Programming the EZ-USB I²C Interface

Introduction

The Cypress EZ-USB family contains a general-purpose I²C interface through which the 8051 accesses standard I²C devices connected to the SCL and SDA pins. The I²C interface is actually dual-purpose, serving as an ID/boot loader during enumeration and then as a general 8051 interface once the 8051 is running. This note deals with the 8051 interface, and presents assembly language code for 8051 byte reads and writes to the I²C bus. Two versions of the code are included, the first polled and the second interrupt-driven.

To test the byte read and write subroutines, the code includes a "loopback" routine that runs on the EZ-USB Development Board. This board contains two Philips PCF8574 IO expander chips, which connect to the I²C bus and provide eight general-purpose input-output bits. Four pushbuttons and two dip-switches are read using one 8574, and a 7-segment readout is driven from the other 8574. The program loop continuously reads one 8574 and writes the other, so the switch states are instantaneously reflected in the 7-segment readout. A general-purpose EZ-USB output pin (PORTA bit 0) is also used to provide useful oscilloscope information for instructional/debug purposes.

Hardware Details

The EZ-USB Development Board uses two Philips PCF8574 IO expander chips to communicate with onboard switches and lights. The PCF8574 is described in detail in Philips Data Handbook IC12 (1996), "I²C Peripherals." These chips are connected as shown in *Table 1*.

Table 1. EZ-USB Dev Board PCF8574 IO Expander Functions

PCF8574	Addr	Sub	Dir	Used To	Notes
U12	0100	001	0	Drive 7-seg readout	Dir: b0=0 for write
U11	0100	000	1	Read switches	Dir: b0=1 for read
	Command Byte				

An I²C peripheral such as the PCF8574 is addressed using three fields:

1. A slave address (0100 for the 8574).
2. A sub-address which is set by strapping three address pins HIGH or LOW.
3. A direction bit (bit 0), 0 for write and 1 for read.

The "Command Byte" indicates the byte values used to address U12 and U11.

I²C Data Transfers

The 8051 communicates with the I²C bus using two registers, shown below:

I2CS			I²C Control and Status				7FA5
b7	b6	b5	b4	b3	b2	b1	b0
START	**STOP**	**LASTRD**	**ID1**	**ID0**	**BERR**	**ACK**	**DONE**
R/W	R/W	R/W	R	R	R	R	R
0	0	0	x	x	0	0	0

I2DAT			I²C Data				7FA6
b7	b6	b5	b4	b3	b2	b1	b0
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

The 8051 initiates a transfer by setting the START bit (I2CS.7), and then writing a command byte (*Table 1*) to the I2DAT register. When the I²C controller is ready (next paragraph) the 8051 then writes or reads data to/from I2DAT. Finally, the 8051 sets the STOP bit to terminate the transaction. The I2CS bits have the following functions:

DONE

When the 8051 initiates an I²C transfer, the DONE bit (I2CS.0) goes LOW, and returns HIGH when the transfer completes and the EZ-USB I²C controller is ready. A DONE bit 0-to-1 transition also generates an I²C interrupt request. The DONE bit is automatically cleared when the 8051 reads or writes I2DAT. The I²C interrupt request is automatically cleared when the 8051 reads or writes I2CS or I2DAT.

STOP

The 8051 terminates the I²C transfer by setting STOP=1. This bit stays HIGH until the I²C controller finishes sending the STOP condition on the I²C bus, at which time it clears the STOP bit. Completion of the STOP condition on the I²C bus has no effect on the DONE and I²C interrupt request bits.

For an I²C read operation, the 8051 must read the last data byte from I2DAT before the STOP condition completes (within about 11 microseconds). Therefore the 8051 code should read the I2DAT register immediately after setting the STOP bit. If the system uses interrupts, they should be disabled while the STOP bit is set and I2DAT is read to ensure that the I2DAT register is read immediately after setting the STOP bit. This is illustrated in the example code.

During the time that the EZ-USB controller is generating the STOP condition, it ignores accesses to I2CS and I2DAT. It is therefore good practice to begin every I²C transfer routine with a check for STOP=0, indicating that the stop condition has completed and the I²C controller is "listening." This allows immediate back-to-back calls to the I²C read and write subroutines.

LastRD

The 8051 sets the LASTRD bit (I2CS.5) before reading the last byte in a read operation. This instructs the EZ-USB I²C controller not to generate an ACK for the last transfer. The lack of an ACK from the I²C master (the EZ-USB I²C controller) signals the I²C peripheral to stop sending.

ACK, BERR

After every transfer (when DONE goes HIGH), two status bits indicate if the transfer received an ACK (I2CS.1=1) and if another I²C device interfered by driving the bus at the same time as the EZ-USB controller (Bus Error, BERR=1). For simplicity the example code does not check these bits after byte transfers.

ID1–ID0

These bits, which indicate the EEPROM type detected by the EZ-USB I²C boot loader, may be safely ignored for these examples. For reference, Chapter 4 of the *EZ-USB Technical Reference Manual* describes the meaning of these read-only bits.

Polled Code Description

Refer to Listing A at the end of this note. Because the code steps are commented in detail, this section gives a code overview.

After setting up PORTA.0 as an output at lines 32–34, the main loop at lines 36–44 pulses PORTA.0 HIGH then LOW, then calls the 'read_i2c_byte' and 'write_i2c_byte' subroutines. *Figure 1* shows the PORTA.0 trigger pulse and the four I²C bus transactions that comprise the read and write operations.

For this simple example the I²C command bytes are hard-coded to 0100 000 1 for the read (lines 60–62) and 0100 001 0 for the write (lines 105–107). These values correspond to the values shown in *Table 1* for the two PCF8574 IO expander chips. For generality you would probably want to keep the

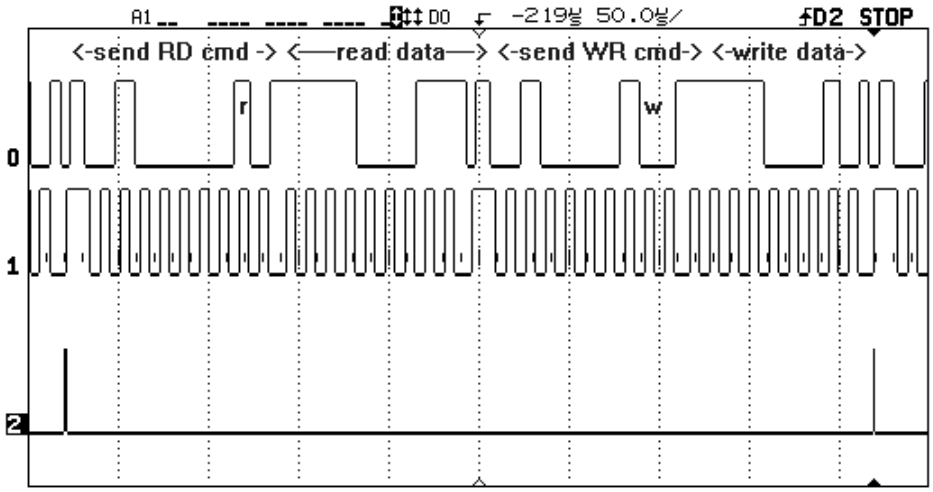


Figure 1. Scope Traces for Listing A

command byte in a variable, as is done in the interrupt example in Listing B.

Both the read and write subroutines begin by waiting until the STOP bit is zero. This is in case the routines are called repeatedly or back-to-back. Without this check, the STOP condition for the previous call may not have completed before the new call, and the I2CS register writes would be ignored.

8051 interrupts are turned off in line 80, just before setting the STOP bit, and turned back on in line 88, just after reading the data from the I2DAT register. This ensures that the I2DAT register is read before the STOP condition completes on the I²C bus.

Interrupt Code Description

Refer to Listing B at the end of this note. Listing B handles the I²C transfers using the I²C interrupt (8051 INT3). The obvious advantage is that the 8051 does not waste time polling the BUSY bit between I²C operations. Instead, the 8051 does operations necessary to start an I²C bus cycle, then returns to the main program. The I²C interrupt goes active when further 8051 action is needed (when the BUSY bit makes a 1-to-0 transition).

Figure 2 shows the four bus transactions that comprise the read and write operations. PORTA.0 (trace 2) drives LOW when the 8051 is executing the Interrupt Service Routine (ISR), and HIGH in the 8051 background program. By eyeballing trace 2 for percentage of the time LOW (in the ISR) it is evident that the I²C processing overhead is less than about ten percent.

The I²C interrupt vector is 8051 INT3, at 0x4B, where a jump to the I²C ISR is inserted at lines 57–58. These read and write routines expect to find the I²C address (command byte) in the variable 'i2addr', and the read or write data in the variable 'i2data'. The program sets a flag called 'i2busy' at the beginning of the ISR and clears it when the I²C transfer is complete. The 8051 background program (a simple loop at lines 72–78) checks the 'i2busy' flag to detect completion of the I²C transfer. Of course your background program will probably have more useful things to do than just polling this bit.

The read subroutine starts at line 80. The ISR uses a state variable called 'i2state' to figure out what to do next. The read subroutine starts the I²C transfer by checking for STOP=0 (line 88), setting START=1 (lines 90–92), and writing 'i2addr' to the I2DAT register (lines 94–97). Then it initializes the state variable to i2state=0, enables the I²C interrupt, and returns (lines 97–102).

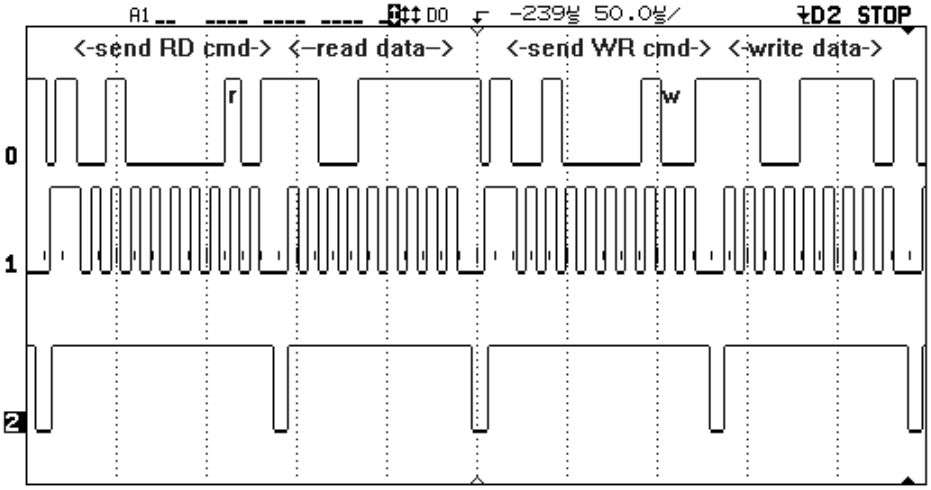
The write subroutine does the same steps as above, but initializes the state variable to i2state=2.

When the 8051 loads the I2DAT register, the I²C controller sends out the read or write command, shown as the first (read) and third (write) byte transfers in *Figure 2*. The little "r" and "w" letters in *Figure 2* indicate the direction bit (bit 0) of the command byte. After the command byte has been acknowledged, the I²C controller asserts the I²C interrupt and the ISR (i2c_isr) is entered at line 107.

The 'scope_lo' macro at line 114 sets PORTA.0 LOW for oscilloscope viewing (*Figure 2* trace 2). Then the I²C interrupt request (INT3) flag is cleared in lines 115–117. The interrupt request flag from the I²C controller must also be cleared, but this is done automatically when the 8051 reads or writes either the I2CS or I2DAT register.

The ISR then checks the 'i2state' variable in lines 119–126, and depending on its value, takes the actions shown in *Table 2*. A common exit point, 'i2exit' at line 155, returns the scope signal high, pops the saved registers, and returns from the interrupt.

The example code enables the I²C interrupt before performing a byte read or write, and disables the I²C interrupt at completion. This is probably an unnecessary step in a final system, where the I²C interrupt can be left enabled.



Trace 0 – SDA
Trace 1 – SCL
Trace 2 – PORTA.0 (DevBd P4-19)

DIP SW 1,2 OFF

Figure 2. Scope Traces for Listing B

Table 2. ISR Actions Based on i2state Variable

i2state	Action
0	Set LastRD bit, initiate an I2C read cycle, set i2state=1
1	Set the STOP=1, read data from I2DAT, disable I2C interrupt
2	Write data byte to I2C bus, set i2state=3
3	Set STOP=1, disable I2C interrupt

Listing A: Polled Code

```
; ---------------------------------------------------------------------------
; i2cpoll.A51  10-6-98  LTH
; EZ-USB i2c byte read and byte write routines.
; Read and write the PCF8574 i2c expander chips on the EZ-USB development board.
; The 8574 at sub-address 000 is used for input, connected to pushbuttons f1-f4
; and dip switches 1-2. The 8574 at sub-address 001 is used for output, and
; connected to a 7-segment readout. To test the i2c byte read and write
; routines, the program endlessly loops between reading the switches and writing
; their settings to the 7-seg readout.
;
; Port pin PA0 (EZ-USB dev board P4-19) is programmed to pulse HI at the beginning
; of the read-write loop. This allows easy scope triggering when viewing the I2C
; bus lines SCL and SDA.
; ---------------------------------------------------------------------------
$NOMOD51           ; disable predefined 8051 registers
$nolist
$INCLUDE (REG320.INC)     ; *** for the integrated 8051 core
$include (ezregs.inc)     ; EZ-USB register assignments
$list
;
NAME        i2cpoll
            ISEG AT 60H
stack:      ds      20
            CSEG AT 0
            ljmp    start
; ------------------------------------------------
            org     200h
start:      mov     SP,#STACK-1    ; set stack
;
; make PA0 an output for scope trigger
;
            mov     dptr,#OEA
            mov     a,#00000001b   ; output enable PA0
            movx    @dptr,a
;
loop:       mov     dptr,#OUTA
            mov     a,#00000001b   ; pulse the scope
            movx    @dptr,a
            mov     a,#00000000b
            movx    @dptr,a
;
            call    read_i2c_byte
            call    write_i2c_byte
            sjmp    loop
;-------------
read_i2c_byte:
;-------------
; 0. Make sure STOP is not in progress
;
            call    stop_check
;
; 1. Set the START bit
;
```

```
54              mov     dptr,#I2CS
55              mov     a,#10000000b   ; b7=start bit
56              movx    @dptr,a
57  ;
58  ; 2. Write the i2C expander address 0100 000 and indicate read operation (b0=1)
59  ;
60              mov     dptr,#I2DAT
61              mov     a,#01000001b
62              movx    @dptr,a
63              call    wait_done
64  ;
65  ; 3. Set LastRD bit
66              mov     dptr,#I2CS
67              mov     a,#00100000b   ; b5=LastRD
68              movx    @dptr,a
69  ;
70  ; 4. Read a dummy data byte to initiate the 9 SCL pulses for the read
71  ;
72              mov     dptr,#I2DAT
73              movx    a,@dptr
74              call    wait_done
75  ;
76  ; 5. Set the STOP bit
77  ;
78              mov     dptr,#I2Cs
79              mov     a,#01000000b
80              clr     EA
81              movx    @dptr,a
82  ;
83  ; 6. Read the data byte
84  ;
85              mov     dptr,#I2DAT
86              movx    a,@dptr
87              mov     r7,a           ; save data in R7
88              setb    EA
89              ret
90  ;--------------
91  write_i2c_byte:
92  ;--------------
93  ; 0. Make sure STOP is not in progress
94  ;
95      call    stop_check
96  ;
97  ; 1. Set the START bit
98  ;
99              mov     dptr,#I2CS
100             mov     a,#10000000b   ; b7=start bit
101             movx    @dptr,a
102 ;
103 ; 2. Write the i2C expander address 0100 001 and indicate write operation (b0=0)
104 ;
105             mov     dptr,#I2DAT
106             mov     a,#01000010b
107             movx    @dptr,a
108             call    wait_done
109 ;
110 ; 3. Write the data byte
111 ;
112             mov     dptr,#I2DAT
113             mov     a,r7
114             movx    @dptr,a
115             call    wait_done
116 ;
117 ; 4. Set the STOP bit
```

```
118 ;
119             mov     dptr,#I2CS
120             mov     a,#01000000b
121             movx    @dptr,a
122             ret
123 ;----------subroutines---------------------------------
124 ;
125 stop_check: mov     dptr,#I2CS
126 stck:       movx    a,@dptr
127             jb      acc.6,stck
128             ret
129 ;
130 wait_done:  mov     dptr,#I2CS
131 cd1:        movx    a,@dptr
132             jnb     acc.0,cd1
133             ret
134 ;
135             END
```

Listing B: Interrupt Code

```
 1  ; ---------------------------------------------------------------------------
 2  ; i2cint.A51  10-6-98  LTH
 3  ; EZ-USB i2c byte read and byte write routines.
 4  ; This is an interrupt driven version of 'i2cpoll.a51' (Polled I2C).
 5  ;
 6  ; Read and write the PCF8574 i2c expander chips on the EZ-USB development board.
 7  ; The 8574 at sub-address 000 is used for input, connected to pushbuttons f1-f4
 8  ; and dip switches 1-2. The 8574 at sub-address 001 is used for output, and
 9  ; connected to a 7-segment readout. To test the i2c byte read and write
10  ; routines, the program endlessly loops between reading the switches and writing
11  ; their settings to the 7-seg readout.
12  ;
13  ; Port pin PA0 (EZ-USB dev board P4-19) is programmed to output HI during background
14  ; code execution, and LO while in the interrupt service routine. This allows easy
15  ; scope triggering as well as measuring how much overhead is consumed by the ISR.
16  ; ---------------------------------------------------------------------------
17  $NOMOD51           ; disable predefined 8051 registers
18  $nolist
19  $INCLUDE (REG320.INC)      ; *** for the integrated 8051 core
20  $include (ezregs.inc)      ; EZ-USB register assignments
21  $list
22
23  NAME        i2cint
24
25  I2READ      equ     0
26  I2WRITE             equ     2       ; constants used to init. i2c state machine
27  READBUTS    equ     01000001b       ; PCF8574 unit 0 connected to switches
28  WRITE7SEG   equ     01000010b       ; PCF8574 unit 1 connected to 7-seg readout
29  ;
30              ISEG AT 60H             ; stack
31  stack:      ds      20
32  ;
33              DSEG AT 20H             ; bit mapped regs
34  flags:      ds      1
35  i2busy      equ     flags.0
36  ;
37  i2addr:     ds      1
38  i2data:     ds      1
39  i2state:    ds      1
40  ;---------------macros--------------
41  scope_hi    MACRO
42              mov     dptr,#OUTA
43              mov     a,#1
44              movx    @dptr,a
```

```
45              ENDM
46  ;
47  scope_lo    MACRO
48              mov     dptr,#OUTA
49              mov     a,#0
50              movx    @dptr,a
51              ENDM
52  ;---------end of macros---------------
53  ;
54              CSEG AT 0
55              ljmp    start
56  ;
57              org     4BH             ; int 3 (i2c) vector
58              ljmp    i2c_ISR
59  ; ------------------------------------------------
60              org     200h
61  ; ------------------------------------------------
62  start:      mov     SP,#STACK-1     ; set stack
63  ;
64  ; make PA0 an output for scope trigger
65  ;
66              mov     dptr,#OEA
67              mov     a,#00000001b    ; output enable PA0
68              movx    @dptr,a
69              scope_hi
70              setb    EA              ; interrupts ON
71  ;
72  loop:       mov     i2addr,#READBUTS
73              call    read_i2c_byte
74  rd_wait:    jb      i2busy,rd_wait
75              mov     i2addr,#WRITE7SEG
76              call    write_i2c_byte
77  wr_wait:    jb      i2busy,wr_wait
78              sjmp     loop
79  ;
80  read_i2c_byte:
81  ;
82  ; This subroutines kicks off an i2c read sequence. Waits for STOP complete,
83  ; sets START bit, writes i2c address, enables the i2c interrupt. The ISR state
84  ; machine, which is invoked each time the I2CS DONE bit goes true, completes the
85  ; transfer.
86  ;
87              setb    i2busy          ; show not done yet
88              call    stop_check
89  ;
90              mov     dptr,#I2CS      ; set the START bit
91              mov     a,#10000000b    ; b7=start bit
92              movx    @dptr,a
93  ;
94              mov     dptr,#I2DAT     ; write i2c addr and b0=1 for read
95              mov     a,i2addr        ; Fmt: uuuusssd where u=unit, s=subaddress, d=direc
96              movx    @dptr,a         ; start sending SCL clocks
97              mov     i2state,#I2READ        ; initialize the state
98  ;
99              mov     a,EIE           ; enable INT3
100             setb    acc.1           ; EIE.1 is EX3 interrupt enable
101             mov     EIE,a
102             ret                     ; INT3 ISR state machine takes over from here...
103 ;
104 ; Read or write an i2c byte using interrupts.
105 ; Implements a state machine, where states 0-1 are for reads and 2-3 for writes.
106 ;
107 i2c_isr:    push    dps
108             push    dpl
```

```
109             push    dph
110             push    dpl1
111             push    dph1
112             push    acc
113 ;
114             scope_lo                ; show we're in the ISR
115             mov     a,EXIF          ; clear the INT3 request
116             clr     acc.5
117             mov     EXIF,a
118 ;
119             mov     a,i2state
120             cjne    a,#0,cs1
121             sjmp    state0
122 cs1:        cjne    a,#1,cs2
123             sjmp    state1
124 cs2:        cjne    a,#2,cs3
125             sjmp    state2
126 cs3:        sjmp    state3
127 ;
128 ; State 0: Set LastRD bit, dummy read to initiate 9 SCL clocks
129 ;
130 state0:     mov     dptr,#I2CS      ; R/W to I2CS clears the i2c IRQ bit
131             mov     a,#00100000b    ; b5=LastRD
132             movx    @dptr,a
133 ;
134             mov     dptr,#I2DAT     ; dummy read to initiate 9 SCL clocks
135             movx    a,@dptr         ; read it
136             mov     i2state,#1      ; next state is 1
137             sjmp    i2exit
138 ;
139 ; State 1: Set the STOP bit, read the data from i2DAT
140 ;
141 state1:     mov     dptr,#I2CS      ; This clears the i2c IRQ bit
142             mov     a,#01000000b    ; b6=STOP bit
143             clr     EA              ; in case we're operating at low priority
144             movx    @dptr,a         ; write the stop bit
145 ;
146             mov     dptr,#I2DAT     ; read the data byte
147             movx    a,@dptr
148             mov     i2data,a        ; save the byte in i2dat
149             setb    EA              ; interrupts back on
150             clr     i2busy          ; indicate completion
151             mov     a,EIE           ; disable INT3
152             clr     acc.1           ; EIE.1 is EX3 interrupt enable
153             mov     EIE,a
154 ;
155 i2exit:             scope_hi
156             pop     acc
157             pop     dph1
158             pop     dpl1
159             pop     dph
160             pop     dpl
161             pop     dps
162             reti
163 ;
164 write_i2c_byte:
165             setb    i2busy          ; show not done yet
166             call    stop_check      ; wait for prior operation to complete
167 ;
168             mov     dptr,#I2CS      ; set the START bit
169             mov     a,#10000000b    ; b7=start bit
170             movx    @dptr,a
171 ;
172             mov     dptr,#I2DAT     ; write i2c addr
```

```
173             mov     a,i2addr
174             movx    @dptr,a         ; start sending SCL clocks
175             mov     i2state,#I2WRITE; initialize the state
176 ;
177             mov     a,EIE           ; enable INT3
178             setb    acc.1           ; EIE.1 is EX3 interrupt enable
179             mov     EIE,a
180             ret                     ; INT3 ISR state machine takes over from here...
181 ;
182 state2:     mov     dptr,#I2DAT
183             mov     a,i2data
184             movx    @dptr,a         ; this clears the i2c IRQ
185             mov     i2state,#3      ; next state is 3
186             sjmp    i2exit
187 ;
188 state3:     mov     dptr,#I2CS      ; set the STOP bit
189             mov     a,#01000000b
190             movx    @dptr,a
191             clr     i2busy          ; indicate completion
192             mov     a,EIE           ; disable INT3
193             clr     acc.1           ; EIE.1 is EX3 interrupt enable
194             mov     EIE,a
195             sjmp    i2exit
196 ;-------------------------------------------------
197 stop_check: mov     dptr,#I2CS
198 stck:       movx    a,@dptr
199             jb      acc.6,stck
200             ret
201 ;
202 wait_done:  mov     dptr,#I2CS
203 cd1:        movx    a,@dptr
204             jnb     acc.0,cd1
205             ret
206 ;
207             END
```