Z80-MBC2

I/O OpCodes

Reference manual

by X-LENT Electronics

Table of Contents

Z80-MBC2 I/O OpCodes

Introduction

In the firmware of the Z80-MBC2 are I/O Operation Codes (OpCodes) available to communicate with the Serial interface, I2C interface, RTC, SD Card or with the GPIO interface when you are working in the Z80 mode. You can address from Assembler, MBASIC, PASCAL, FORTH, C or which ever programming language you use under CP/M, the Z80-MBC2 interfaces via the firmware.

There are OpCodes to address the GPIO header connected to a MCP23017 16 bits IO extender, RTC, USER KEY, USER LED, I2C interface (see LCD I2C OpCodes) and also the possibility to configure the GPIO header as a parallel printer port which emulate a SPP interface.

The OpCodes are defined as Write or Read operations. Some OpCodes are reserved to use with the SD card, RTC, GPIO or other system IO. Watch out when you are using those in your application!

Firmware version

This Reference manual is written for firmware version S220718-R290823 and hardware version A040618. However, the Reference manual describes also additional features and OpCodes for which you need the modified firmware version S220718-R290823-A141225.
This version supports the reading of the Weekday from the RTC and the option for a I2C LCD2004 display. The Reference manual will be mentioned which firmware is required.

I2C LCD2004 support

Firmware S220718-R290823-A311025 supports a I2C LCD to display start-up and OpCode information, but you can also address the LCD via Opcodes to display information from your programs.

Connect the I2C LCD to the I2C J1 (IOEXP) connector of the Z80-MBC2 board. The I2C address needs to be set to 0x27 (7 bits notation). The Firmware detect the I2C LCD and displays additional information.

In the *IOS: Select Boot mode or System parameters* menu is added an extra menu item;
A: LCD display OpCodes (-> Enabled). Here you can enable or disable the LCD for displaying OpCode information. Disable the OpCode displaying when you do not need debugging, there in the LCD slows down the OpCode READ operation.

Note: Use an external power supply, there the USB power via the USB2Serial adapters is too less to power the board with an LCD attached to it.

How to address an OpCode

The firmware in the Atmega32A monitors the Z80 address line A0. When A0 becomes High (1) an I/O OpCode is set to the datalines.

The order to execute an I/O operation is by first sending a defined I/O OpCode which tells the firmware which I/O interface to address. The firmware in the Atmega32A stores the defined I/O Opcode temporarily. In the firmware code a defined OpCode is also referred to as STORE OpCode.

An Opcode can be a write or read Opcode, if the I/O operation is read or write. The stored OpCode operation must always precede an EXECUTE WRITE OpCode or EXECUTE READ OpCode operation. For multi-byte read, read sequentially all the data bytes without sending the defined I/O OpCode (STORE Opcode

operation) before each data byte to read.

Example GPIO WRITE

Example works with firmware version S220718-R290823 and up.

```
10 GPIOA = 3
20 IODIRA = 5
30 OUT 1, 5 : REM Write I/O OpCode to address the GPIO header is sent to the firmware
40 OUT 0, 255 : REM Write 255,  All GPIOA pins are defined as OUTPUT
50 OUT 1, 3 : REM  Write I/O OpCode to tell the GPIOA that the next byte will define the pin state
60 OUT 0, 255 : REM Write 255, all GPIOA pins will be set to 1
70 OUT 1, 3 : OUT 0, 0 : REM Write 0, all PIOA pins will be set to 0
```

Example GPIO READ

Example works with firmware version S220718-R290823 and up.

```
10 GPIOA = 3
20 IODIRA = 5
30 OUT 1, 5 : REM Write I/O OpCode to address the GPIO header is sent to the firmware
40 OUT 0, 0 : REM Write 0,  All GPIOA pins are defined as INPUT
50 X = INP(0) : REM Read the GPIOA pins
```

Defined Opcodes for I/O write operations:

KeyWord	OpCode HEX	OpCode DEC	Bytes	Notes
USER LED	0x00	00	1	
SERIAL TX	0x01	01	1	
GPIOA Write	0x03	03	1	MCP23017
GPIOB Write	0x04	04	1	MCP23017
IODIRA Write	0x05	05	1	MCP23017
IODIRB Write	0x06	06	1	MCP23017
GPPUA Write	0x07	07	1	MCP23017
GPPUB Write	0x08	08	1	MCP23017
SELDISK	0x09	09	1	Reserverd
SELTRACK	0x0A	10	2	Reserverd
SELSECT	0x0B	11	1	Reserverd
WRITESECT	0x0C	12	512	Reserverd
SETBANK	0x0D	13	1	Reserverd
SETIRQ	0x0E	14	1	Reserverd
SETTICK	0x0F	15	1	Reserverd
SETOPT	0x10	16	1	Reserverd
SETSPP	0x11	17	1	PRINTER Mode

KeyWord	OpCode HEX	OpCode DEC	Bytes	Notes
WRSPP	0x12	18	1	PRINTER Mode
No operation	0xFF	255	1	

Defined Opcodes for I/O read operations:

KeyWord	OpCode HEX	OpCode DEC	Bytes	Notes
USER KEY	0x80	128	1	
GPIOA Read	0x81	129	1	MCP23017
GPIOB Read	0x82	130	1	MCP23017
SYSFLAGS	0x83	131	1	Reserverd
DATETIME	0x84	132	7	RTC DS3231
ERRDISK	0x85	133	1	Reserverd
READSECT	0x86	134	512	Reserverd
SDMOUNT	0x87	135	1	Reserverd
ATXBUFF	0x88	136	1	Reserverd
SYSIRQ	0x89	137	1	Reserverd
GETSPP	0x8A	138	1	PRINTER Mode
No operation	0xFF	255	1	

GPIO OpCodes

GPIO header J7 pins

The GPIO header is built around the MCP23017. The MCP23017 is a I2C I/O port expander with 16 individually configurable inputs/ outputs. The I/O pins are divided into two groups of ports of 8 bits each and named GPIOA and GPIOB. Each pin can handle 25mA in Input or Output mode. The total current to drain or sink of the MCP23017 is 150mA. Make sure you do not go above this otherwise it will damage the MCP23017.

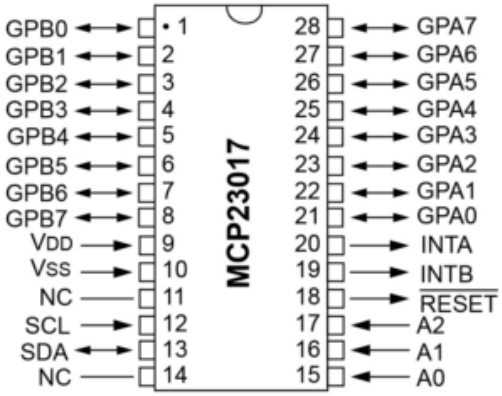
The I2C address of the MCP23017 used in the Z80-MCB2 is 0x20 (7 bits notation). This address is reserved for the GPIO header and cannot be used for other I2C devices.

The GPIO header layout for the Z80-MBC hardware A040618

J7 (GPIO)

PIN #	NAME		NAME	PIN #
1	VCC		VCC	2
3	GPB0		GPA7	4
5	GPB1		GPA6	6
7	GPB2		GPA5	8
9	GPB3		GPA4	10
11	GPB4		GPA3	12
13	GPB5		GPA2	14
15	GPB6		GPA1	16
17	GPB7		GPA0	18
19	GND		GND	20

GPIO header



MCP23017

GPIO Opcodes

GPIO OpCodes works with firmware version S220718-R290823 and up.

The available OpCodes to address the GPIO header are;

KeyWord	OpCode HEX	OpCode DEC	Bytes	Note
GPIOA Write	0x03	03	1	OUT 1, 3 : OUT 0, <data>
GPIOB Write	0x04	04	1	OUT 1, 4 : OUT 0, <data>
IODIRA Write	0x05	05	1	OUT 1, 5 : OUT 0, <data>
IODIRB Write	0x06	06	1	OUT 1, 6 : OUT 0, <data>
GPPUA Write	0x07	07	1	OUT 1, 7 : OUT 0, <data>
GPPUB Write	0x08	08	1	OUT 1, 8 : OUT 0, <data>
GPIOA Read	0x81	129	1	OUT 1, 129 : x = INP(0)
GPIOB Read	0x82	130	1	OUT 1, 130 : x = INP(0)

The **IODIRA** or **IODIRB** register programs the pins corresponding to bits D [7..0] as input or output. A 1 set the port to input, a 0 to output. So IODIRB = b10001000 means that the pins GPB7 and GPB3 are input, the remaining are output.

The **GPIOA** or **GPIOB** register contains the value of the logic status of the pins corresponding to the bits D [7..0] of the micro-controller. For example, during the writing phase, it brings the output of the GPIOA pins [7..0] to the logical state as indicated by the byte passed. On the other hand, during the reading phase, it copies the logic state of the GPIOA pins [7..0] into the peripheral input variable of the program.

The **GPPUA** or **GPPUB** register enables or disables the internal pull-up. If the pin is marked as input and the corresponding bit of GPPUA or GPPUB is at 1, then the pull-up 100 kΩ resistor is applied.

Operationally, to activate a register, a “1” must be sent to the corresponding address, then to write a “0” followed by the byte must be sent.

Example in MBASIC: 10 OUT 1, <register> : OUT 0, <data>

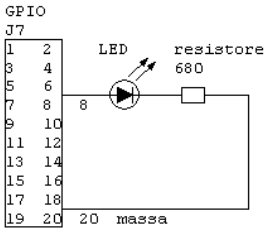
In other programming languages it might be OUT(1, <register>) and OUT(0, <data>)

Examples

Write example

To blink a LED, connect in this case a LED with his anode to GPIOA port 5 (GPA5). GPA5 is connected to the GPIO header pin 8. Connect the cathode of the LED via a resistor to ground, GPIO header pin 19 or 20. Activate the IODIRA register, which OpCode is 0x05 and define GPA5 as OUTPUT

The code LED blink;

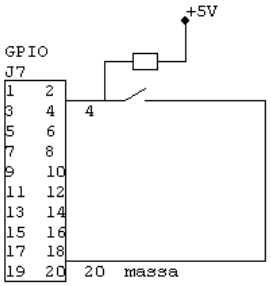



```
10 IODIRA = 5 : REM IODIRA write Opcode (0x05)
20 GPIOA = 3 : REM GPIOA write Opcode (0x03)
30 OUT 1, IODIRA : OUT 0,0 : REM Set all GPAx as output (IODIRA=0x00)
40 PRINT "Now blinking GPA5 (GPIO port pin 8)..."
50 REM * * * * * BLINK LOOP
60 OUT 1, GPIOA : OUT 0, 32 : REM Set GPA5=1, GPAx=0 (GPIOA=b00100000=32)
70 GOSUB 150
80 OUT 1, GPIOA : OUT 0, 0 : REM Set GPA5=0, GPAx=0 (GPIOA=b00000000=0)
90 GOSUB 150
100 GOTO 60
150 FOR J=0 TO 250
160 REM Introduce some delay
170 NEXT J
180 RETURN
```

Read example

To read the state of a button, connect a push button to GPA7, GPIO header pin 4. Connect a resistor of 10K between GPA7 and 5V.

The code to read a button state;




```
10 GPPUA = 7
20 GPIOA = 3
20 IODIRA = 5
30 OUT 1, GPPUA
40 OUT 0, b1000000 : REM Set GPA7 pull-up resistor
50 OUT 1, GPIOA
60 OUT 0, b1000000 : REM Set GPA7 as Input
70 FLOP = 1
80 WHILE FLOP
90 OUT 1, GPIOA
100 IF INP(0) = 128 THEN FLOP = 0
110 PRINT “Button is Pressed”
120 WEND
```

RTC – DS3231 OpCode

The Real Time Clock, DS3231, is an add-on module to the Z80-MBC2 board. The reserved I2C address of the RTC is 0x68 (7 bits notation). To address the RTC the following OpCode is available;

KeyWord	OpCode HEX	OpCode DEC	Bytes	Notes
DATETIME	0x84	132	7	RTC DS3231

The RTC is a read only OpCode. To set the correct Date, Time and Weekday, this is done from the *IOS: Select Boot mode or System parameters:* menu, option 9.

To use the RTC in a program, you have to read all RTC data at once. When you use firmware S220718-R290823-A311025 also the WeekDay number is given.

The data is order as follows: seconds, minutes, hours, weekofday, day, month, year, temperature

Example

For firmware version S220718-R290823, remove from the code at least line 150 and line 370.

```
10 REM ****************************************
20 REM
30 REM DATETIME PRINT
40 REM
50 REM ****************************************
60 PRINT CHR$(27);"[2J"; : PRINT CHR$(27);"[1;1H"
70 DIM WD$(7)
80 FOR I = 1 TO 7
90 READ WD$(I)
100 NEXT I
110 OUT 1,132 : REM Write the DATETIME read Opcode
120 SEC = INP(0) : REM Read a RTC parameter
130 MINUTES = INP(0) : REM Read a RTC parameter
140 HOURS = INP(0) : REM Read a RTC parameter
150 WDAY = INP(0) : REM Read a RTC parameter
160 DAY = INP(0) : REM Read a RTC parameter
170 MNTH = INP(0) : REM Read a RTC parameter
180 YEAR = INP(0) : REM Read a RTC parameter
190 TEMP = INP(0) : REM Read a RTC parameter
200 IF TEMP < 128 THEN 220 : REM Two complement correction
210 TEMP = TEMP – 256
220 PRINT
230 PRINT "*** Read the Date, Time & Temperature from the RTC module ***"
240 PRINT
250 PRINT "THE TIME IS: ";
260 IF HOURS < 10 THEN PRINT "0"; : PRINT USING "#"; HOURS;
270 IF HOURS > 9 THEN PRINT USING "##"; HOURS;
280 PRINT ":";
290 IF MINUTES < 10 THEN PRINT "0"; : PRINT USING "#"; MINUTES;
300 IF MINUTES > 9 THEN PRINT USING "##"; MINUTES;
310 PRINT ":";
320 IF SEC <10 THEN PRINT "0"; : PRINT USING "#"; SEC
```

```
330 IF SEC > 9 THEN PRINT USING "##"; SEC
340 PRINT
350 PRINT "THE DATE IS: ";
360 YEAR= YEAR+ 2000
370 PRINT WD$(WDAY); : PRINT ", ";
380 IF DAY < 10 THEN PRINT "0"; : PRINT USING "#"; DAY;
390 IF DAY > 9 THEN PRINT USING "##"; DAY;
400 PRINT "-";
410 IF MNTH < 10 THEN PRINT "0"; : PRINT USING "#"; MNTH;
420 IF MNTH > 9 THEN PRINT USING "##"; MNTH;
430 PRINT "-";
440 PRINT USING "####"; YEAR
450 PRINT
460 PRINT "THE TEMPERATURE IS: ";
470 PRINT USING "##"; TEMP;
480 PRINT CHR$(96);
490 PRINT " Celsius"
500 PRINT
510 FOR I = 0 TO 825
520 REM Dealy of app. 1 second
530 NEXT I
540 PRINT CHR$(27);"[2J"; : PRINT CHR$(27);"[1;1H"
550 GOTO 110
560 DATA "Sunday", "Monday", "Tuesday", "Wednesday"
570 DATA "Thursday", "Friday", "Saturday"
```

LCD I2C OpCodes

The LCD OpCodes works only with firmware version S220718-R290823-A311025.

The firmware S220718-R290823-A311025 supports a I2C LCD module LCD2004, a 20 character by 4 lines display. You can address the I2C LCD via a set of predefined OpCodes which represent a specific Keyword. The Keywords are compatible with the Liquied_I2C library. The I2C address of the I2C LCD is hardcoded in the firmware and must to be set to 0x27 (7 bits notation).

The OpCodes to access the LCD are;

KeyWord	OpCode HEX	OpCode DEC	Bytes	Note
clear	0x13	19	1	OUT 1, 19 : OUT 0, 0
home	0x14	20	1	OUT 1, 20 : OUT 0, 0
noDisplay	0x15	21	1	OUT 1, 21 : OUT 0, 0
display	0x16	22	1	OUT 1, 22 : OUT 0, 0
noBlink	0x17	23	1	OUT 1, 23 : OUT 0, 0
blink	0x18	24	1	OUT 1, 24 : OUT 0, 0
cursor_off	0x19	25	1	OUT 1, 25 : OUT 0, 0
cursor_on	0x1A	26	1	OUT 1, 26 : OUT 0, 0
scrollDisplayLeft	0x1B	27	1	OUT 1, 27 : OUT 0, 0

KeyWord	OpCode HEX	OpCode DEC	Bytes	Note
scrollDisplayRight	0x1C	28	1	OUT 1,28 : OUT 0, 0
leftToRight	0x1D	29	1	OUT 1, 29 : OUT 0, 0
rightToLeft	0x1E	30	1	OUT 1, 30 : OUT 0, 0
noBacklight	0x1F	31	1	OUT 1, 31 : OUT 0, 0
backlight	0x20	32	1	OUT 1, 32 : OUT 0, 0
autoscroll	0x21	33	1	OUT 1, 33 : OUT 0, 0
noAutoscroll	0x22	34	1	OUT 1, 34 : OUT 0, 0
setCursor	0x23	35	3	OUT 1, 35 : OUT 0, <lcdLine> : OUT 0, <Position> lcdLine is 0 or 1 for LCD1602 or 0 to 3 for LCD2004. Position is 0 to 15 for LCD1602 or 0 to 19 for LCD2004.
print	0x24	36	1	OUT 1, 36 : OUT 0, <ASCII>

LCD Keyword explained

setCursor

The LCD is organized in lines and positions. The LCD2004 has 20 positions, 0 to 19, by 4 lines, 0 to 3.
The LCD1602 has 16 positions, 0 to 15, by 2 lines, 0 to 1.

To set the cursor on a specific position on the LCD you sent first the OpCode byte followed by the line byte and then the position byte.

To set the cursor to line 2 at position 15 on a LCD2004, you sent the following OUT command order;

OUT 1, 35 : OUT 0, 2 : OUT 0, 15 or OUT 1, &H23 : OUT 0, &H02 : OUT 0, $H0F

print

To print a character on the LCD, you have to sent the ASCII value of the character. To send a complete word or sentence, you chop up the word or sentence and sent the ASCII value one by one. The best way to do this is to write in your program a PRINT procedure.

To print a capital “A” to the LCD; OUT 1, 36 : OUT 0, 65 (Decimal 65 is the capital ASCII value for A)

Example LCD

Print Hello World! and Z80-MBC2 on line 2, position 5 on the display.

```
10 PRINT "LCD I2C OpCode Test"
20 LCDCLEAR = 19
30 LCDSETCURSOR = 35
40 LCDPRINT = 36
50 OUT 1, LCDCLEAR : OUT 0, 0
```

```
60 TEXT$ = "Hello World!"
70 GOSUB 120
80 TEXT$ = "Z80-MBC2"
90 OUT 1, LCDSETCURSOR : OUT 0, 2 : OUT 0, 5
100 GOSUB 120
110 END
120 FOR I = 1 TO LEN(TEXT$)
130 OUT 1, LCDPRINT : OUT 0, (ASC(MID$(TEXT$, I, 1)))
140 NEXT I
150 RETURN
```

I2C Interface OpCodes

The Firmware uses the Wire.h Arduino library to access a I2C Device. In the firmware not all Wire.h keywords are implemented, only the most common used. The used I2C Device address must be presented in 7 bit notation. To understand how Wire.h works, see documentation here.

The I2C OpCodes works only with firmware version S220718-R290823-A311025.

KeyWord	OpCode HEX	OpCode DEC	Bytes	Note
Wire.beginTransmission	0x30	48	1	OUT 1, 48 : OUT 0, <addr>
Wire.write	0x31	49	1	OUT 1, 49 : OUT 0, <data>
Wire.endTransmission	0x32	50	1	OUT 1, 50 : OUT 0, <reStart> reStart is 0 or 1. When reStart is 0, a I2C Stop condition is send to the i2C bus. When reStart is 1, a restart condition is sent so that a next byte can be send to the I2C device.
Wire.read	0x90	144	2	OUT 1, 144 : OUT 0, <addr> : x = INP(0)
Wire.read Extended	0x91	145	4	OUT 1, 144 : OUT 0, <bytesToRec> : OUT 0, <multiRead> : OUT 0, <addr> : x = INP(0) byteToRec is 1 to number of bytes you want to read form the I2C device multiRead is 0 or 1. When 0, a stop condition is sent after the read, when 1 restart condition is sent.
I2CDetect	0x92	146	1	OUT 1,146 : OUT 0, <addr> : X = INP(0)

How to use I2C Interface OpCodes with the Z80-MBC2

Writing to I2C Device

Wire.begin.Transmission OpCode

To start the write sequence to a I2C device we first sent the Wire.begin.Transmission OpCode and the I2C Device address in 7 bit notation.

To start I2C sequence in MBASIC we use:

OUT 1, 144 : OUT 0, <addr> or OUT 1, &H90 : OUT 0, &H<addr>

Wire.write OpCode

After the Wire.beginTransmission we can start writing data to the I2C Device. To write date we use the Wire.write Opcode, followed by the data we want to write.

To start I2C sequence in MBASIC we use:

OUT 1, 49 : OUT 0, <data>. <data> is a number between 0 and 255.
OUT 1, &H31 : OUT 0, &H<data>. <data> is a number between &H00 and &HFF.

You can sent multiple writes at once by repeating the Wire.write sequence, however the data will only be passed through to the I2C Device after a Wire.endTransmission OpCode.

Wire.endTransmission OpCode

Every Wire.write sequencer must end with the Wire.endTransmission OpCode. After the Wire.endTransmission OpCode the Wire.write data will be sent to the I2C device.

The Wire.endTransmission OpCode comes with 2 options, send I2c Stop condition or I2C Restart condition. Depending on you I2C Device type you can send a Stop or Restart condition.

For a Stop condition: OUT 1, 50 : OUT 0, 0 or OUT 1, &H32 : OUT 0, &H0
For a Restart condition: OUT 1, 50 : OUT 0, 1 or OUT 1, &H32 : OUT 0, &H1

A Restart condition is mostly used for a I2C Device which has a built-in register increment.

Example Writing to a I2C Device

As example we use a PCF8575 device. I2C address of the PCF8574 is set to 33 decimal (0x21 hex). A0 to 5V, A1 and A2 to GND. Connect to P0 to P7 LEDS. Anode of the LED to 5V via a resistor of 1K to P0 to P7.

```
10 PRINT “PCF8574 Blinking LED”
20 I2CSTART = 48
30 I2CWRITE = 49
40 I2CSTOP = 50
50 I2CADDR = 33
60 OUT 1, I2CSTART : OUT 0, I2CADDR
70 OUT 1, I2CWRITE : OUT 0, 0
80 OUT 1, I2CSTOP : OUT 0, 0
90 FOR I = 0 TO 150
100 REM Delay of app 500mS
110 NEXT I
120 OUT 1, I2CSTART : OUT 0, I2CADDR
130 OUT 1, I2CWRITE : OUT 0, 255
140 OUT 1, I2CSTOP : OUT 0, 0
150 FOR I = 0 TO 150
160 REM Delay of app 500mS
170 NEXT I
180 GOTO 60
```

Read from I2C Device

To read from a I2C Device we use the Wire.read OpCode. The wire.read Opcode has 2 options, a standard or extended one. Depending on the type of I2C Device you can use the standard (0x90) or the extended (0x91)

Wire.read and Wire.read Extended needs more clock cycles to read from a I2C Device then a Wire.write OpCode. A Wire.read or Wire.read Extended will slow down the execution of your program. Keep that in mind for time critical operations.

Wire.read

This is the standard Wire.read OpCode. It read from the selected I2C Device one byte and sends a Stop condition to the I2C bus.

The sequence for the Wire.read in MBASIC is as follows;

OUT 1, 144 : OUT 0, <addr> : x = INP(0) or OUT 1, &H90 : OUT 0, &H <addr> : x = INP(0

Example Wire.read from I2C Device

```
10 PRINT “PCF8574 Input test”
20 I2CSTART = 48
30 I2CWRITE = 49
50 I2CSTOP = 50
60 I2CREAD = 144
70 I2CADDR = 33
80 X = 0
90 OUT 1, I2CSTART : OUT 0, I2CADDR
100 OUT 1, I2CWRITE : OUT 0, 255
120 OUT 1, I2CSTOP : OUT 0, 0
130 OUT 1, I2CREAD : OUT 0, I2CADDR
140 X = INP(0)
150 IF X > 127 THEN PRINT "BUTTON PRESSED"
160 GOTO 80
```

Wire.read Extended

The Wire.read Extended is used when you need to read multi bytes from a I2C Device, used with I2C Devices which support multi byte read.

The Wire.read Extended is a complex one due to the options that needs to be placed in the right order. To read from a I2C Device multiple bytes, the firmware needs to know from which device, how many bytes are going to be read and if a stop or restart condition needs to be sent to the I2C bus.

The sequence for the Wire.read Extended in MBASIC is as follows;

OUT 1, 145 : OUT 0, <byteToRead> : OUT 0, <StopRestart> : OUT 0, <addr> : x = INP(0) or
OUT 1, &H91 : OUT 0, &H <byteToRead> : OUT 0, &H<StopRestart> : OUT 0, &H <addr> : x = INP(0)

The variable byteToRead is a value between 1 and 255.
The variable StopRestart is 0 for Stop condition and 1 for a Restart condition.
The variable addr is the well known I2C Device address in 7 bit notation.

Example Wire.read Extended Read from I2C Device

As example we use a PCF8575 device. I2C address of the PCF8574 is set to 33 decimal (0x21 hex). A0 to 5V, A1 and A2 to GND. Connect a pushbutton between Vcc and P7. Connect a resistor of 10K between P7 and GND. The example is meant to show the Wire.read Extended sequence in a program. The program read only one byte and sends a Stop condition.

```
10 PRINT “PCF8574 Wire.read Extended test”
20 I2CSTART = 48
30 I2CWRITE = 49
50 I2CSTOP = 50
60 I2CREADEXT = 145
70 I2CADDR = 33
80 X = 0
90 OUT 1, I2CSTART : OUT 0, I2CADDR
100 OUT 1, I2CWRITE : OUT 0, 255
120 OUT 1, I2CSTOP : OUT 0, 0
130 OUT 1, I2CREADEXT : OUT 0, 1 : OUT 0, 0 : OUT 0, I2CADDR
140 X = INP(0)
150 IF X > 127 THEN PRINT "BUTTON PRESSED"
160 GOTO 80
```

Example I2C Read & Write

In this example 7 LEDS are connected via a 1K resistor to P0 .. P6. A pushbutton is connected between Vcc and P7. A resistor of 10K is connected between P7 and GND. The LEDS will be blinking. When the pushbutton is pressed, the text BUTTON PRESSED! is printed. The Wire.read standard OpCode is used in this program.

```
10 PRINT "* * * I2C READ / WRITE Test with PCF8574 * * *"
20 I2CSTART = 48
30 I2CWRITE = 49
40 I2CSTOP = 50
50 I2CREAD = 144
60 I2CADDR = 33
70 X = 0
80 OUT 1, I2CSTART : OUT 0, I2CADDR
90 OUT 1, I2CWRITE : OUT 0, 0
100 OUT 1, I2CSTOP : OUT 0, 0
110 FOR I = 0 TO 150
120 REM Delay :-)
130 NEXT I
140 OUT 1, I2CSTART : OUT 0, I2CADDR
150 OUT 1, I2CWRITE : OUT 0, 255
160 OUT 1, I2CSTOP : OUT 0, 0
170 FOR I = 0 TO 150
180 OUT 1, I2CREAD : OUT 0, I2CADDR
190 IF (INP(0) AND 128) = 128 THEN PRINT "BUTTON PRESSED!"
200 REM Delay :-)
210 NEXT I
220 GOTO 80
```

I2CDetect

I2CDetect returns a 0 when a I2C Devices is detected. I2CDetect is best used in the initialization part of the program to detect if the I2C Device is present.

The sequence for the I2CDetect in MBASIC is as follows;

OUT 1, 146 : OUT 0, <addr> : x = INP(0) or OUT 1, &H92 : OUT 0, &H<addr> : x = INP(0)

Return values

I2CDetect returns the following values on which you can check.

- *0*: success.
- *1*: data too long to fit in transmit buffer
- *2*: received NACK on transmit of address
- *3*: received NACK on transmit of data
- *4*: other error
- *5*: timeout

I2C Wire.read Programmers note

When you run the I2C Read & Write example, you will see that the LEDs not on and off for the same period of time although the FOR .. NEXT loops are the same. This has to do that the Wire.read OpCode takes several clock cycles to execute. By changing line 170 to FOR I = 0 TO 75, the delay to blink the LEDS are more in the same rhythm.

The I2C Wire.read Extended even takes double the clock cycles than the Wire.read OpCode to execute. Keep in mind when you want to use the Wire.read or Wire.read Extended OpCode that it slows down your program. For time critical readings you must be aware of this.