RENESAS

Application Note

Implementing a Software I²C Master with the K-Line Microcontroller

Document No. U17206EE1V1AN00
Date Published September 2005

Printed in Germany

Table of Contents

List of Figures

Chapter 1 Introduction

The I²C bus (I²C = IIC = Inter-Integrated Circuit) is a bi-directional two wire clock synchronous bus operating in a master / slave relationship. It consists of a data line (SDA) and a clock line (SCL). The master device always generates the clock. Maximum throughput is 100 Kbit/s for standard devices, 400 Kbit/s for fast mode and in 1998 version 2.0 was introduced, operating at up to 3.4 Mbit/s.

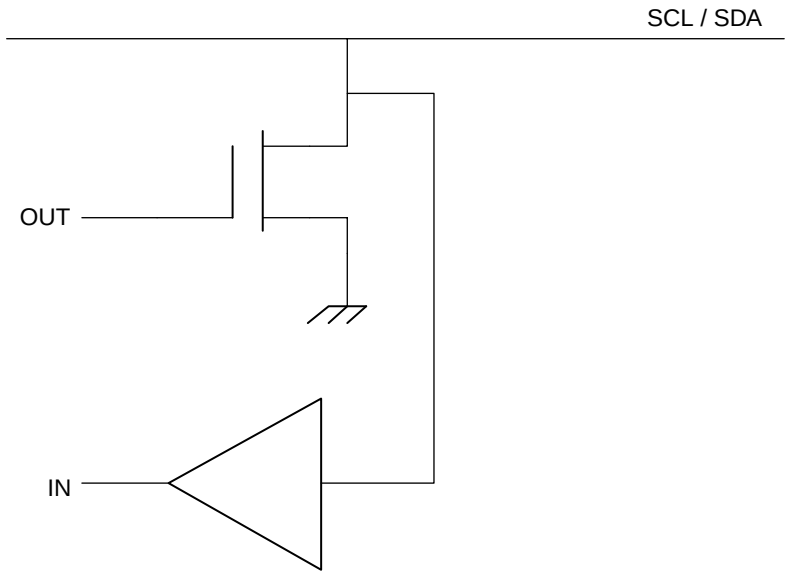
Some NEC microcontrollers are available with I²C hardware; for those parts without an I²C port a collection of software routines are presented here that can be used to create an I²C master with any NEC MCU. Only two bi-directional port pins are needed.

I²C is a registered trademark of Philips Corporation.

Chapter 2 Hardware Arrangement

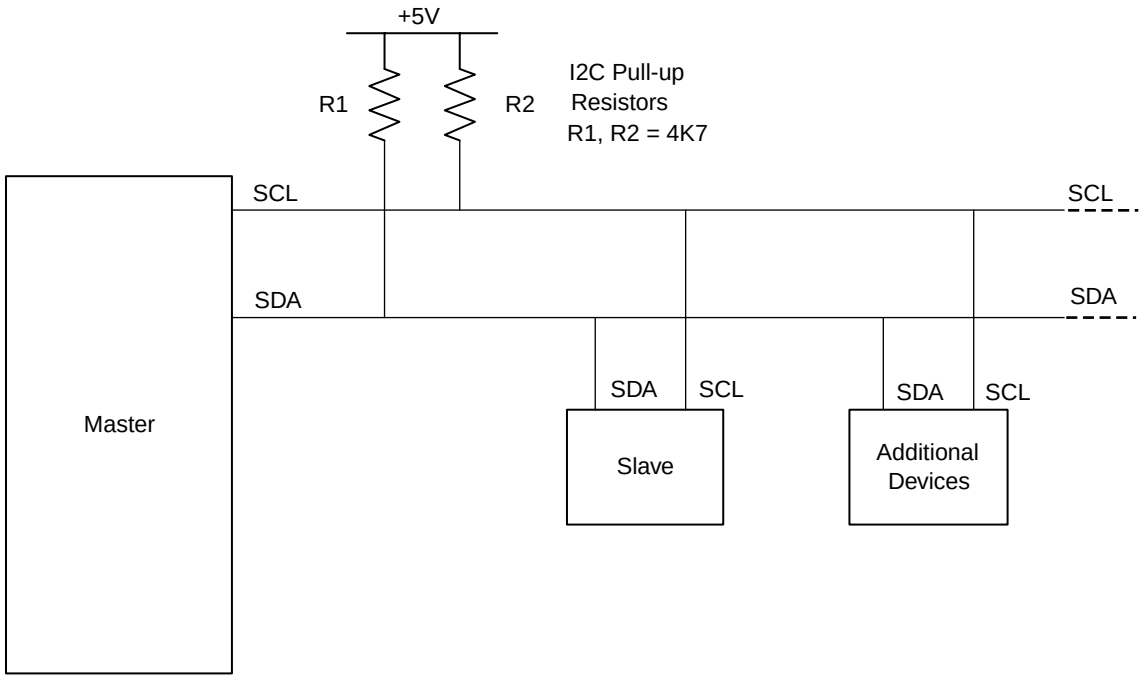
The I²C bus operates on a wired-AND principle, allowing cascading of any number of devices on a single bus. (In practice the number of devices is limited by the number of device addresses available).
Figure 2-1 shows internal I²C bus interface circuitry in a simplified form:

Figure 2-1: Simplified I²C Driver Architecture



It is clear from the above that some form of pull-ups are required for the output open drain transistor to function correctly.
Figure 2-2 shows how I²C devices are typically interconnected, using a pull-up resistor for SCL (Serial Clock) and SDA (Serial Data). The exact value of these resistors depends on supply voltage, bus capacitance and the number of devices on the bus. The maximum bus capacitance permitted is 400 pF. Active pull-ups can be used in difficult conditions (e.g. where long PCB tracks give rise to high capacitance). The value of 4K7 shown below works satisfactorily for most small systems.
For further information, refer to the Philips publication “The I²C Bus Specification Version 2.1”, January 2000.
The I/O port pins of the NEC MCU may be switched from input to output as required to emulate the arrangement above.

Figure 2-2: I²C Bus Typical Interconnection

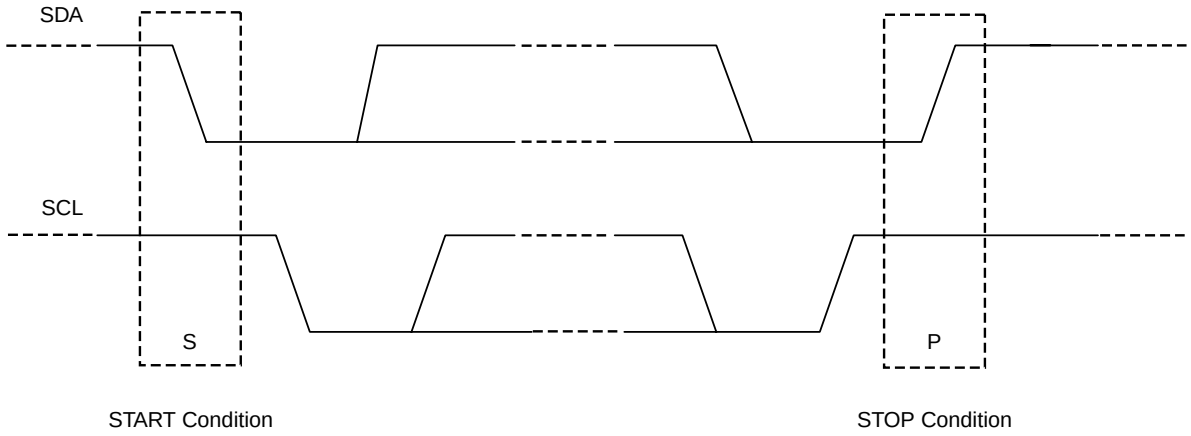


Chapter 3 I²C Bus Data Format

3.1 START and STOP Conditions

All data transfers are initiated and terminated with a unique bus condition. A HIGH to LOW transition on the SDA line while SCL is HIGH is considered a START (S) condition while a LOW to HIGH transition on SDA with SCL HIGH is a STOP (P) condition. See Figure 3-1 below.

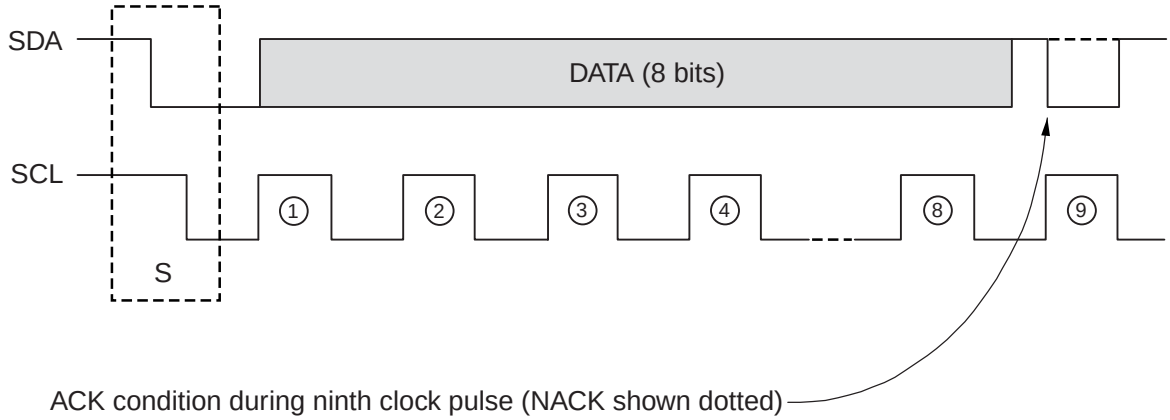
Figure 3-1: I²C START and STOP Conditions



3.2 Acknowledgement

All I²C byte transfers must end with an acknowledgement (ACK) from the receiving device. This is done by the master releasing the SDA line (i.e. switching it to an input) and transmitting a *ninth* clock pulse. During this ninth clock period the receiving slave device must keep SDA pulled to a stable LOW (ACK). If the receiver is unable to service the transfer it may leave SDA HIGH, this is called a not-acknowledge (NACK) condition. The transmitter can then act upon the NACK, either ending the transfer with a STOP condition or attempting the action again with a repeated START. (Repeated START is the term given to a START condition that appears in the middle of a transfer. It is also used during an I²C read operation, see later.)

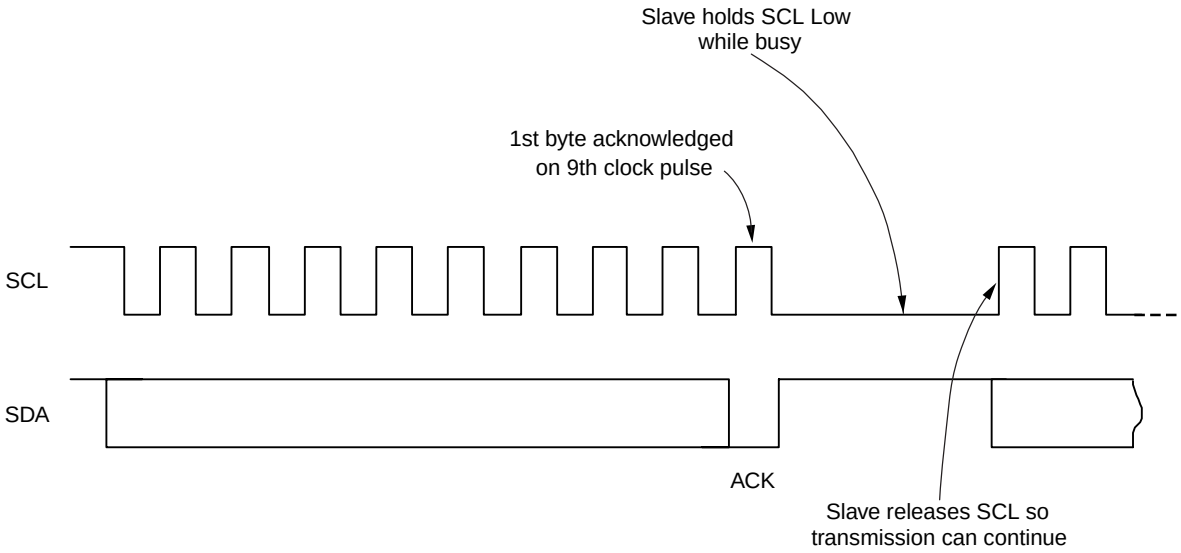
Figure 3-2: I²C ACK and NACK Conditions



3.3 Clock Stretching

Sometimes the master device will need to access a slave device that cannot respond immediately to the read or write request. This may be because the slave is busy or it is just an inherently slower device. A clock stretching mechanism is available for this situation: a slave is permitted to hold the SCL line low while it is busy, and then release it so the master can continue the transmission. For example, in Figure 3-3, the first byte could be the address of a byte to be read from a slave device. This is acknowledged by the slave, which may then take a relatively long time to retrieve the data at this address, so it holds SCL low while it does this. The master must poll the SCL line to detect its release; then clocking of SCL may continue.

Figure 3-3: Clock Stretching



The firmware presented later does not allow for clock stretching as it is, but may easily be modified to do so if the application requires it. It is important to note the macros to control SCL will need to be modified so that rather than switching SCL high or low they will switch it from input to output (with the port value always 0) to avoid contention with a slave that is trying to hold the line low, and also to facilitate polling of the SCL line. A timeout timer may also be added to ensure the master does not wait indefinitely should there be a fault in the slave device.

3.4 Data Transfer Format

Data is always transmitted in 8 bit bytes, MSB (Most Significant Byte) first. The first byte to be sent has the slave address in the seven MSB's, followed by the read / write bit, which is set to read and clear to write. The slave address is defined by the device manufacturer and is unique to a particular device, thus allowing many devices of different manufacturer to co-exist on the same bus. Some parts, especially memories, have an address *range* specified at manufacture but leave several pins free for the user to connect to define the exact address within the range, so for example four 2K EEPROMS may be connected to the bus with no additional hardware; their base address defined at manufacture and their individual addresses defined by the logic levels on their address pins. Ten bit addressing is possible with I²C but will not be covered by this application note.

Figure 3-4 shows the data format for a master reading and writing a slave device, and for a combination transfer.

When the master wants to write to the slave, the following happens:

1. Master sends START condition.
2. Master sends slave address with R/W bit CLEAR.
3. Slave issues ACK on ninth clock pulse.
4. Master sends first data byte.
5. Slave issues ACK.
6. Steps 4 and 5 are repeated for all data bytes.
7. Transfer ends with either a STOP condition after the last data byte / ACK pair if the master has no more data to send, or if the slave does not wish to take more data it can inform the master by issuing a NACK after the last data byte. The master then issues a STOP condition as usual to terminate the transfer.

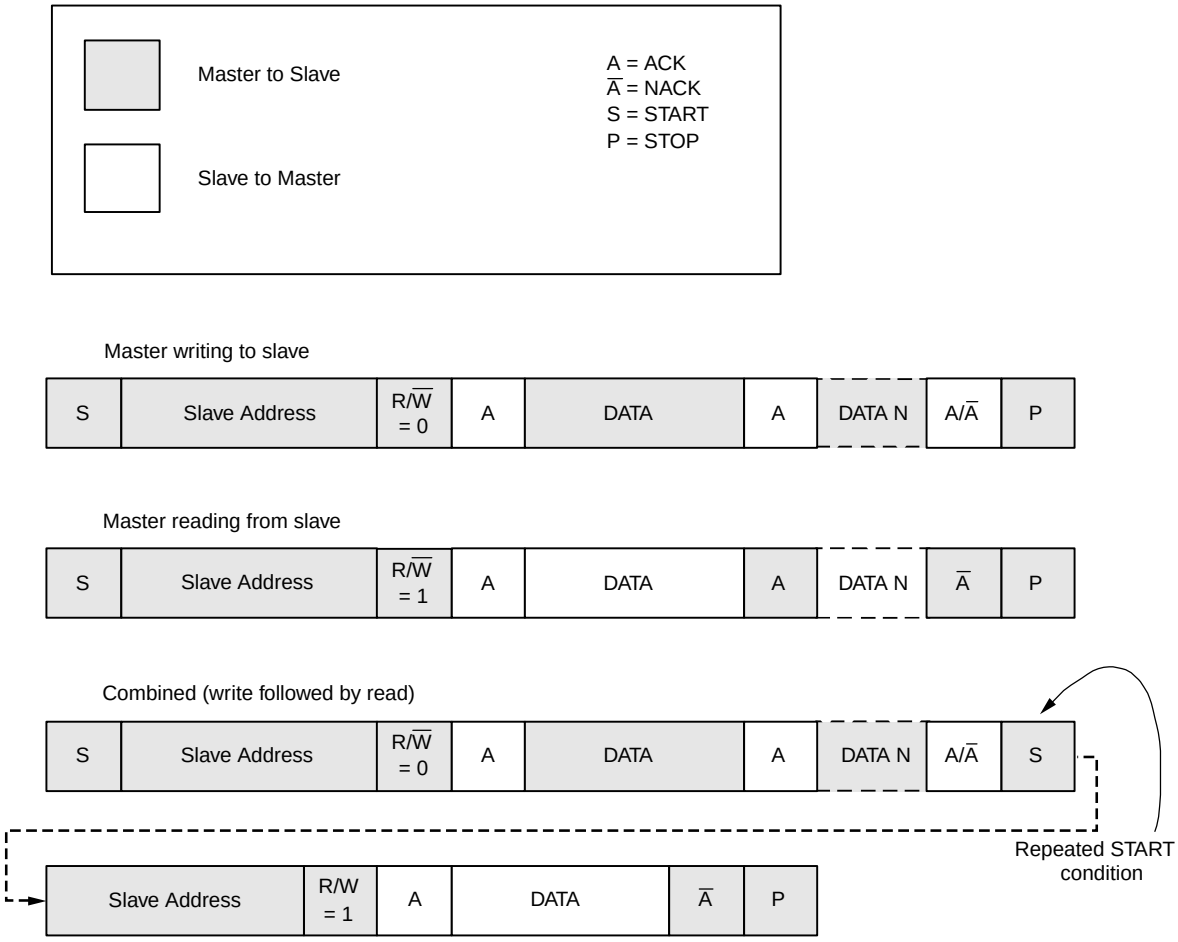
When the master wants to read from the slave, the following happens:

1. Master sends START condition.
2. Master sends slave address with R/W bit SET.
3. Slave issues ACK on ninth clock pulse.
4. Master reads first data byte.
5. Master issues ACK.
6. Steps 4 and 5 are repeated for all data bytes except the last.
7. After reading the last byte, the master issues a NACK to inform the slave there is no more data to be transferred.
8. The master issues a STOP condition.

For the combined transfer (example shown here is write to slave followed by read from slave) the following happens:

1. The slave address and bytes to be *written* are sent in the same manner as for a straightforward write as described above.
2. A *repeated START* is issued by the master followed by the slave address, this time with the R/W bit SET (read).
3. Data is read from the slave in the usual way.
4. The master issues a NACK to indicate to the slave it no longer wishes to read data.
5. The transfer ends with a STOP condition.

Figure 3-4: Data Transfer Format



3.5 EEPROM Read and Write

The firmware listed in this application note can perform, in addition to basic read byte / write byte operations, reads and writes to a 24CXX EEPROM (Electrically Erasable Read Only Memory). More details of the EEPROM can be found in the relevant data sheet, but the format for writing data follows “master writing to slave” in Figure 3-4 above, i.e:

1. Write device address (R/W bit set to WRITE)
2. Write address *within* device
3. Write data byte

To read data, follow “combined” above, i.e:

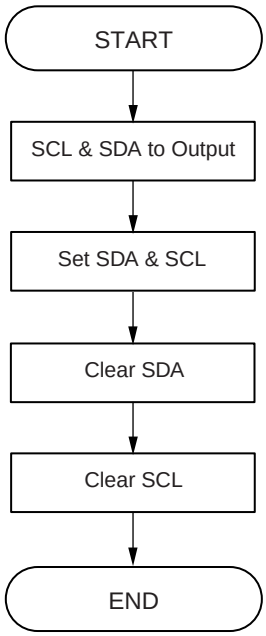
1. Write device address (R/W bit set to WRITE)
2. Write address *within* device
3. Issue repeated START
4. Write device address (R/W bit set to READ)
5. Read data byte

Chapter 4 Firmware – Flow Diagrams and Description

Before the firmware listing is presented, flow diagrams for each function are shown here.

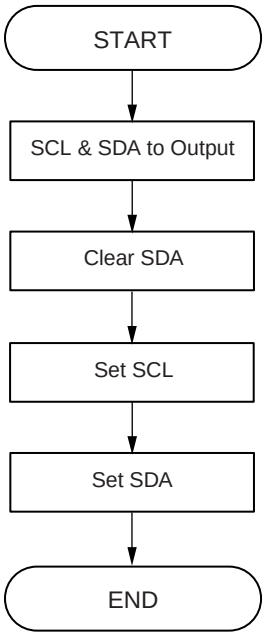
4.1 START Condition

Figure 4-1: START Condition



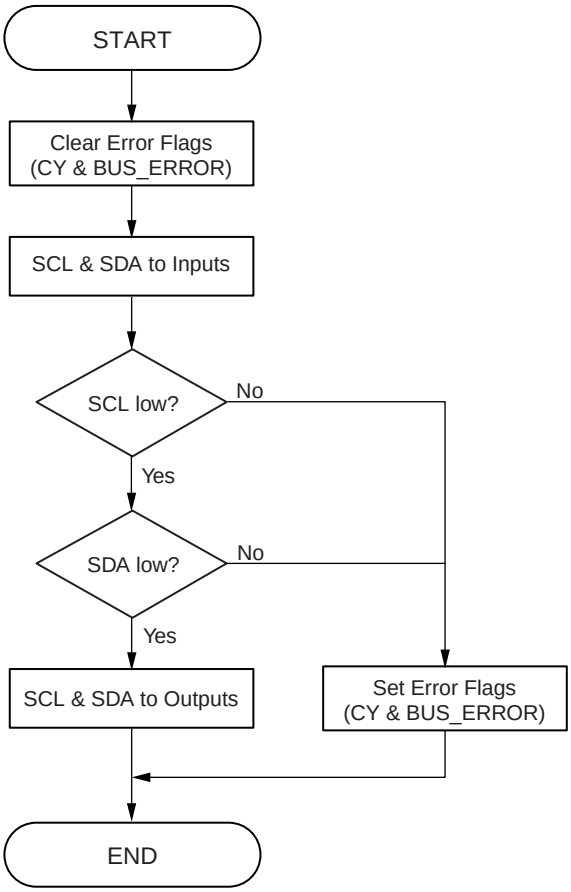
4.2 STOP Condition

Figure 4-2: STOP Condition



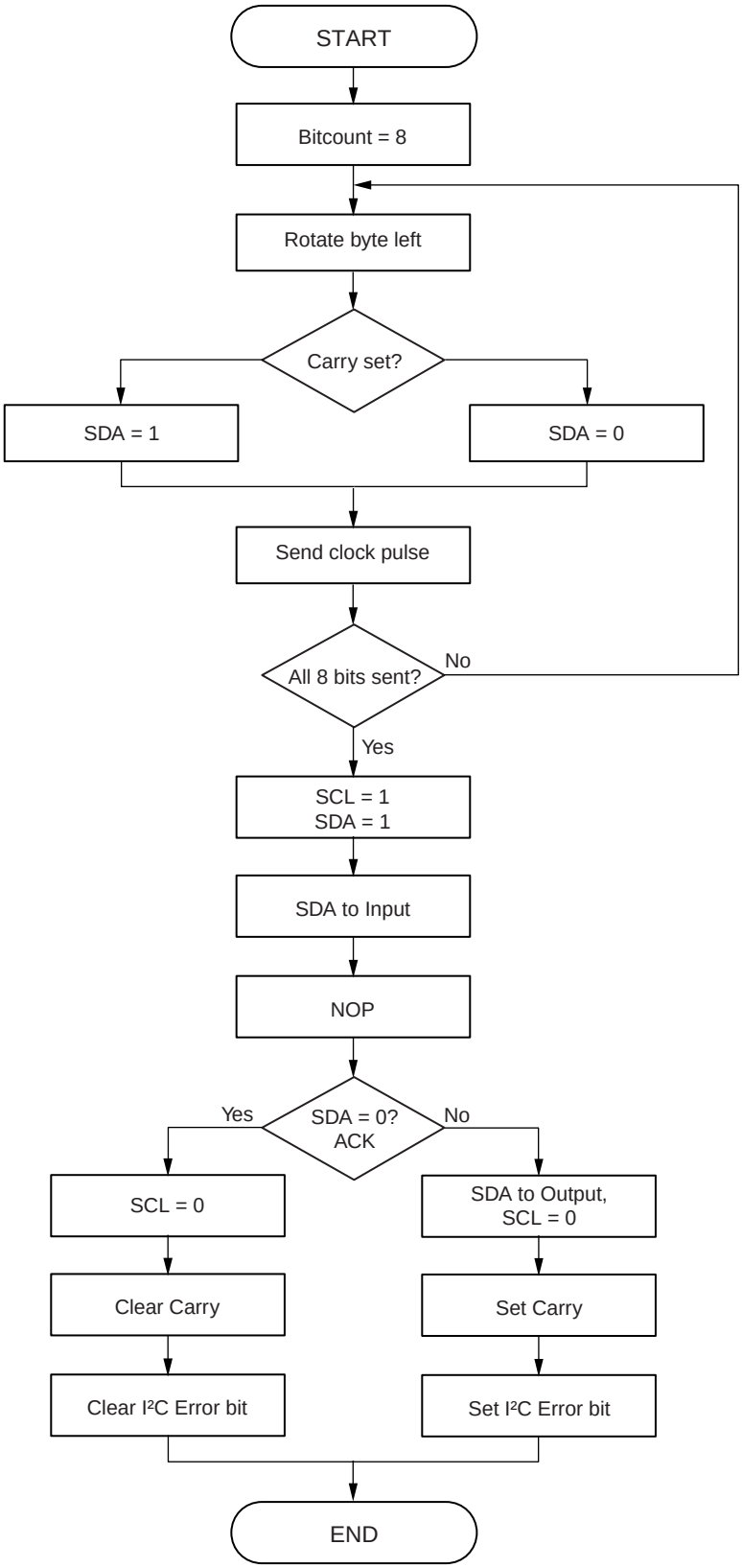
4.3 Bus Check

Figure 4-3: Bus Check



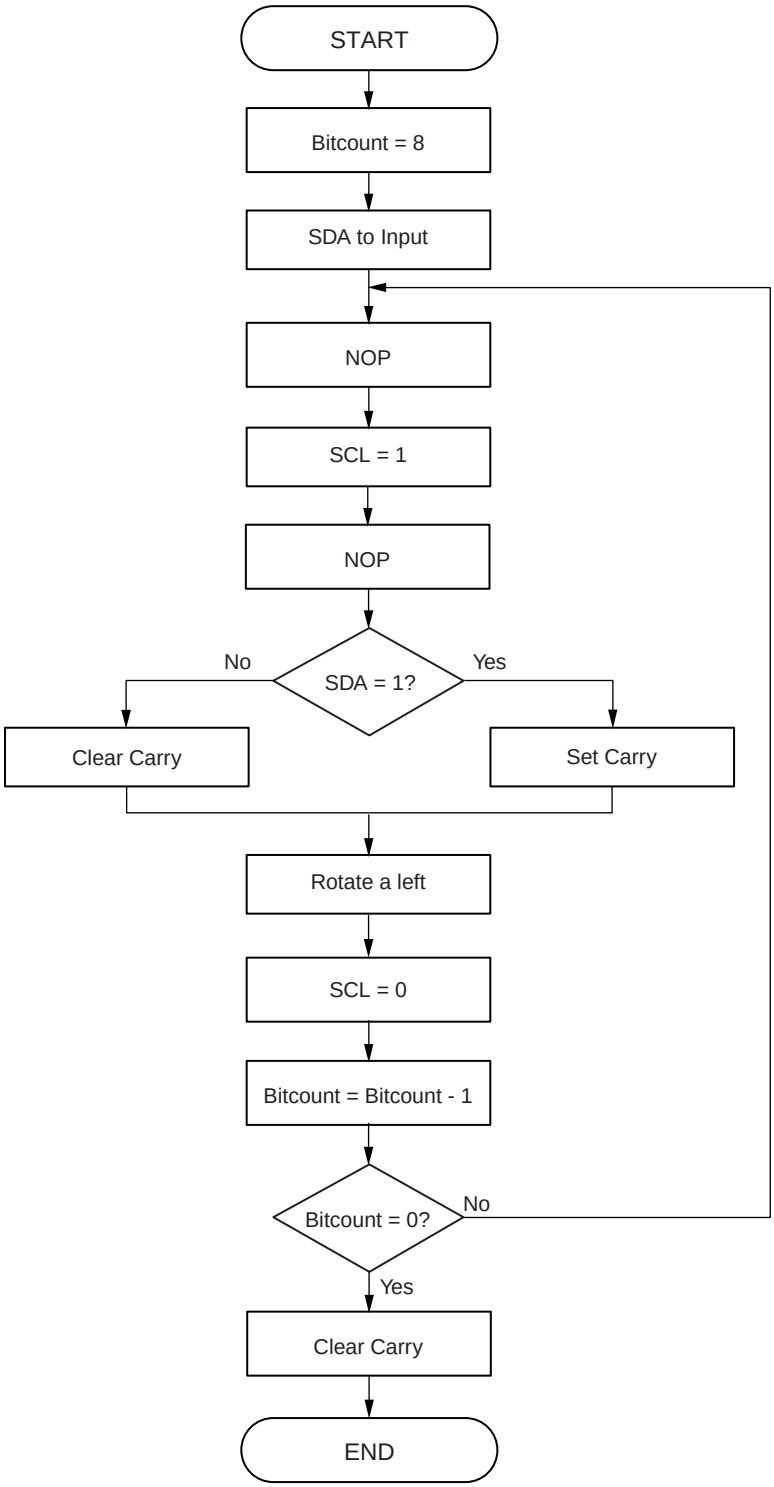
4.4 Send Byte

Figure 4-4: Send Byte



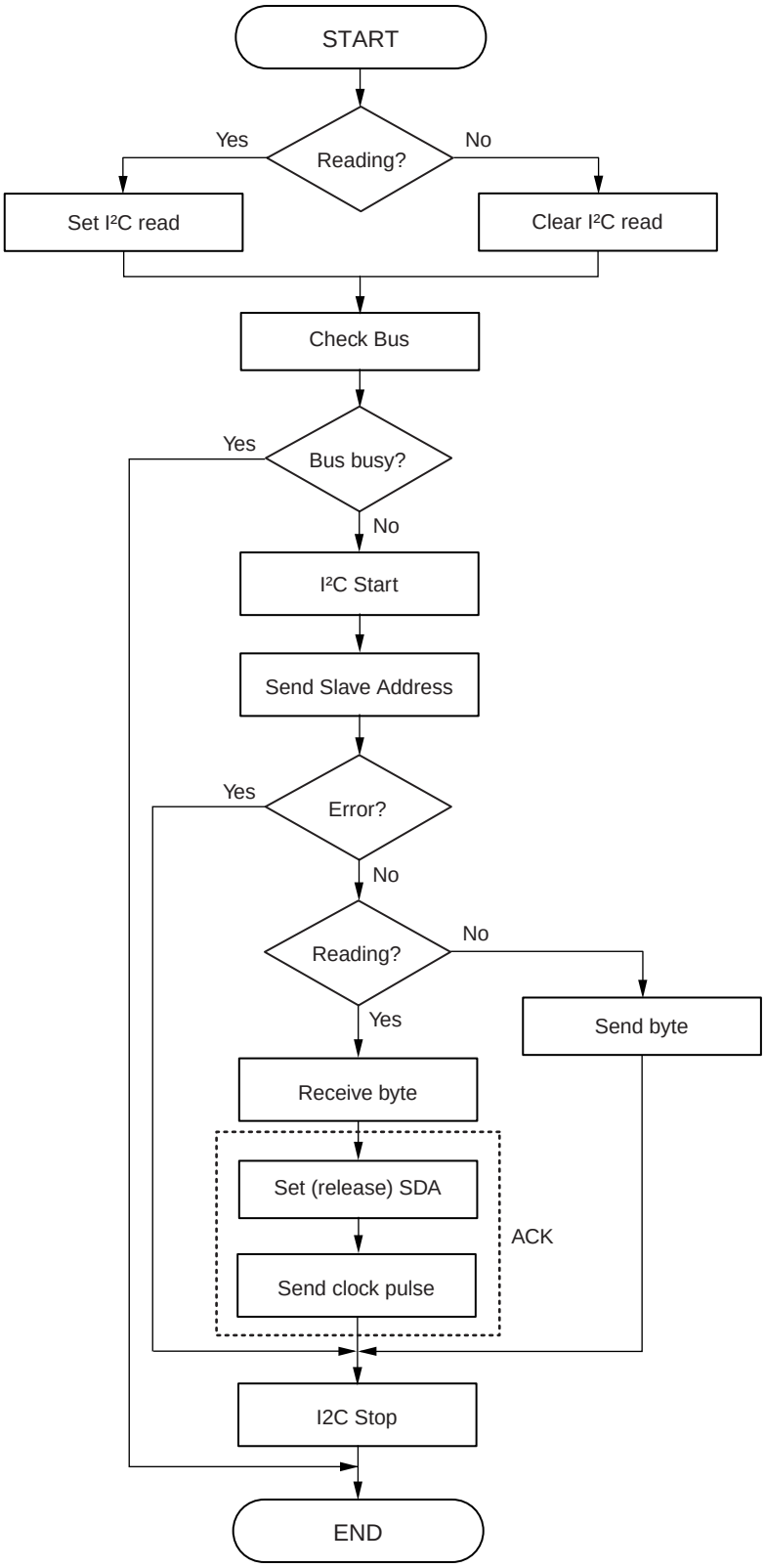
4.5 Receive Byte

Figure 4-5: Receive Byte



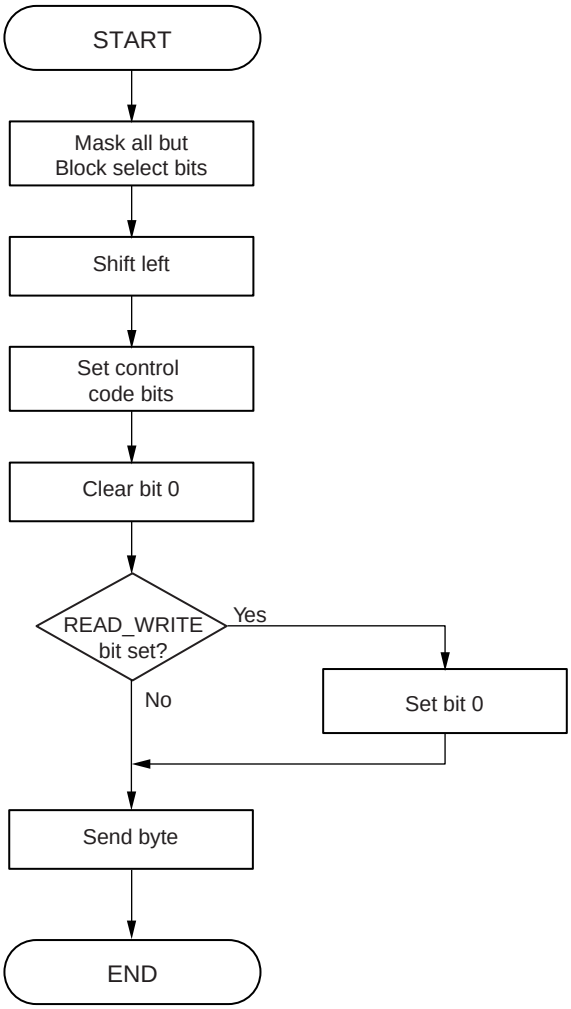
4.6 Get / Put I²C Byte

Figure 4-6: Get / Put I²C Byte



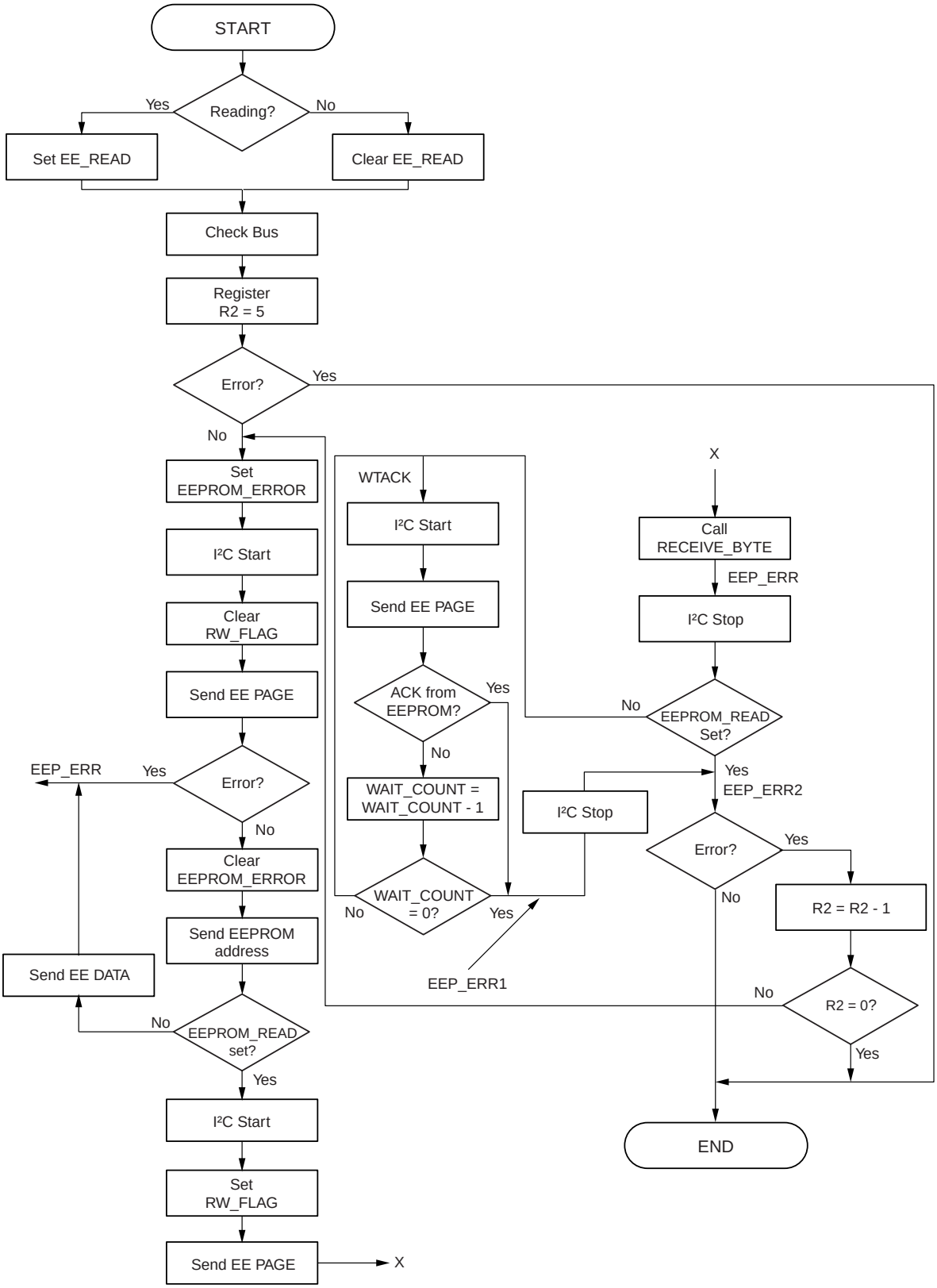
4.7 Send EEPROM Page

Figure 4-7: Send EEPROM Page



4.8 EEPROM Read and Write

Figure 4-8: EEPROM Read and Write



Chapter 5 Firmware – Program Listings

5.1 Main (test) Program

The program was developed with IAR's Embedded Workbench, with i2c_eeprom.c and i2c.msa added to the project under "Options → Files...".
The main program (in C) simply uses the _SEND_EEPROM and _RECV_EEPROM assembly language routines to repeatedly write numbers 0 – 19 to EEPROM locations 0 – 19, and read them back into a buffer as a means of testing and demonstrating the I²C operation. Other subroutines can be called from the users application following the same convention.

```
/*============================================================================
** PROJECT      = I2C_1.prj
** MODULE       = i2c_eeprom.c
** VERSION      = 0.1
** DATE         = 18.03.2001
** LAST CHANGE  = 01.06.2004
** ============================================================================
** Description: Operation as 16-bit timer interrupt
**
** ============================================================================
** Environment:  Device:      uPD78911x
**                            Assembler:            A78000 Version 3.34.2.4
**                            C-Compiler:           ICC78000 Version 3.34.2.4
**                            Linker:               XLINK Version 4.55.9.0
**
** ============================================================================
**      By:            NEC Electronics (Europe) GmbH
**                     Arcadia Strasse 10
**                     D-40472 Duesseldorf
**      and:
**                     NEC Electronics (Europe) GmbH
**                     Cygnus House
**                     Sunrise Parkway
**                     Milton Keynes MK14 6NP
**
** ============================================================================
Changes:
** ============================================================================
*/

/* ============================================================================
** pragma
** ============================================================================
*/
#pragma language = extended

/* ============================================================================
** include
** ============================================================================
*/
#include <in78000.h>
#include "df9116a.h"
#include "i2c.h"

/* ============================================================================
** type definitions (function prototypes)
** ============================================================================
*/
```

```
/* ============================================================================
** variable definitions
** ============================================================================
*/

saddr char count1 = 0, received_data[20];


/* ============================================================================
** variable init
** ============================================================================
*/

void hdwinit (void){

        // port setting
        PM0  = 0xF0;                                    // port 0 = output
        PM1  = 0xFC;                                    // port 1 = output
        PM2  = 0xC0;                                    // port 2 = output
        PM5  = 0xF0;                                    // port 5 = output
        PU0  = 0x00;                                    // no pull up-resistors
        PUB2 = 0x00;

        // clock generator setting
        PCC = 0x00;                             // with speed

}



/* ============================================================================
** main function
** ============================================================================
*/

void main(void){

        hdwinit ();                     // peripheral settings
        _Reset_Bus();


        for(;;){                        // endless loop - main loop

                for (count1 = 0; count1 < 20; count1++){
                        _Send_Eeprom (0, count1, count1);

                }

                for (count1 = 0; count1 < 20; count1++){
                        received_data [count1] = _Recv_Eeprom(0, count1);

                }

        }

}
```

5.2 Header File – i2c.h

```
/*=============================================================================
** PROJECT      = I2C_1.prj
** MODULE       = i2c.h
** VERSION      = 0.1
** DATE         = 20.12.2001
** LAST CHANGE  = 01.06.2004
** ============================================================================
** Description: Header file for the I2C communication
**              needs also i2c.msa file
** ============================================================================
** Environment:  Device:      uPD789xxx
**                            Assembler:            A78000 Version 3.34.2.4
**                            C-Compiler:           ICC78000 Version 3.34.2.4
**                            Linker:               XLINK Version 4.55.9.0
**
** ============================================================================
**      By:            NEC Electronics (Europe) GmbH
**                     Arcadia Strasse 10
**                     D-40472 Duesseldorf
**      and:
**                     NEC Electronics (Europe) GmbH
**                     Cygnus House
**                     Sunrise Parkway
**                     Milton Keynes MK14 6NP
**
** ============================================================================
Changes:
** ============================================================================
*/


extern void _Reset_Bus(void);

extern void _Put_I2C_Byte (unsigned char a,unsigned char d);
extern char _Get_I2C_Byte (unsigned char a);
extern void _Put_I2C_Reg (unsigned char a,unsigned char r,unsigned char d);
extern unsigned char _Get_I2C_Reg (unsigned char a,unsigned char r);
/* the variable names stand for:
                                    a = device-address
                                    r = register-address (address in device)
                                    d = data byte
*/

extern void _Send_Eeprom(unsigned char page,unsigned char a,unsigned char d);
extern unsigned char _Recv_Eeprom(unsigned char page,unsigned char a);
/* the variable names stand for:
                                    address is fixed in the .msa file to 0xA0
                                    page = device page or chip address
                                    a = memory adress
                                    d = data byte
*/



extern unsigned char eeprom_bitreg;

/*
 this variable is declared in the msa file and used for several flags:

    Bit0:  internally used, read flag for i2c communication
    Bit1:  set to 1, if eeprom error occurs after 5 attempts to access the eeprom
    Bit2:  internally used, read/write flag to distinguish between the eeprom read
           and write operations
    Bit3:  set to 1 if the bus is not free
```

```
    Bit4:  set to 1 if i2c communication error occurs
    Bit5:  internally used, read/write flag to distinguish between the i2c read
           and write operations

The error handling has to be done by the C software.
The wait time for the write cycle of the eeprom is done by polling the acknowledge
after sending the device address again (max. 100 times)


*/
```

5.3 Assembly Language Subroutines

```
;NEC Electronics Europe
;General purpose I2C driver routines
;with EEPROM routines


#include <df9116a.h>

 public  _Put_I2C_Reg
 public  _Get_I2C_Reg
 public  _Put_I2C_Byte
 public  _Get_I2C_Byte
 public  _Send_Eeprom
 public  _Recv_Eeprom
 public  _Reset_Bus
 public  eeprom_bitreg

;
;I2C I/O...
;

SDA             equ     P5.1
SCL             equ     P5.0

#define SDAIN           SET1 PM5.1
#define SDAOUT          CLR1 PM5.1
#define SCLIN           SET1 PM5.0
#define SCLOUT          CLR1 PM5.0

;
;flags
;

rw_flag         equ eeprom_bitreg.0
eeprom_error    equ eeprom_bitreg.1
eeprom_read     equ eeprom_bitreg.2
bus_error       equ eeprom_bitreg.3
i2c_error       equ eeprom_bitreg.4
i2c_read        equ eeprom_bitreg.5


wait            equ     100          ;check x times for acknowledge after write


; ============================================================================================

;Macros...

;Set SCL
Set_SCL MACRO
    set1            SCL              ;3 times to guarantee pulse width
    set1            SCL
    set1            SCL
  ENDM

;Clear SCL
Clr_SCL MACRO
    clr1            SCL
    nop
    ENDM
;
;Pulse SCL
Emit_Clock MACRO
    Set_SCL
    Clr_SCL
```

```
    ENDM

; ================================================================================================

;
;variable definition
;
  RSEG UDATA2
  SADDR

bitcount DS 1
eeprom_bitreg ds 1



; ================================================================================================

;
;Start of executable code
;
  RSEG CODE
;
;Subroutines...
;
; ================================================================================================

;Start Sequence
Start:
    SDAOUT
    SCLOUT
    set1    SDA
    Set_SCL
    clr1    SDA
    Clr_SCL
    ret


; ================================================================================================

;Stop Sequence
Stop:
    SDAOUT
    SCLOUT
    clr1    SDA
    Set_SCL
    set1    SDA
    ret

; ================================================================================================

;
;Bus check routine, checks if I2C bus is free
;if not flag bus_error is set
;
Bus_check:
    clr1   bus_error
    clr1   cy
    SCLIN
    SDAIN
    bf     SCL,bus_fault
    bf     SDA,bus_fault    ;jump if bus fault
    SDAOUT
    SCLOUT
    ret
bus_fault:                  ;bus fault
    set1    bus_error       ;set error code
    set1    cy
```

```
    ret

; ============================================================================================

;
;Transmit a byte over the I2C bus
;input: acc contains byte to transmit
;output: cy = 0 if sequence completes
;        cy = 1 if unable to transmit
;on error the i2c error flag is set
;
Xmit_Byte:
        mov     bitcount,#8           ;8 bits to send
xb1:    rolc     a,1
        bnc     xb1a
        set1     SDA                  ;put bit on pin
        br      xb1b
xb1a:   clr1    SDA
xb1b:   Emit_Clock                    ;emit clock pulse
        dbnz    bitcount,xb1          ;loop until done

;setup to accept ACK from slave device
        set1    SDA                    ;release data pin
        Set_SCL                        ;SCL high
        SDAIN
        nop
        bf      SDA,xb2                ;jump if ACK seen
        SDAOUT
        Clr_SCL                        ;drop SCL
        set1    cy                     ;set error code
        set1    i2c_error
        ret
xb2:    SDAOUT
        Clr_SCL                        ;drop SCL
        clr1     cy                    ;set completion code
        clr1     i2c_error
        ret

; ============================================================================================

;
;Receive a byte over the I2C bus
;output: acc contains received byte
;        cy is dummied up with a 0
;
Rec_Byte: mov bitcount,#8             ;8 bits to receive
    SDAIN
zb1:nop
    Set_SCL                           ;SCL high
    nop
    bf     SDA,zb10
    set1     cy               ;pick bit off of pin
    br      zb11
zb10:  clr1  cy
zb11:  rolc     a,1
    Clr_SCL                           ;SCL low
    dbnz    bitcount,zb1      ;more bits to receive?
    clr1    cy                        ;must complete ok
    ret

; ============================================================================================

;
;Public routines...
;

;
```

```
;Reset Bus routine, tries to clear the bus after hang-up
;if no clearance is possible, flag bus error is set
;
_Reset_Bus:
    mov eeprom_bitreg,#0
    mov bitcount,#9
    SDAIN
_reset_loop:bt SDA,Reset_end
    Clr_SCL
    nop
    nop
    Set_SCL
    nop
    dbnz bitcount,_reset_loop
    set1 bus_error
Reset_end: ret

; ============================================================================================

;
;Transmit and Receive routine for adressable data
;Transmit device-address and register-adress over I2C bus
;transmits or receives databyte
;input: r1 contains slave address and contains received data
;       r3 contains register address
;       r2 contains data byte, if transmit is used
;output: cy = 0 if sequence completes
;        cy = 1 if unable to transmit
;

_Get_I2C_Reg:
    set1 i2c_read
    br   xrd1
_Put_I2C_Reg:
    clr1   i2c_read
xrd1:
    call   Bus_check
    bc     xrd_end
    mov    x,a              ;setup slave address
    call   Start            ;set Start condition
    call   Xmit_Byte        ;send slave address
    bc     xrd2             ;jump on error
    mov    a,r3             ;setup register address
    call   Xmit_Byte        ;send register address
    bc     xrd2             ;jump on error
    bt     i2c_read,xrd1b
    mov    a,r2             ;setup data byte
    call   Xmit_Byte        ;go send
    br     xrd2
xrd1b:
    call   Start            ;set repeated Start
    mov    a,x              ;setup slave address
    set1   a.0              ;indicate read operation
    call   Xmit_Byte        ;send slave address again
    bc     xrd2             ;jump on error
    call   Rec_Byte         ;go receive
                            ;store data byte
                            ;sequence complete, return code is already in cy:
    set1   SDA              ;set SDA idle
    Emit_Clock              ;emit clock pulse

;
;set Stop condition, return code is already in cy
xrd2:
    call   Stop             ;set Stop condition
xrd_end:
    ret
```

```
; ================================================================================================

;
;Transmit and receive routine for standard devices
;Transmit address and receives or transmits a data byte over I2C bus
;input: r1 contains slave address and contains received data
;       r3 contains data byte in transmit-mode
;output: cy = 0 if sequence completes
;        cy = 1 if unable to transmit
;
_Get_I2C_Byte:
    set1    i2c_read
    set1    a.0             ;indicate read operation
    br      xrdb1
_Put_I2C_Byte:
    clr1    i2c_read
xrdb1:
    call    Bus_check
    bc      xrdb1_end
                            ;setup slave address
    call    Start           ;set Start condition
    call    Xmit_Byte       ;send slave address
    bc      xrdb2           ;jump on error
    bt      i2c_read,xrdb1b
    mov     a,r3            ;setup data byte
    call    Xmit_Byte       ;send data byte
    br      xrdb2
xrdb1b:
    call    Rec_Byte        ;go receive
                            ;store data byte

;sequence complete, return code is already in cy
    set1    SDA             ;set SDA idle
    Emit_Clock              ;emit clock pulse

;set Stop condition, return code is already in cy
xrdb2:
    call    Stop            ;set Stop condition
xrdb1_end:
    ret

; ================================================================================================

;
;Transmit and Receive routine for serial I2C-EEprom type 24Cxx
;Transmit device-address and memory-adress over I2C bus
;transmits or receives one databyte
;input: r1 contains slave address and contains received data
;       r3 contains register address
;       r2 contains data byte, if transmit is used
;output: cy = 0 if sequence completes
;        cy = 1 if unable to transmit
;on error the eeprom error flag is set
;


_Recv_Eeprom:
        set1    eeprom_read
        br      eep1
_Send_Eeprom:
        clr1    eeprom_read
eep1:   push    rp2
        call    Bus_check
        bc      eep_end
        mov     x,a             ;save eeprom page
        push    rp1
```

```
        pop     rp2             ;free register R2 R3
        mov     r2,#5
eep_loop:
        set1    eeprom_error
        call    Start
        clr1    rw_flag
        mov     a,x
        call    send_eeprom_page
        bc      eep_err
        clr1    eeprom_error
        mov     a,r5            ;setup eeprom address
        call    Xmit_Byte
        bt      eeprom_read,eep2
        mov     a,r4            ;setup eeprom data
        call    Xmit_Byte
        br      eep_err
eep2:
        call    Start
        set1    rw_flag
        mov     a,x
        call    send_eeprom_page
        call    Rec_Byte

eep_err:
        call    Stop
        bt      eeprom_read,eep_err2
wtack:
        call    Start
        mov     R3,#wait
        mov     a,x
        call    send_eeprom_page
        bnc     eep_err1
        dbnz    R3,wtack
eep_err1:
        call    Stop

eep_err2:
        bf      eeprom_error,eep_end
        dbnz    r2,eep_loop

eep_end:
        pop     rp2
        ret


send_eeprom_page:
        and     a,#00000111b
        rol     a,1
        or      a,#10100000b
        clr1    a.0
        bf      rw_flag,send_eeprom_addr
        set1    a.0
send_eeprom_addr:
        call    Xmit_Byte
        ret

; ===============================================================================================


  end


; ===============================================================================================
; ===============================================================================================
```

Chapter 6 Conclusion

This note has given an outline of I²C theory, and has shown how an I²C master can be implemented using an NEC K-line microcontroller that has no dedicated I²C port. Speed of operation is determined largely by device choice and operating frequency, and often operation far below the nominal 100 KHz is acceptable. Although used here to access an external EEPROM, the routines may be used to interface to any I²C device, such as a Real Time Clock (RTC), analog to digital converter etc.