterea de date – acest lucru se mai numește si „clock stretching").
Spre deosebire de alte metode de comunicație serială, magistrala I2C este de tip "open drain", ceea ce înseamnă ca pot trage o anumita linie de semnal în 0 logic dar nu o pot conduce spre 1 logic. Așadar, se elimina problema de „bus contention", unde un dispozitiv încearcă să tragă una dintre linii in starea „high" în timp ce altul o aduce în „low", eliminând astfel posibilitatea de a distruge componentele. Fiecare linie de semnal are un pull-up rezistor pe ea, pentru a putea restaura semnalul pe „high", când nici un alt dispozitiv nu cere „low".



Protocolul de transmisie I2C.

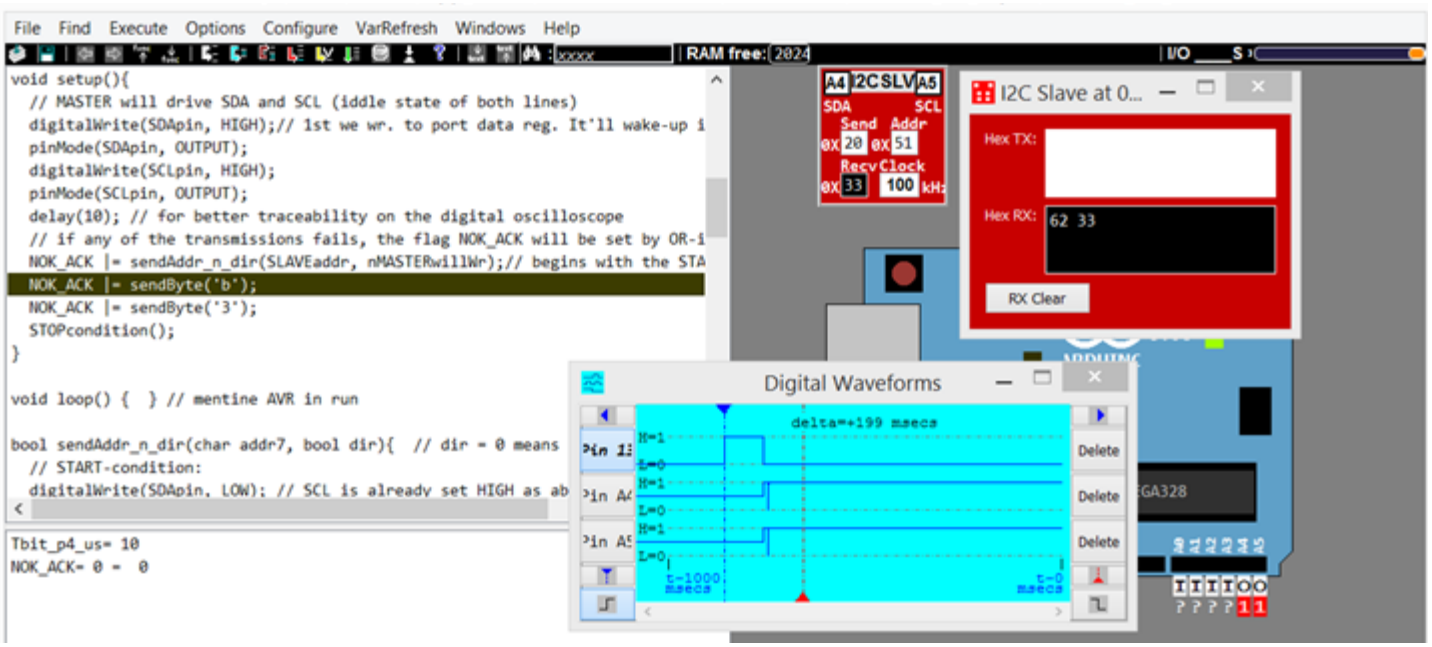
Există 2 tipuri de cadre (frames): cadre de adresă, unde masterul indică slave-ul la care va fi trimis mesajul și unul sau mai multe cadre de date care conțin mesaje pe 8 biți pasate de la master la slave sau viceversa. Datele sunt puse pe linia SDA după ce linia SCL ajunge la nivel low și sunt eșantionate când linia SCL ajunge în HIGH. Timpul între nivelul de ceas și operațiile de citire/scriere este definit de dispozitivele conectate pe magistrală şi poate varia de la cip la cip.
Condiția de start (Start Condition)
Pentru a inița cadrul de adresă, dispozitivul master lasă SCL în high și trage SDA în low. Acest lucru pregătește toate dispozitivele slave întrucât o transmisie este pe cale să înceapă. Dacă două dispozitive master doresc să își asume busul la un moment dat, dispozitivul care trage la nivel low SDA primul câștigă arbitrajul și implicit controlul busului.
Cadrul de adresă (Address Frame) : Cadrul de adresă este întotdeauna primul în noua comunicație. Mai întâi, se trimit sincron biții adresei, primul bit fiind cel mai semnificativ (MSb), urmat de un semnal de R/W pe biți, indicând dacă aceasta este o operație de citire (1) sau de scriere (0). Bitul 8 al cadrului este bitul NACK / ACK. Acesta este cazul pentru toate cadrele (date sau adresă). După ce primii 8 biți ai cadrului sunt trimiși, dispozitivului receptor îi este dat controlul asupra SDA. ***Dacă dispozitivul de recepție nu trage linia SDA în 0 logic înainte de cel de-al 7-lea puls de ceas,*** se poate deduce că dispozitivul receptor fie nu a primit datele, fie nu a știut cum să interpreteze mesajul.
Cadrele de date (Data Frames): După ce cadrul de adresă a fost trimis (pe 8 biti), datele pot începe să fie transmise (pe 8 biți + 1 bit). Masterul va continua să genereze impulsuri de ceas, la un interval regulat, iar datele vor fi plasate pe SDA, fie de master, fie de slave, în funcție de starea biților R/W (care indică dacă o operație este citire sau scriere). Numărul de cadre de date este arbitrar.

Condiția de oprire (Stop condition)

De îndată ce toate cadrele au fost trimise, masterul va genera o condiție de stop. ***Condițiile de stop sunt definite de tranziții low-> high (0-> 1) pe SDA***, ***după o tranziție 0->1 pe SCL cu SCL rămânând pe high***. În timpul operațiilor de scriere, valoarea din SDA nu ar trebui să se schimbe când SCL e high pentru a evita condițiile de stop false.

Practic, exceptând primitivele START și STOP, comunicatia se desfasoară întotdeauna pe 7 sau 8 biți plus un bit de acknowledge. Dacă masterul trimite un byte, slave-ul răspunde imediat cu un bit care poate fi ACK sau NoACK. Când slave-ul trimite un octet (byte), masterul trebuie să îi răspundă imediat cu un bit care poate fi ACK sau NoACK (atenție, masterul/microcontrolerul trebuie să aibă pregătit (setat intern) bitul de ACK sau NoACK înainte să preia byte-ul de la slave).
Comunicatia se desfășoară bidirecțional între master și slave. Initial, master-ul trimite date, urmând ca slave-ul să trimită ulterior date. Practic, slave-ul nu trimite niciodată date fără să i se ceară.
Semnalul de SCL este generat de master. Pentru slave, acest semnal este necesar pentru sincronizare, biții care intră și ies prin SDA sunt sincronizati cu SCL. Viteza cu care se mișcă SCL-ul este determinată de modul în care este setat microcontrolerul.
Totul începe când masterul trimite o condiție de START. După această primitivă, masterul trimite un byte care contine atât headerul tipic IIC, cât și adresa slave-ului cu care vrea să comunice. Adresa unui slave pe o magistrală este unică și este setată legând la masă sau la VCC pinii dedicați ai slave-ului. Acest prim byte trimis de master conține pe ultima pozitie (pe LSb) bitul de R/W. **Dacă bitul este 0**, atunci următorii octeți vor fi trimiși de **master** (sensul este de la master la slave). **Dacă R/W este 1**, atunci, următorii octeți (toți până la apariția unui STOP), vor veni de la **slave**. Odată setată direcția, ea nu se mai schimbă până la apariția unei condiții de STOP.

Aplicatie: Se va implementa în simulator următoarea schemă:



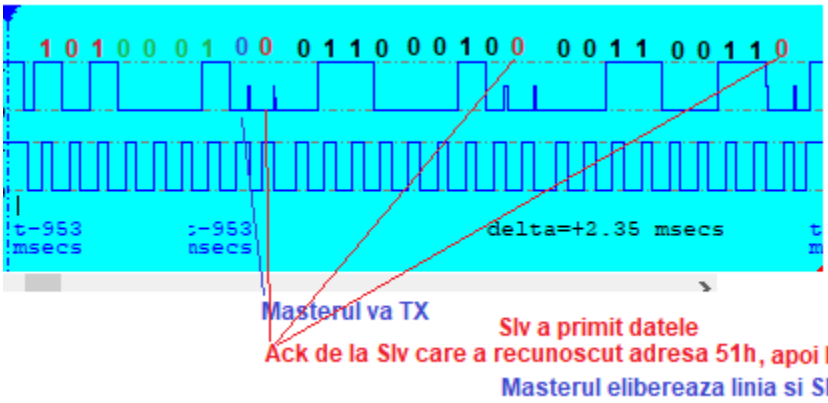
Bitul de R/W =0 (M->S) =1 (M<-S) , in unele foi de catalog se cons ca fiind al 8-lea bit si atunci : daca adresa = para avem M->S, daca e impara avem M<-S; sunt 7 biti de adresa (avem 128 dispozitive slave maxim)
f.u. =1/2
Fr cazator pe SDA cand SCL=H = “start”, Fr urcator pe SDA cand SCL=H = “stop”

/* As an i2c MASTER, send two bytes of data, e.g. 'b' and '5' to the i2c Slave
(the red I2CSLV on the UnoArduSim lab-desk, wired to A4(SDA) and A5(SCL), configured to i2c address 51hex and TX speed of 25kb/s)
Both the START- and the STOP-condition will last 3us.
NOTE: 1. When the last transmitted databit's SCL=H palier ends, MASTER should release SDA (switch sense to input) immediately before SCL falling edge because SLAVE is hunting this edge to pull down SDA as an ACK
2. Checking SCL=L by slave receiver before taking SCL H is not implemented
*/

```
#define SDApin A4
#define SCLpin A5
#define TXspeed 25000                    // 25kb/s
//? do we need to set this value into the red I2CSLV's textbox as well.
// no, the default 100kHz means its own internal logic's attack-clock.
```

```
#define SLAVEaddr 0x51                  // ! do not forget to set this with the mouse to I2CSLV as well
#define nMASTERwillWr 0     // if  nMASTERwillWr is defined as value 1, MASTER will read: M<-S

bool checkACK();  // compiler wants this function's prototype here. For the others, not.

int Tbit_per4_us = (1000000/TXspeed)>>2;        // global variable, T bit is dividing to 4, to get _|--|_ (LHHL)
bool NOK_ACK = 0;                               // 0 means OK,

void setup(){
        // MASTER will drive SDA and SCL (iddle state of both lines)
        digitalWrite(SDApin, HIGH);   // first, we write to port data register. It will wake-up in HIGH
        pinMode(SDApin, OUTPUT);
        digitalWrite(SCLpin, HIGH);
        pinMode(SCLpin, OUTPUT;
        delay(10);                              // for better traceability on the digital oscilloscope

        // if any of the transmissions fails, the flag NOK_ACK will be set by OR-ing with the returned.value
        // if any of the transmission fails, NOK_ACK will be set -> to be observed in the variable area to be 0 !!!
        NOK_ACK |= sendAddr_n_dir(SLAVEaddr, nMASTERwillWr);        // begins with the START condition
        NOK_ACK |= sendByte('b');
        NOK_ACK |= sendByte('3');
        STOPcondition();
}
void loop() {         }
bool sendAddr_n_dir(char addr7, bool dir){          // dir = 0 means "MASTER will write", dir=1 means "SLAVE has to write"
        // START-condition:
        digitalWrite(SDApin, LOW);   // SCL is already set HIGH as above...
        delayMicroseconds(3);
        digitalWrite(SCLpin, LOW);
        delayMicroseconds(3);                   // this may be redundant, but it does not harm

        // Now we send out the SLAVE address to the i2c bus (The i2c address is on 7bits.)
        send_bits(addr7, 7);                    // value to be sent and it's number of bits

        // now MASTER sends the direction-bit, 0 means MASTER will write, 1 means it will read
        digitalWrite(SDApin, dir);
        clockPulse(8);                          //  value 8 orders SDA sense reverting, to allow ACK checking
        return checkACK();
}
void send_bits(char b, char n_bits){
        for(int i=1; i <= n_bits; i++){    // MSb, ... , b0
                digitalWrite(SDApin, b & 1<<(n_bits-i)); // operators precedence: '<<' = 4, ...'&'= 7
                clockPulse(i);                                  // last call is with i=7 for address and i=8 for data
                // if the bits are an address, we have to send also the 8-th bit (direction),
                // so, the sense of SDA should not be reverted yet, but only after sending the LSb.
        }
}
void clockPulse(char bit_nro){                  // _|--|_ (LHHL)
        delayMicroseconds(Tbit_per4_us);        // SCL is already in LOW
        digitalWrite(SCLpin, HIGH);             // taking SCL H
        delayMicroseconds(Tbit_per4_us << 1);   // wait 2 periods
        if(bit_nro==8)                          // we switch the SDA-sense immediately BEFORE taking SCL LOW
                pinMode(SDApin, INPUT);         // now i2c SLAVE will pull down SDA if RX is OK. (ACK)
        digitalWrite(SCLpin, LOW);              //falling edge of SCL
        delayMicroseconds(Tbit_per4_us);
}                                               // the address is on 7 bits because the 8-th bit is for dir

bool checkACK(){
        // we already made SDA sense input before the last falling edge of SCL and sustained SCL = LOW for Tbit/4
        delayMicroseconds(Tbit_per4_us);        // we do a same-shaped SCL-pulse to get the ACK bit
```

```
        digitalWrite(SCLpin, HIGH);
        delayMicroseconds(Tbit_per4_us);          // SLAVE has to pull down if OK (ACK)
        bool ACKbit = digitalRead(SDApin);   // sampling SDA at the middle of Tbit
        delayMicroseconds(Tbit_per4_us);

        digitalWrite(SCLpin, LOW);          // at falling edge of SCL, i2c SLAVE will release SDA to High Impedance
        digitalWrite(SDApin, LOW);          // prepared for STOP condition, in case this was the last transmission
        pinMode(SDApin, OUTPUT);            // MASTER takes back SDA
        delayMicroseconds(Tbit_per4_us);
        return ACKbit;
}
bool sendByte(char x){
        send_bits(x, 8);// 8 databits, 8 means "this is not an addr.
        return checkACK();
}
void STOPcondition(){
        digitalWrite(SCLpin, HIGH)    ; SCL=H
        delayMicroseconds(3);
        digitalWrite(SDApin, HIGH)    ; SDA=H
}
```



START: fr caz pe date cand clk e sus
STOP: fr urc pe date cand clk e sus

Recepția a 2 caractere prin I2C:

```
/* As an i2c MASTER, receive two bytes of data, e.g. 'a' and 'b' from
i2c Slave(the red I2CSLV on the UnoArduSim lab-desk, wired to A4(SDA) and A5(SCL), configured to i2c addr. 51hex and TX speed of 25kb/s )
Both the START- and the STOP-condition will last 3us.
NOTE:
1. When the last TX-ed databit's SCL=H palier ends
MASTER should release SDA (switch sense to input) immediately before SCL falling edge
because SLAVE is hunting this edge to pull down SDA as an ACK
2. Checking SCL=L by slave receiver before taking SCL H is not implemented
*/

#define SDApin A4
#define SCLpin A5
#define TXspeed 25000 // 25kb/s
//? do we need to set this value into the red I2CSLV's textbox as well.
// no, the default 100kHz means its own internal logic's attack-clock.
#define SLAVEaddr 0x51 // ! do not forget to set this with the mouse to I2CSLV as well
#define nMASTERwillWr 1 // if nMASTERwillWr is defined as value 1, MASTER will read

int Tbit_p4_us = (1000000/TXspeed)>>2;          // GLOB-var, many fcn's will use-it
bool NOK_ACK = 0;           // 0 means OK,
//it's GLOB-var to be visible in UnoArduSim's var's pane. It will be OR-ed with every ACK-bit...

void checkACK();            // compiler wants this fcn.'s prototype here. For the others not.
char rcvByte();             // prototype

void setup(){
char c;
```

```
Serial.begin(9600);               // we'll print into SerialMonitor what I2CSLV sends
// idle line condition. May be tw0 pull-up rezistors would be OK, because upon RESET all GPIO pins are inputs
digitalWrite(SDApin, HIGH);   // 1st we wr. to port data reg.s They'll wake-up in HIGH-state
digitalWrite(SCLpin, HIGH);
pinMode(SDApin, OUTPUT);
pinMode(SCLpin, OUTPUT);
delay(10);                        // for better traceability on the digital oscilloscope
sendAddr_n_dir(SLAVEaddr, nMASTERwillWr);   // begins with the START condition
/*do{
        Serial.print(c = rcvByte());            // atribuirea insasi returneaza o valoare, pe cea atribuita
    } while(c != 0);                // Our convention: terminator-val., to allow arbirtrary nbr of bytes sentby I2CSLV
*/
Serial.print(c = rcvByte());
Serial.print(c = rcvByte());
STOPcondition();
}

void loop() { }                         // mentine AVR in run

void sendAddr_n_dir(char addr7, bool dir){      // dir = 0 means "MASTER will write", dir=1 means "SLAVE has to write"
// START-condition:
digitalWrite(SDApin, LOW);                      // SCL and SDA are set HIGH in setup()...
delayMicroseconds(3);
digitalWrite(SCLpin, LOW);
delayMicroseconds(3);                           // this may be redundant, but it does not harm
// Now we send out the SLAVE addr. to the i2c bus (The i2c addr. is on 7bits.)
send_bits(addr7, 7);                            // value to be sent by MASTER and it's nbr.of bits
// now MASTER sends the direction-bit, 0 means MASTER will write, 1...read
digitalWrite(SDApin, dir);                      // direction-bit, 0=Master will Write, sent by MASTER as the 8-th addr.-bit
clockPulse(8);                                  // value 8 orders SDA sense reverting(right before SCL H2L), to allow ACK checking
checkACK();
  if(dir == 1)
        pinMode(SDApin, INPUT);
}

void send_bits(char b, char n_bits){
for(int i=1; i <= n_bits; i++){                 // MSB, ... b0
        digitalWrite(SDApin, b & 1<<(n_bits-i));        // oper. precedence: '<<' = 4, ...'&'= 7
        clockPulse(i);                                  // last call is with i=7 for addr. and 8 for data
        // if the bits are an addr. we have to send also the 8-th bit, direction,
        // so, the sense of SDA should not be reverted yet after sending the LSB.
        }
}

void clockPulse(char bit_nro){ // _|--|_ (LHHL)
delayMicroseconds(Tbit_p4_us);
digitalWrite(SCLpin, HIGH);             // taking SCL H
delayMicroseconds(Tbit_p4_us << 1);
if(bit_nro==8)                          // we switch the SDA-sense immediately BEFORE taking SCL LOW
pinMode(SDApin, INPUT);                 // now i2c SLAVE will pull down SDA if RX is OK. (ACK)
digitalWrite(SCLpin, LOW);              //falling edge of SCL
delayMicroseconds(Tbit_p4_us);
}                                       // the address is on 7 bits because the 8-th bit is for dir. Some are naming-it "the 8-th addr.-bit".

void checkACK(){                        // Master checks for ACK-bit, pulled down by slave upon any received byte(addr. or data)
// we already made SDA-sense input before the last falling edge of SCL, in clocKPulse()
// and sustained SCL = LOW for Tbit/4
delayMicroseconds(Tbit_p4_us);          // we do a same-shaped SCL-puls to get the ACK bit
digitalWrite(SCLpin, HIGH);
delayMicroseconds(Tbit_p4_us);          // SLAVE has to pull down if OK (ACK)
NOK_ACK |= digitalRead(SDApin);         // sampling SDA @ the middle of Tbit
delayMicroseconds(Tbit_p4_us);
```

```
digitalWrite(SCLpin, LOW);              // at falling edge of SCL i2c SLAVE will release SDA to HZ
digitalWrite(SDApin, LOW);              // prepared for STOP cond, in case this was the last TX
pinMode(SDApin, OUTPUT);                // MASTER takes back SDA
delayMicroseconds(Tbit_p4_us);
}

void sendByte(char x){
send_bits(x, 8);                        // 8 databits, 8 means "this is not an addr.
checkACK();
}

void STOPcondition(){
digitalWrite(SCLpin, HIGH) ;
delayMicroseconds(3);
digitalWrite(SDApin, HIGH) ;
}

char rcvByte(){                 // SCL does: _|-rd_SDA-|_ (LHHL) for every bit
char rxByte = 0, i;
pinMode(SDApin, INPUT);     // SDA sens intrare
for (i=7;i>=0;i--){             // pack the coming bits to a char
        delayMicroseconds(Tbit_p4_us);
        digitalWrite(SCLpin, HIGH);
        delayMicroseconds(Tbit_p4_us);                  // wait till the mid. of the bit-dur.
        bitWrite(rxByte, i, digitalRead(SDApin)&1);     // !!!!!!!!!!!! my be &1 is redundant
        delayMicroseconds(Tbit_p4_us);
        if(i==0){                                       // after last bit is read
                digitalWrite(SDApin, LOW);// avoid hazard
                digitalWrite(SCLpin, LOW); // falling edge of SCL. Now slave starts hunting for SCL LHL to read the ACK bit
                pinMode(SDApin, OUTPUT);// now i2c MASTER now pulls down SDA for one bit-period, as ACK to slave
                clockPulse(8);// param.=8 will switch SDA (back) to input, rihgt before SCL falling edge
                }
        else{
                digitalWrite(SCLpin, LOW);
                delayMicroseconds(Tbit_p4_us);// sustain to cover the whole bit-duration
                }
        }
/* Master sends ACK. When the TX-ed ACK=0 databit's SCL=H palier ends
MASTER should release SDA (switch sense to input) immediately *before* SCL falling edge
because SLAVE is hunting this edge to take-over driving SDA with the MSB of the next byte*/
return rxByte;
}
```