# Microprocessors

Prepared by

**Dušan Ponikvar**

**University of Ljubljana**

November 2001

## Table of contents


1. FOREWORD TO EXPERIMENTS WITH MICROPROCESSOR BASED INSTRUMENTS ..... 5
2. MICROPROCESSOR SYSTEM ..... 14
3. RANDOM ACCESS MEMORY ..... 19
4. LIQUID CRYSTAL DISPLAY ..... 23
5. MULTIPLEXED LED DISPLAY ..... 28
6. MICROPROCESSOR BASED COUNTER ..... 33
7. RS 233 CONNECTION ..... 38
8. MICROPROCESSOR WITH BUILT IN ADC ..... 45
9. MICROPROCESSOR INTERRUPTS ..... 49
10. MICROPROCESSOR BASED MULTICHANNEL ANALYZER ..... 53

# 1. FOREWORD TO EXPERIMENTS WITH MICROPROCESSOR BASED INSTRUMENTS

**OBJECTIVE**

The objective of these experiments is to get familiarised with basics of microprocessor systems and signals within such system. Oscilloscope should be used to observe important signals. Test programs for microprocessor system are supplied in advance. Programming and copying of EPROMs can be practised.

**REVIEW**

Microprocessor based instruments perform the same function as their "microprocessor-less" counterparts, but their functions, handling, and in particularly user interface are enriched by the use of microprocessor. Microprocessor usually takes care of those functions which could be implemented in hardware but would take many integrated circuits (like displaying results, reading keyboard, counting and calculating results).

Such microprocessors usually do not need and have big computational power. However, many instruments require some special electronics (counters, interrupts, AD and DA converters, RS232 connection, printing, …). Microprocessors are introduced in instruments to reduce the number of components and to add flexibility of the design. That is why microprocessors used in instrumentation are usually not the same as used in personal computers. Industry is using microprocessors in instruments for more than two decades, and it has developed some standard types. Examples are 8031/8032 series from Intel, and 6800 series from Motorola. Other companies have similar products.

The 8031/32 series microprocessors includes two or three gated counters, 32 input/output lines, two external interrupt lines, 256 bytes of RAM (memory). They run at 12MHz providing 1μs execution time per instruction. In addition to normal byte-wide instructions they include powerful set of bit-wide instructions. Some even include ROM or Flash ROM. The price for such microprocessors is modest (at present from about 2USD up). Instruction set is rather modest, but highly oriented towards control applications. The detailed description of such microprocessor can be found in data books. Those standard microprocessors have been expanded in many directions (providing on chip storage space for program and data, providing more input/output lines…). At present a good expansion seems to be the SAB80C535, a CMOS version of the 8032 microprocessor. It has more (48) input/output lines. The microprocessor includes 8 analogue inputs to the built in ADC (analog to digital converter) with resolution of eight bits. There is also a watchdog timer.

Other microprocessors are oriented towards computation, and do not have any input/output lines. Alternatively they can deal with 16 bit numbers and have richer instruction set. Typical example is the 8085 series, the brother of the well known 8088 which is used in XT personal computers. Such microprocessors can also be used in

controlling instruments, but require more interfacing hardware between the instrument and microprocessor. They are microprocessors only; any counters, RS232 connections, watchdog timers and alike must be added separately. Our experiments will be limited to eight bit microprocessors, as they are typically used in microprocessor based instruments.

A microprocessor system based on SAB 80C535 will be assembled for experiments. The complete wiring diagrams of microprocessor system are given at fig.1 (auxiliary circuit for driving the bus for external devices), fig.2 (microprocessor system), and fig.3 (auxiliary circuits for experiments. The stuffing diagram for PCB is given at fig. 4, and the input/output connections of the board are given at fig.5.

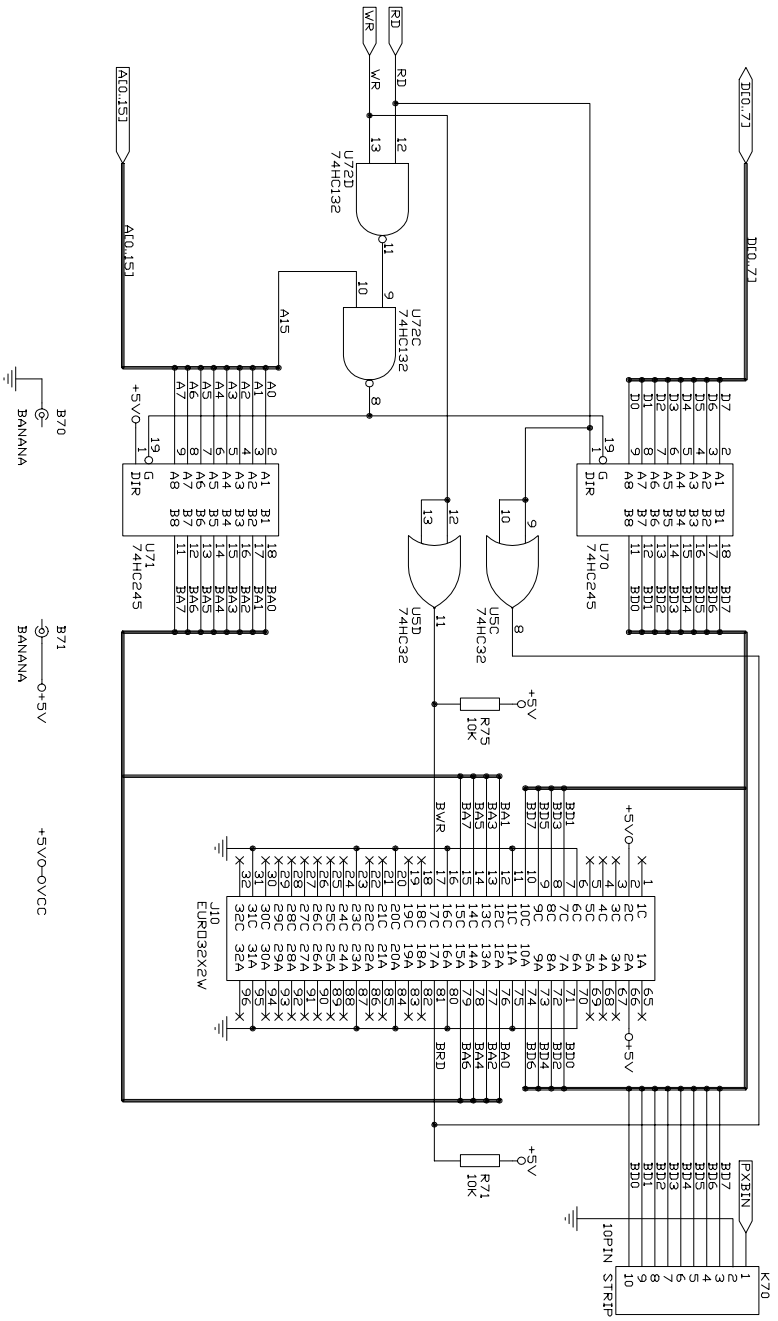



*Figure 1: Bus driver for external devices*

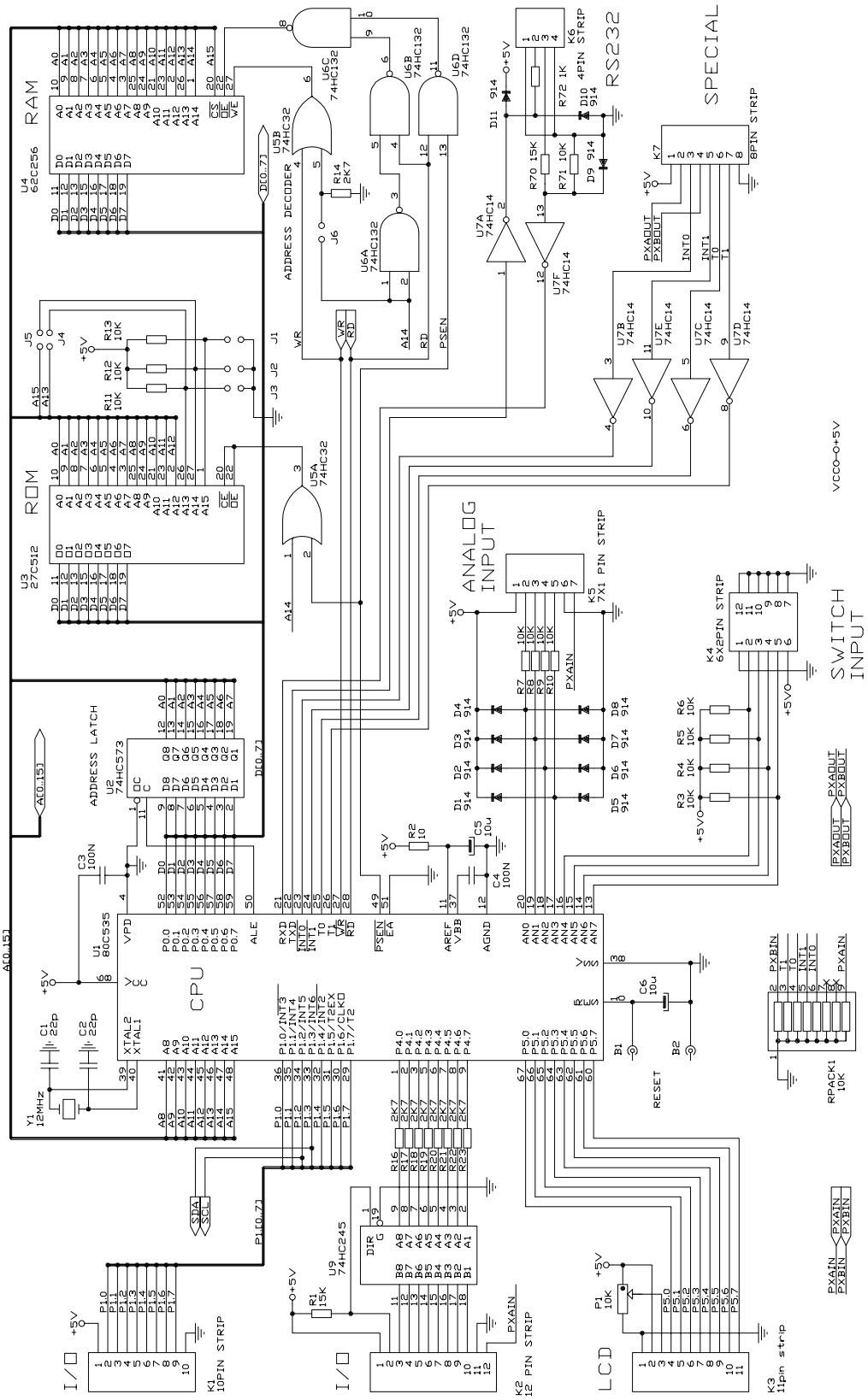



*Figure2: Wiring diagram of the microprocessor system*

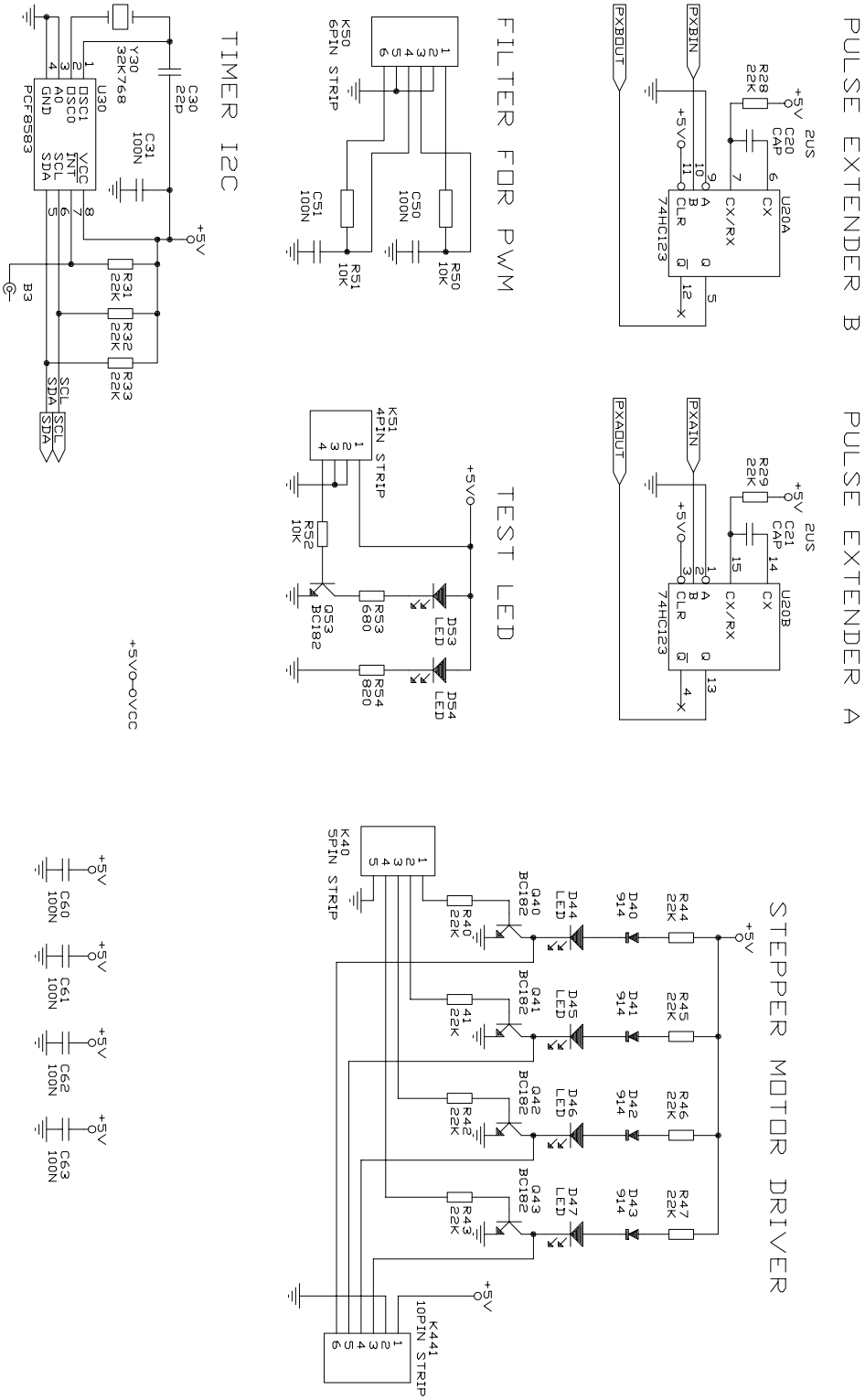



*Figure3: Auxiliary circuits for experiments*

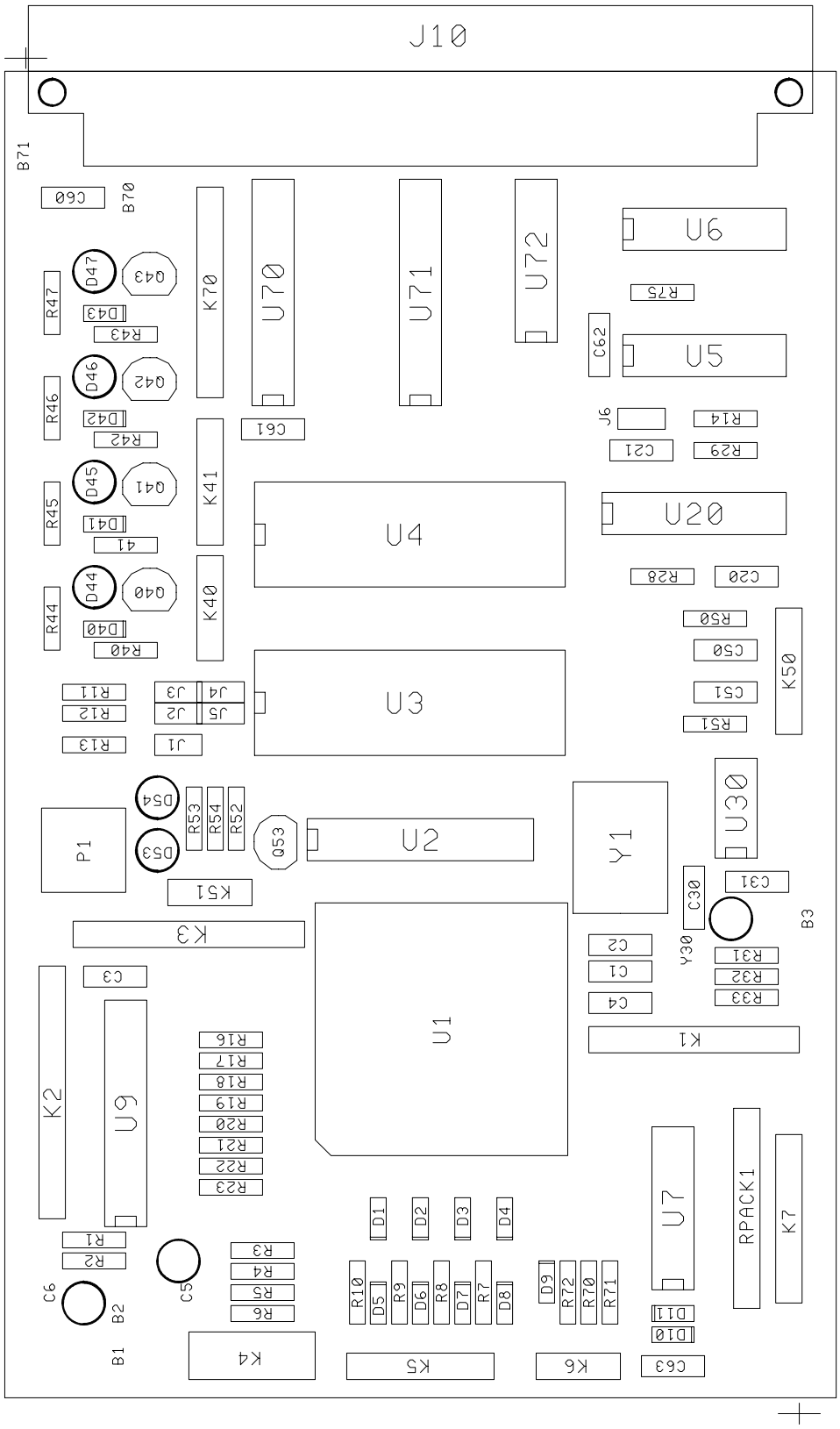



*Figure 4: Stuffing diagram of the printed circuit board*

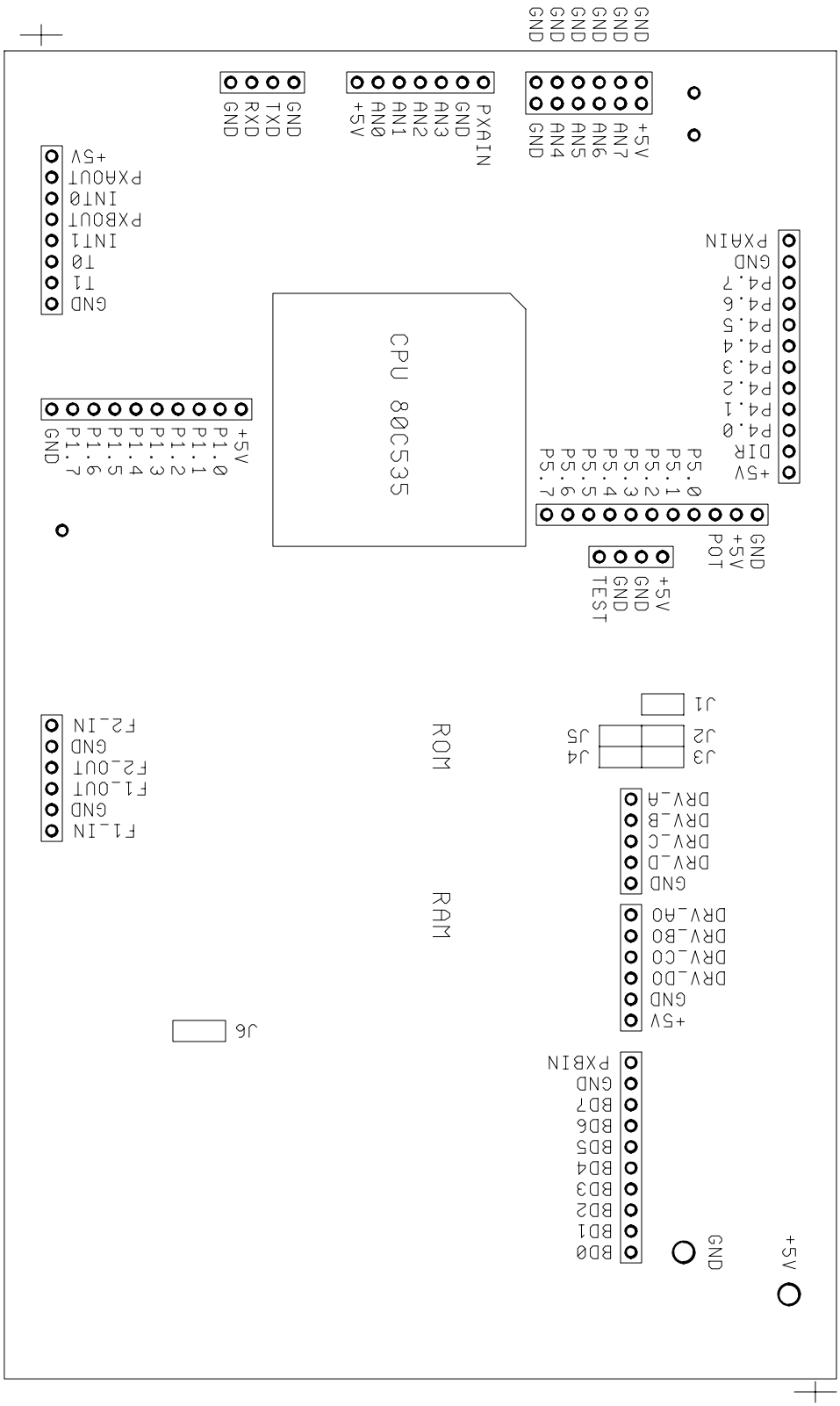



*Figure 5: Connections of the microprocessor system*

List of components for 80C535 based microprocessor system follows:

| Item | Quantity | Reference | Part |
|---|---|---|---|
| 1 | 3 | C1,C2,C30 | 22p |
| 2 | 7 | C3,C4,C31,(C50,C51),C60, C61,C62,C63 | 100N |
| 3 | 2 | C6,C5 | 10u |
| ( 4) | 2 | C21,C20 | selected |
| 5 | 15 | D1,D2,D3,D4,D5,D6,D7,D8, D9,D10,D11,D40,D41,D42, D43 | 1N914 |
| ( 6) | 6 | D44,D45,D46,D47,D53,D54 | LED 3mm, RED |
| ( 7) | 1 | J10 | EURO32X2W |
| 8 | 3 | K1,K441, (K70) | 10 PIN STRIP socket |
| 9 | 1 | K2 | 12 PIN STRIP socket |
| 10 | 1 | K3 | 11 pin strip socket |
| 11 | 1 | K4 | 6X2 PIN STRIP socket |
| 12 | 1 | K5 | 7X1 PIN STRIP socket |
| 13 | 2 | K51,K6 | 4 PIN STRIP socket |
| 14 | 1 | K7 | 8 PIN STRIP socket |
| 15 | 1 | K40 | 5 PIN STRIP socket |
| (16) | 1 | K50 | 6 PIN STRIP socket |
| 17 | 1 | RPACK1 | 8x10K, SIP |
| 18 | 1 | P1 | 10K |
| 19 | 15 | R3,R4,R5,R6,R7, R8,R9,R10,R11,R12,R13, (R50,R51),R52,(R71,R75),R76 | 10K |
| 20 | 5 | Q40,Q41,Q42,Q43,Q53 | BC182 |
| 21 | 2 | R1,R70 | 15K |
| 22 | 1 | R2 | 10 |
| 23 | 9 | R14,R16,R17,R18,R19,R20, R21,R22,R23 | 2K7 |
| 24 | 11 | (R28,R29),R31,R32,R33,R40 R41,R42,R43,R44,R45,R46, R47 | 22K |
| 25 | 1 | R53 | 680 |
| 26 | 1 | R54 | 820 |
| 27 | 1 | R72 | 1K |
| 28 | 1 | U1 | SAB 80C535 |
| 29 | 1 | U2 | 74HC573 |
| 30 | 1 | U3 | 27C512 |
| 31 | 1 | U4 | 62C256 or 62C64 |
| 32 | 1 | U5 | 74HC32 |
| 33 | 1 | U6, (U72) | 74HC132 or 74HC00 |
| 34 | 1 | U7 | 74HC14 |
| 35 | 1 | U9,(U70,U71) | 74HC245 |
| (36) | 1 | U20 | 74HC123 |
| (37) | 1 | U30 | PCF8583 |
| 38 | 1 | Y1 | 12MHz and 11.059MHz |
| (39) | 1 | Y30 | 32K768 |

Items in brackets are not needed for experiments presented in this book.

## Assembling instructions and precautions

Microprocessors and most of the other integrated circuits are CMOS devices, meaning that they do not tolerate any shocks from either static electricity or overvoltage at its inputs or outputs. Some handling precautions are necessary to keep these devices alive.

These devices usually come in static protective packaging; the plastic material wrapped around these devices is slightly conductive. This is enough to prevent any damage. Problems start, when the integrated circuit is taken out of the package. If any pin of the IC is exposed to the voltage, which is significantly different than the voltage at any other pin, the chip may die.

The chip may die completely, or be only partially damaged. A dead chip is easy to recognize, while the partially damaged chip may fail some functions, and work perfectly well otherwise. If you suspect the damaged chip, replace it with another one. It is very unlikely, that two chips are damaged in the same way.

It is very easy to prevent any shock from the static electricity. We must simply follow a rule to first bring anything, that will touch the chip, to the same electrical potential as the chip. To simplify: if the chip is wrapped into plastic, then we must first touch the plastic with one hand to bring us to the same potential of the plastic wrapping, and only then the chip. If the chip is inserted in the board, we must first with one hand take the board, and then with the other hand the chip. If the chip is at the table inserted in conductive foam, we must first take with one hand the conductive foam, and then with the other hand the chip from the foam. If we have the chip in our hand and want to insert it into the board, then we must first take and hold the board with the free hand, and then with the other hand insert the chip into the board, while still holding the board. If we have the chip in one hand and we want to give the chip to the other person, then we must first touch and hold this person with the free hand, and then give the person a chip while still holding the person.

Another question is what builds-up static electricity. We all know: very good insulation and rubbing of the materials with different properties. Can we avoid it? We can avoid good insulation (conductive foam, conductive mats, high humidity environment), but we can not really avoid rubbing of materials with different properties. Sometimes small things are enough: during the laboratory work in one of the IAEA courses eight groups experimented with microprocessor circuits. One out of eight groups killed several microprocessors, others did not make any damage. It was checked, the group did not make any errors in handling, but the chips kept on dying. Finally it was determined, that one participant of the training course from that group wears a shirt or underwear made of synthetic material, and that this could cause troubles. The next day he came in cotton, and no microprocessors were killed any more!

The power supply voltage for the CMOS digital circuits should never go above +5V, since otherwise chips might be damaged. Recent technology requires even +3V maximum to reduce the power consumption. This is also the truth for inputs and outputs from the circuit. Input voltages are limited into the range from 0V to +Vcc. The best way to kill a chip is to apply the input voltage, which in out of this allowed range. So guard your inputs, if you don't want to waste money.

As far as assembling of the circuit is concerned, always solder thin passive components (resistors, small capacitors) first. The reason is simple: it is easy to insert, and then put the stuffed PCB on the table upside-down. The table will press components into the board, and you will solder easily. Then come sockets (thicker in profile), then

come connection wires. The last to be inserted in sockets are the integrated circuits. It is a good idea to use the soldering iron with the grounded tip, and that the PCB itself is grounded during the soldering, but this is not mandatory. The author never uses grounding, a good soldering iron (Weller soldering station) and the above mentioned handling precautions suffice in a reasonable climate.

So do not be alarmed: the soldering of otherwise sensitive devices is not critical, if you follow some simple rules stated above. Try and see the results.

# 2. MICROPROCESSOR SYSTEM

**OBJECTIVE**

Microprocessor SAB 80C535 will be used throughout experiments. A board with microprocessor system will be assembled, and programs prepared in advance will be run on such system. Depending on the program and few additional electronic components different microprocessor based instruments will be introduced and their behaviour studied.

## Description of microprocessor system

**REVIEW**

Microprocessor system consists of at least three blocks, see fig.1. Main block is the microprocessor itself. In order for microprocessor to do anything useful, it must run a program. The program is stored in ROM (read only memory) or EPROM (erasable read only memory) as a chain of instructions. The microprocessor reads program instruction by instruction from the ROM in and executes it.

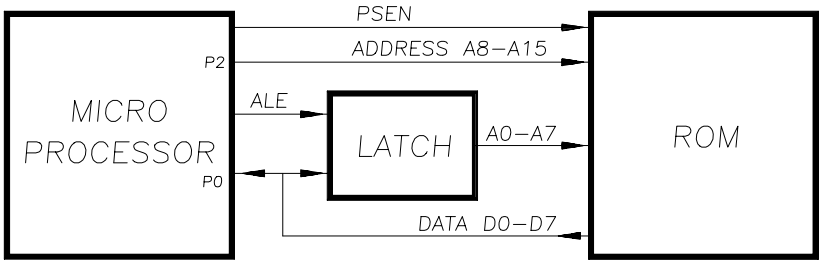



*Figure 1: The microprocessor system in blocks*

Microprocessor decides which instruction to read and when to read it. Therefore microprocessor supplies address to ROM (the location in ROM from where next instruction is stored) and signal for reading, this is named PSEN. Logic low on line PSEN requests instruction from ROM. Instructions are fetched from ROM into microprocessor through data lines D0 to D7. If either ROM or program in it is small then only lower part of address lines might be connected to ROM.

Producers of microprocessors tend to reduce the number of pins on microprocessor package to reduce the price of package. That is why lower eight address lines and eight data lines share the same eight pins on the package. Those pins are marked AD0 to AD7 and data lines are multiplexed with address lines. That means that in one moment lower eight address lines are present on eight pins AD0 to AD7 and later the same eight pins are used for eight data lines. Additional signal named ALE distinguishes the double use of the pins, see fig.2 for timing diagram. On high-to-low transition of this signal lower eight address lines are present at AD lines. Some names: program counter high (PCH, high byte of address in memory), program counter low (PCL, low byte of address in memory). An eight bit latch 74HC573 is used to separate lower address lines from data lines and this is the third block in diagram. High-to-low transition of signal AEN is used to latch address lines A0 to A7.

Wiring diagram of microprocessor system to be assembled and tested is given in fig.3. Only highlighted areas are tested in this experiment, and they were represented by the three blocks from fig.1.

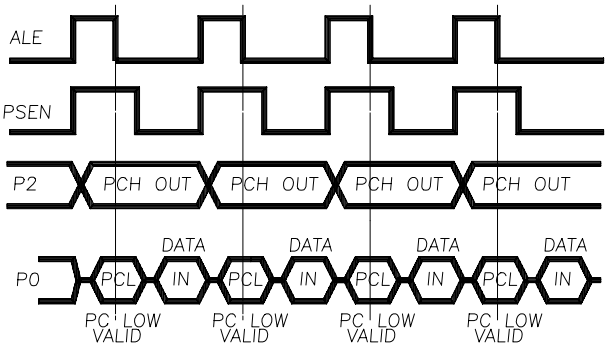



*Figure 2: Multiplexed address and data at port P0*

Note that some implementations of microprocessors include on-chip ROM, and therefore do not need external ROM and latch as shown here. In this case free data and address pins on microprocessor package can be used for other input/output of data.

Some remarks about the EPROM on board. The EPROM used is 64K by 8 bit EPROM. This is a large EPROM, and since test programs are short, it is never exploited fully. Such big EPROM is used for two reasons: it is at present cheaper than any other EPROM, and many programs can be programmed into it at the same time. Upper three address lines of EPROM are used to separate the whole EPROM space into eight blocks of eight kilobytes each, and each of these blocks can contain one program, see fig. 3. These upper three address lines are wired to pull up resistors, but can be grounded by inserting jumpers J1, J2 and J3. The insertion of jumpers selects one of 8K blocks from 64K EPROM. Example: insertion of jumpers J1, and omitting any other jumpers selects block No. 4, so microprocessor runs program written into block 4. Other address lines from A0 to A12 are supplied by microprocessor.

Microprocessor SAB80C535 includes 256 bytes of on chip random access memory, RAM. This memory can be used at programmers wish, and is sufficient for most control tasks. For other tasks as multichannel analysers external memory can be added, see next experiment.

Microprocessor has three eight bit input/output ports. These are marked P1, P4 and P5. Data can be sent out or read into microprocessor through these ports. Signals at these ports are TTL compatible.

**<u>Pins of microprocessor are quite sensitive. Do not apply any voltage out of 0-5V range, it may damage the processor !</u>**

**EXPERIMENT**

Assemble the highlighted part of circuit from fig. 3 using the stuffing diagram from experiment μP.0. Test the microprocessor system using programs from master EPROM. Programs are as follows:

Program #0: Start at address 0, then loops forever; the listing is given below:

```
$MOD515
        ORG     0H          ;start at 0 hex
start:  jmp     start       ;and repeat
        END
```

Program #1: Start at address 0, sends consecutive numbers to port P1, P4 and P5; the listing follows.

```
$MOD515
        ORG     0H          ;start at 0 hex
start:  inc     p1          ;increment port P1
        inc     p4          ;increment port P4
        inc     p5          ;increment port P5
        jmp     start       ;and repeat
        END
```

Program #2: Starts at address 0, reads port P4 and copies it to port P2; the listing follows.

```
$MOD515
        ORG     0H          ;arbitrary start
start:  mov     a,p1        ;copy content of P1 to accu,
        mov     p4,a        ;send accumulator out to P4
        jmp     start       ;and repeat
        END
```

Program #3: Toggles the line P4.3 twice per second; the listing follows.

```
$MOD515
        ORG     0H              ;arbitrary start
start:  setb    p4.3            ;set port P4, bit 3
        call    dly_1s          ;spend 1 second
        clr     p4.3            ;clear port P4, bit 3
        call    dly_1s          ;spend 1 second
        jmp     start           ;and repeat

dly_1s: mov     r1,#10          ;this is delay routine
o_loop: mov     r2,#100
m_loop: mov     r3,#250
i_loop: djnz    r3,i_loop
        djnz    r2,m_loop
        djnz    r1,o_loop
        ret
        END
```

For every program observe the activity on data and address line, and in particular the activity of PSEN line.

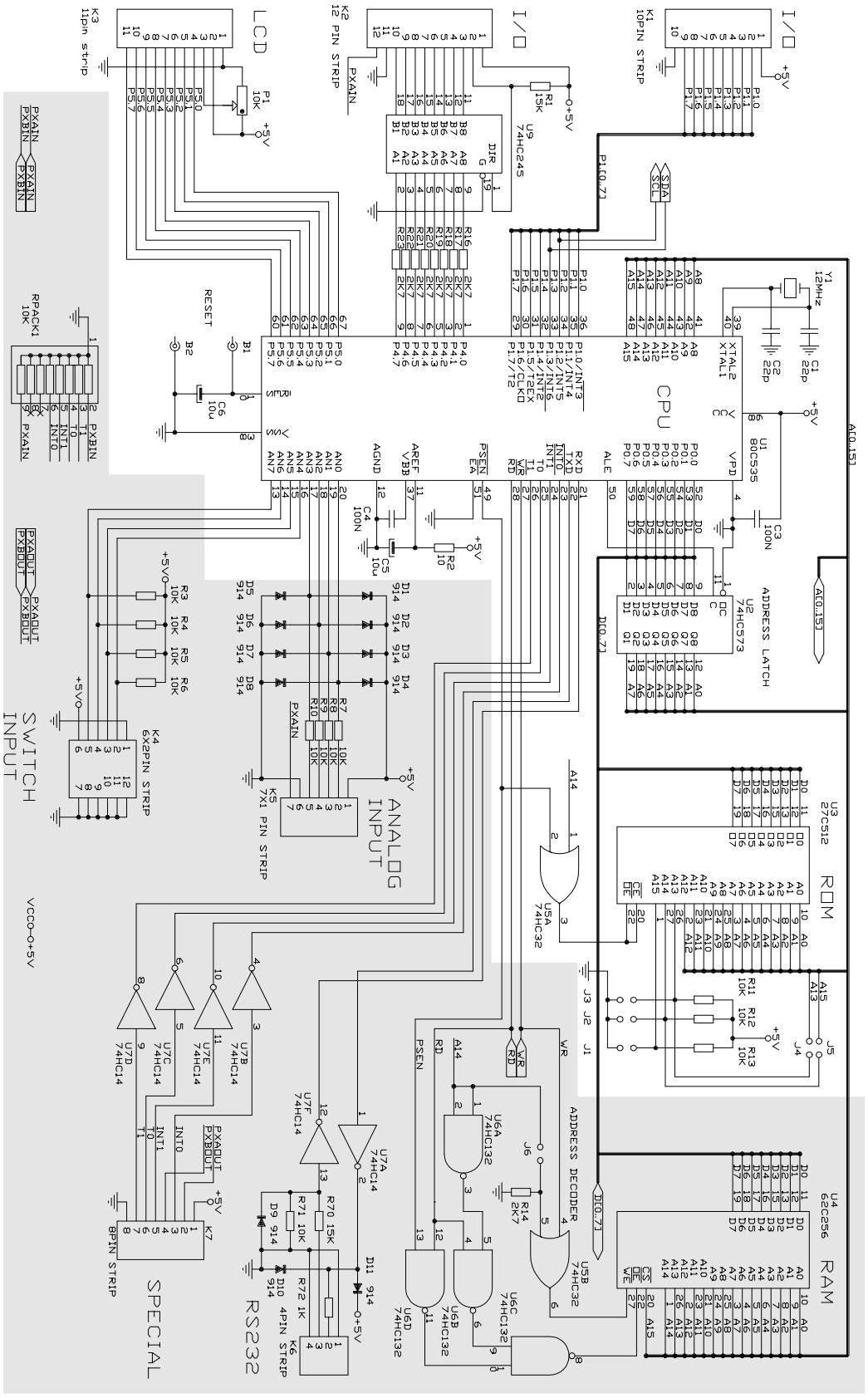



*Figure 3: Microprocessor and ROM*

## Power-up procedure and initial testing

Assume that our PCB was just soldered, and that the chips were inserted. We do not know if everything is OK, so it is not wise to apply the power from a universal power supply +5V/5A and see the result. Some smoke might come out of our circuit if the assembling was not done correctly, and ICs could die.

Therefore it is wise, that even after good optical inspection of our assembled circuit, we are still cautious about it's quality. Let us apply the power only for a moment (just a turn-on and immediate turn-off), and observe the power supply line with oscilloscope. Since +5V were applied for a moment, out board got the power. Then the power supply is off, and current for our circuit still comes from the capacitors in the power supply. If our board has a fault, then it consumes a lot of current, the capacitors will discharge rapidly and the +5V will drop immediately. But if everything is OK, then the power supply will slowly decrease to 0V. The speed depends on the power consumption of our circuit and the charge stored in the power supply after the power-off, but it is safe to say the if the +5V decreases back to 0V in more than 5 seconds, our board does not have a short between the +5V line and ground, and that the power supply can be on for longer time.

Once the power is on, we can use the oscilloscope to see if the microprocessor is alive. Every instruction is brought to the microprocessor from ROM through the data lines. The signal PSEN enables the output of a program code from ROM onto the data bus. The PSEN line should toggle periodically, it is low for every fetch of the instruction from ROM for about 0.3μs, and high for about 0.6μs in between with the clock frequency of 12MHz.

Instructions are read from the ROM at the specified address, and the address bus is multiplexed with the data bus for this microprocessor. Only the '573 latch separates address lines from data bus. The latch needs signal ALE to store lower address bus into itself, check the line ALE for toggle with a frequency of about 6MHz! Since the microprocessor reads program from ROM at different addresses, we expect that at least lower portion of address lines will change: check lower address lines for toggling! Be aware, that the program for the microprocessor might be very short and written in few memory locations at the beginning only, and that upper address lines may never become high; this is the case with almost all our short programs.

The program code from ROM should be transferred to microprocessor, we can check the toggling of the data lines at the microprocessor. It is extremely unlikely, that the value read from the ROM is always the same. We expect that different bits will be read from the ROM, therefore all data lines at microprocessor should toggle!

If all these lines toggle, then we can believe, that microprocessor system is alive. It is now the software, that should be checked.

By the way: if two address or data lines are shorted, then the one pulling low is stronger (see the section on digital electronics), and the resulting signal is low. If any signal observed is not high or low, but stays in between, then something is most likely bad.

The power consumption of the microprocessor is about 40mA, and the rest of the circuit should not take more than 20mA. This, multiplied by 5V, gives the total power consumption of 0.2W, which can not heat up any component. If something is hot, then something is wrong! This is not necessarily the component itself, also its outputs could be shorted. Check this if something is warm!

# 3. RANDOM ACCESS MEMORY

**OBJECTIVE**

The objective of this experiment is to introduce the idea of the microprocessor bus, control signals, and address decoding. A random access memory is added to microprocessor system.

**REVIEW**

Microprocessor communicates with other devices through a bus. A bus can be divided into three sub-busses: data bus, control lines, and address bus.

Data bus is bi-directional, and eight bits wide for an eight bit microprocessor. Microprocessor can get data or give data through these lines. Data can have any meaning (a command, a number, a result, a character…), but it is eight bit in width for eight bit microprocessor, as we are using it.

Microprocessor defines direction of data travelling through data bus. Microprocessor has control lines, and logic levels on these lines define the direction. Well known Intel family of microprocessors has control lines named RD, WR, and PSEN. When data is not transferred all these lines are logic high. When processor is reading data line RD is low, and line WR is low during sending data to a device on bus. Line PSEN has a special meaning; it is low every time microprocessor is reading instruction from read-only memory (ROM).

Since all devices are connected to the bus microprocessor must select one of them to communicate with. This is done through address bus. It is 16 bits wide for Intel 8031/32 family of microprocessors. There are 65536 possible combinations of signals at 16 bit address bus. A typical device will respond to one of them only; an address decoder associated with the device takes care of this. Microprocessor selects one device by asserting appropriate logic levels on address bus.

Every device connected to a bus must have unique address so only one of them is accessed by microprocessor at a time. Special circuit called address decoder takes signals from address bus and derives from them so-called chip select lines. A chip select line becomes active (usually this means going low) when a certain combination of signals is present at address bus. Observe wiring diagram at fig. 1. Signal CS3 goes low when combination of address lines A15 to A13 is 101 and control line WR is low. Address decoder like this can be used to write the content of data bus from micro-processor into latch 74HC574, if the line CS3 is used as clock for latch and processor asserts aforementioned combination of signals (processor writes to address A000 hex). Note that this address decoder is not particularly demanding since it

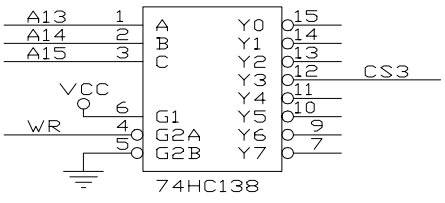



*Figure 1: A simple address decoder*

is possible to write into the same latch also at address A001hex, A002hex…. Much of the address space is wasted because one device responds to many addresses.

Memory devices (random access memory, RAM) require larger amounts of address space. In this case it is mandatory to use only upper part of address bus in address decoder, and lower part of the address bus is connected to RAM which makes decoding internally. Here a 32Kbyte (8Kbyte is also possible) RAM is used. Address lines A14 to A0 are connected to RAM, and address line A15 is connected to chip select (CS) input of RAM. This means that microprocessor can communicate with RAM only if it reads or writes to address space below 32Kbyte boundary (below 8000hex, in this case line A15 is low).

*Explanation of address decoder circuit used:* The RAM in microprocessor system is used for data. It should be possible to write into RAM or read from it regardless of the address within allowed space from 0 to 7FFF hex (32KBytes). However, the system presented can be used also for debugging and testing of assembly language programs. Programming of EPROM is time-consuming. It is much faster to copy program into RAM and then run it from there. To do this RAM should be able to send its content to data bus on two occasions: data read (line RD is low), or instruction read (line PSEN is low and address line A14 is high). Lines A14, PSEN and RD are combined in address decoding circuit U6A, U6B, U6C and U6D. If any of the two lines is low (the microprocessor request for instruction or data), then output of RAM is enabled and its content goes to data bus.

Microprocessor should be able to write into any location in RAM if RAM is used as data memory. This is true if jumper J6 is not inserted; control line WR solely controls write enable (WE) input of RAM. However, when testing assembly language routines as mentioned above, it is useful if microprocessor can not overwrite a program that was already copied into RAM. The insertion of jumper J6 prevents writing into RAM above address 3FFF hex (above 16kBytes). The testing procedure is then:

- remove jumper J6,
- copy test program into RAM from address 4000 hex on,
- insert jumper J6, and
- run test program.

**EXPERIMENT**

Assemble the highlighted part of the circuit, given at fig.2. Run prepared programs and check signals on the bus. Listings of programs are given below.

Program #1: Write number from port P1 into RAM

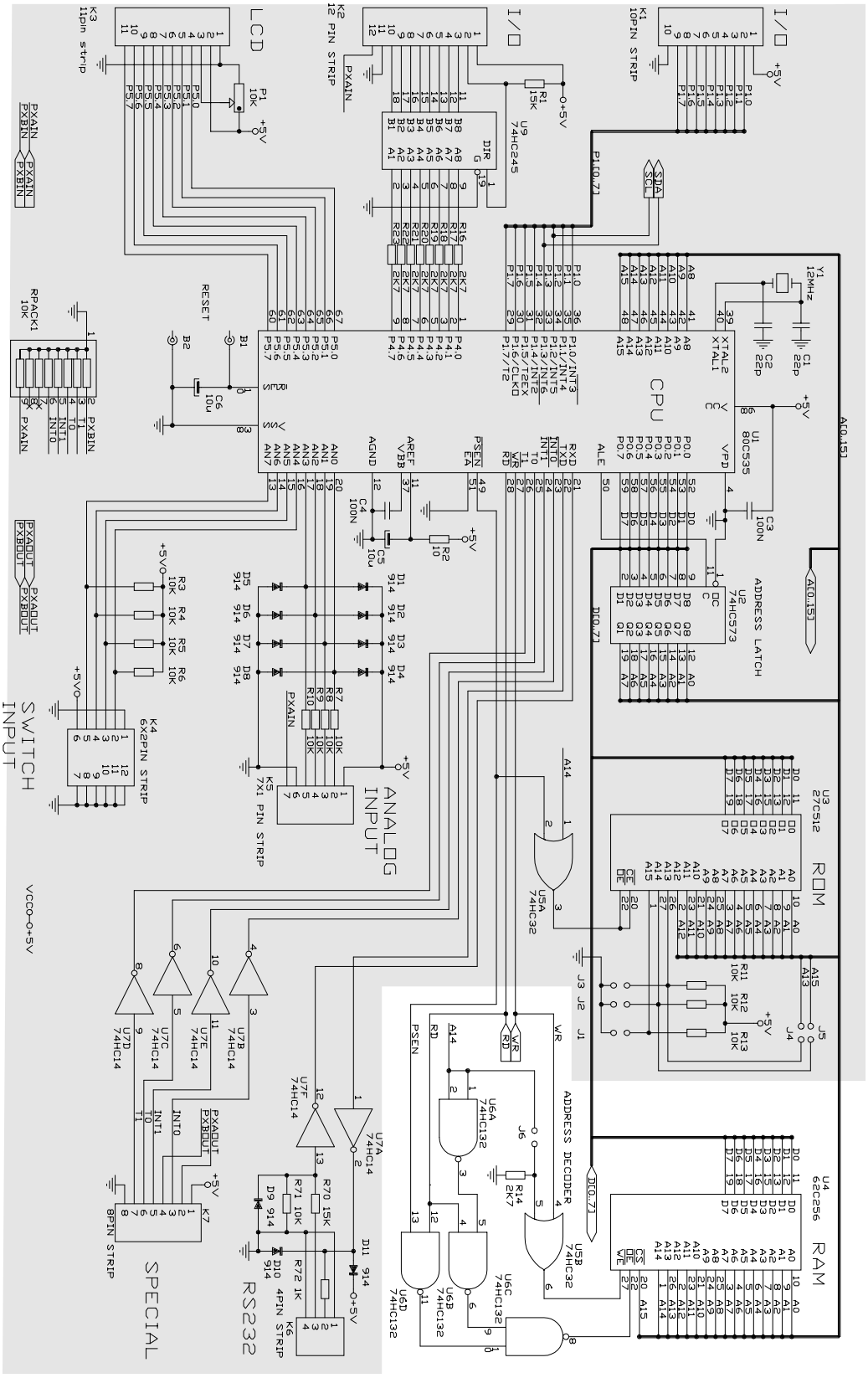



*Figure 2: Assemble the highlighted part of the circuit*

```
$MOD515
        ORG     0H              ;arbitrary start
start:  mov     dph,#0          ;set address in RAM to
16
        mov     dpl,#10h        ;
loop1:  mov     a,P1            ;read data from port P1
        movx    @dptr,a         ;write data to memory
        jmp     loop1           ;and repeat
        END
```

Program #2: Write number from port P1 into RAM at address given by port P5. Then read the same number from memory and copy it into port P4.

```
$MOD515
        ORG     0H              ;arbitrary start
start:  mov     dph,#0          ;upper addr bits are 0
loop1:  mov     dpl,P5          ;lower are at P5
        mov     a,P1            ;read data from P1
        movx    @dptr,a         ;write data to memory
        movx    a,@dptr         ;read data from memory
        mov     P4,a            ;send it to port P4
        jmp     loop1           ;and repeat
        END
```

## Fault identification

When microprocessor executes a program and writes into memory, a WR line is low for every write operation. Similarly, the RD line is low for every read from the memory. The read and write operations typically take 0.5μS. Observe the CS, WE and OE signals at the RAM: For writing the CS and WR should be low, for reading the CS and OE are low. If this is not so, then the data can not enter or exit the RAM.

You might want to try removing the RAM if you find the WR or OE lines stucked to either high or low when running the above programs. Stucked lines mean the either RAM is not operating (when stucked high) or that the RAM is disturbing the microprocessor data bus (when stucked low, the RAM could place it's content onto the data bus permanently and prevent ROM to send program code to the micro controller). If the lines remain stucked after the removal of the RAM, the address decoding circuit should be checked. If the lines are not stucked any more, the RAM could be faulty.

# 4. LIQUID CRYSTAL DISPLAY

**OBJECTIVE**

The objective of this experiment is to get familiarised with alphanumerical liquid crystal displays used in microprocessor based instruments.

**REVIEW**

Many microprocessor based instruments use liquid crystal display (LCD) for displaying results. The reason for widespread of LCDs is the possibility of displaying characters as well as numbers (to the limited extend even graphics), and low power consumption, typically few mA at most. These displays are available in variety of forms and shapes from widely used general purpose alphanumerical display modules to custom designed displays with special graphic symbols and characters.

An alphanumerical display module with 2 rows of 16 characters in each row will be used for experimenting, see fig. 1 for connections of the LCD module. The module includes interface to computer and memory for displayed characters. Data is transferred from computer to module and back in bytes through lines D0 to D7. Control signals read/write (R/W), enable (E) and register select (RS) are used to synchronise data flow from microprocessor. Voltage at pin CONT determines the contrast of characters displayed. It should be adjusted accordingly to the position of viewer and surrounding luminosity.

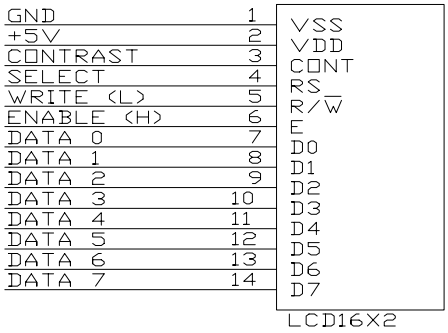



*Figure 1: Connections of LCD module*

Data sent to LCD module can be of two kinds: control bytes and characters to be displayed. Control bytes can change the appearance of display: number of rows, cursor, blinking… .

The content of display comes as ASCII data and is written into memory of the interface. Note that since data is written into memory it does not need to be updated frequently as with LED displays. Data written once is displayed until LCD module has power or until rewritten with another data.

The LCD module can either be connected directly to the microprocessor bus (this is possible with microprocessor running at 12MHz max.) or attached to a microprocessor port. If the LCD module is connected directly to the microprocessor bus, then some address decoding is necessary, see fig. 2 for example of interfacing to the 80C32 microprocessor bus. Data is transferred to the display in bytes of 8 bits.

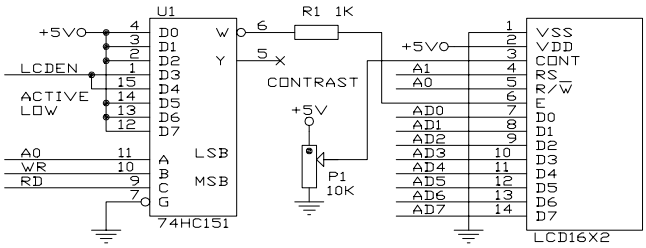



*Figure 2: The connection of LCD to the bus*

In the experiment presented LCD module is connected to microprocessor port P5. Port P5 is 8 bits wide. This is sufficient if LCD module is used in four bit mode; instead of sending eight bits simultaneously to module, a byte can be transferred in two parts by four bits each. LCD module can be programmed to accept this kind of data. Connector K3 of the microprocessor board connects directly to LCD module, see wiring diagram at fig.3 for connection. Voltage at pin 3 of LCD module determines contrast of characters on display. This voltage comes from a trimmer P1 on microprocessor board.

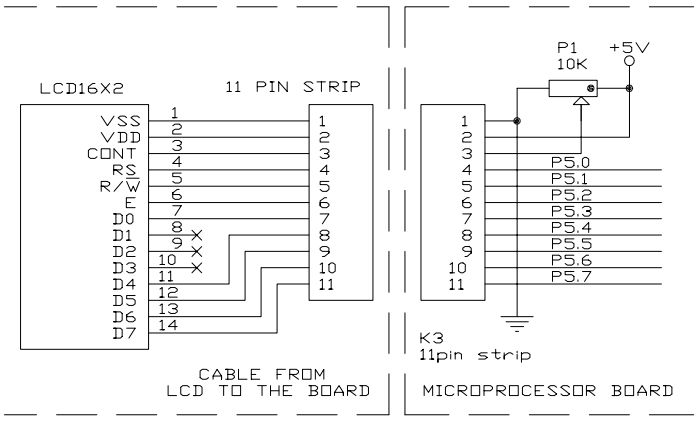



*Figure 3: The connection of LCD module to microprocessor system*

The controller in LCD module is automatically reset after power-up. It should be initialised before use. During initialisation the number of rows and the length of a row must be told to the controller. Also the required appearance of the display must be specified.

Throughout experiments a set of prepared routines for displaying characters, strings and numbers at specified position on LCD module will be used. A list of routines with calling parameters follows.

| Routine name | Action | Calling parameters |
|---|---|---|
| lcd_ini | initialises LCD | none |
| lcd_home | cursor home | none |
| lcd_clr | clear LCD | none |
| lcd_charpos1 | display single character at given position in first line | R1: ASCII code of character<br>R2: position on LCD |
| lcd_charpos2 | display single character at given position in second line | R1: ASCII code of character<br>R2: position on LCD |

| | | |
|---|---|---|
| lcd_string | display string of characters from ROM at given position; string is null terminated | DPTR: pointer to string<br>R2: position on LCD |
| lcd_1digits | display 1 BCD digit at given position | ACC, R1: corrupted<br>R2: position on LCD<br>R5: BCD digit, lower nibble |
| lcd_2digits | display 2 BCD digits at given position | ACC, R1: corrupted<br>R2: position on LCD<br>R5: two BCD digits<br>flag LZB: leading zero blanking |
| lcd_2digdot | display 2 BCD digits at given position, separated by comma | ACC, R1: corrupted<br>R2: position on LCD<br>R5: two BCD digits<br>flag LZB: leading zero blanking |
| lcd_4bcd | display 4 BCD digits at given position, leading zero blanking | ACC, R1: corrupted<br>R2: position on LCD<br>R5: less significant 2 BCD digits<br>R6: more significant 2 BCD digits<br>flag LZB: leading zero blanking |
| lcd_6bcd | display 6 BCD digits at given position, leading zero blanking | ACC, R1: corrupted<br>R2: position on LCD<br>R5: less significant 2 BCD digits<br>R6: more significant 2 BCD digits<br>R6: most significant 2 BCD digits<br>flag LZB: leading zero blanking |
| bin2bcd | convert 2 bytes binary into 6 BCD digits | R0: corrupted<br>R3: more significant byte<br>R4: less significant byte<br>R5: less significant 1 BCD digits<br>R6: more significant 2 BCD digits<br>R7: most significant 2 BCD |

| | | |
|---|---|---|
| | | digits |
| lcd_pos1 | set cursor at given position in first line | ACC: position |
| lcd_pos2 | set cursor at given position in second line | ACC: position |
| lcd_coff | hide cursor | none |
| lcd_con | show cursor | none |
| lcd_conb | show blinking cursor | none |
| del_1s | delay of 1 second | none |
| del_15ms | delay of 15 milliseconds | none |
| del_5ms | delay of 5 milliseconds | none |
| del_100 | delay of 100 microseconds | none |
| del_50 | delay of 10 microseconds | none |

**EXPERIMENT**

Connect LCD module to microprocessor system and run test programs which were written in EPROM. Observe sequence of signals at port P5.

Program: Write value from port P1 to LCD module.

```
$MOD515
        ORG     0H              ;arbitrary start
        call    lcd_ini
        call    lcd_coff        ;hide cursor
start:  mov     a,P1            ;data to be displayed
        mov     r4,a            ;set call for bin2bcd
        mov     r3,#0           ;
        call    bin2bcd         ;and convert bin2BCD
        mov     r2,#4           ;set call displaying
        clr     LZB             ;acti. lead zero blank
        call    lcd_4bcd        ;send to LCD
        call    del_5ms         ;wait 1 second
        jmp     start           ;and repeat

$include(535lcd.asm)

        END
```

Note: port P1 defaults to high, and it should be terminated to GND to give 0 reading.

## Fault identification

The LCD is a device, which is connected to the microprocessor in a similar way as a ROM. There are data lines, address line (RS, register select), enable line (E), and a read/write line (R/W). The microprocessor writes to the display using these lines. The

lines must therefore toggle for every character sent to the LCD! We can check it with the oscilloscope.

A broken alphanumeric LCD is easy to identify. It is more difficult, when the LCD is not working properly. Possibilities are:

- there is nothing on the LCD:
  - The contrast setting may be improperly set, use the screwdriver and rotate the trimmer for contrast. In our schematics: P1. Rotation of this trimmer should make digits from completely invisible to completely black. Somewhere in the middle the characters should be seen.
  - The LCD may be blank, because the microprocessor does not write anything to it. Check the data, RS, E and R/W lines. They should toggle from time to time.
- the digits of the LCD are completely black:
  - The contrast setting may be wrong, rotate the trimmer for contrast.
  - The LCD is bad, replace it. It is very unlikely, that the microprocessor writes characters, that are completely black!
- The characters at the LCD are readable, but they have no meaning:
  - Most likely the microprocessor sends erroneous messages to the LCD. The reason might be the software, or bad connection of the data line to the LCD
- The characters at the LCD are not readable:
  - Most likely the LCD is bad, since it can not correctly interpret the characters sent from the microprocessor.

# 5. MULTIPLEXED LED DISPLAY

**OBJECTIVE**

Many micro controller based instruments use 7-segment LED display. The purpose of this experiment is to get familiarised with multiplexing technique for such displays. Hardware implementation of multiplexing is also presented.

**REVIEW**

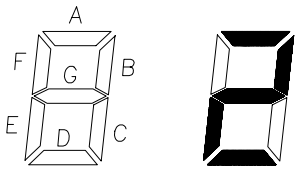



*Figure 1: LED display*

Light emitting diode display (LED display, or seven segment display) consists of seven light emitting diodes (LED) arranged as shown at fig. 1 left. Any number can be shown by turning appropriate LEDs on. A LED is on when a current flows through it (from few mA to max. 25mA). The LED display at fig. 1 right side shows number 2 since LEDs A, B, D, E, and G are on.

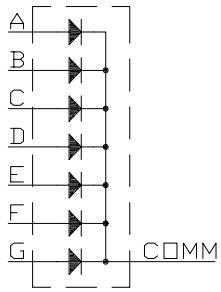



*Figure 2: Diodes in LED display*

One side of all LEDs in array is usually common to save pins on the package. Electrical connection of such array of LEDs with common cathode is given at fig. 2. Displays often have a LED for decimal point, and this gives altogether nine pins at the package.

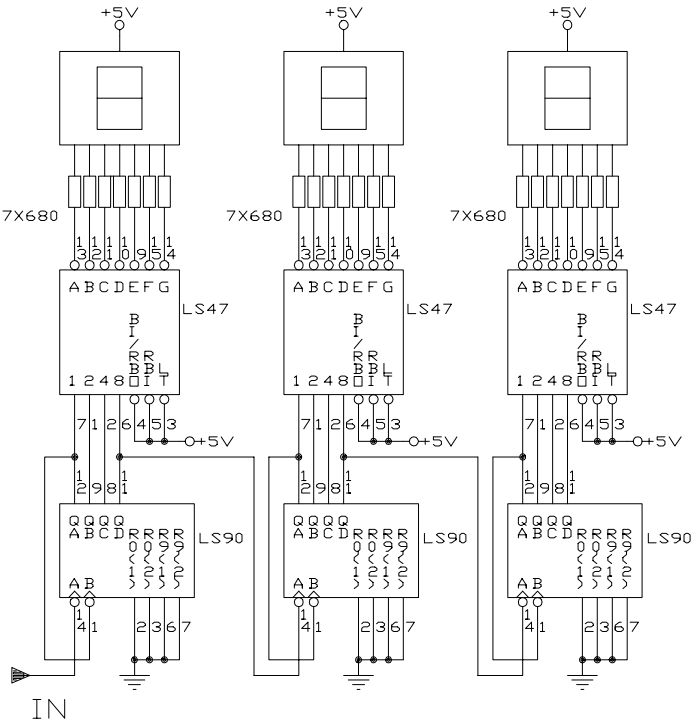



*Figure 3: Three digit BCD counter with LED displays*

Numbers to be shown are given in binary coded decimal (BCD) format. A decoder for conversion from BCD format to seven LEDs is needed. Such decoders are CD4511 in CMOS technology (for common cathode displays), and 74LS47 in LS TTL technology (for common anode displays).

Set of LED displays can be used to show numbers with many digits. Wiring diagram at fig. 3 shows an example of three digit BCD counter.

Note that resistors must be added for every LED, otherwise the intensity of LED depends on the number of LED conducting.

Wiring diagram from fig. 3 could be expanded to many digits by using more counters, LED displays and decoders. However, such approach is wasting many components. For every new digit in a counter, a LED display, a set of resistors and decoder is needed.

The preferred approach uses a multiplexer, one decoder and one set of resistors in series with LEDs in display. Wiring diagram is given at fig.4. Similar segments of all displays are connected in parallel. Displays are driven one by one by shorting to ground only one of the common electrodes of displays at a time. Only one decoder 4511 is used. Multiplexer MUX connects outputs of the appropriate counter to inputs of the decoder. The frequency of switching is typically about 1000Hz. Since human eye is slow the blinking can not be noticed. Fig. 5 gives timing diagram for signals from wiring diagram at fig. 4.

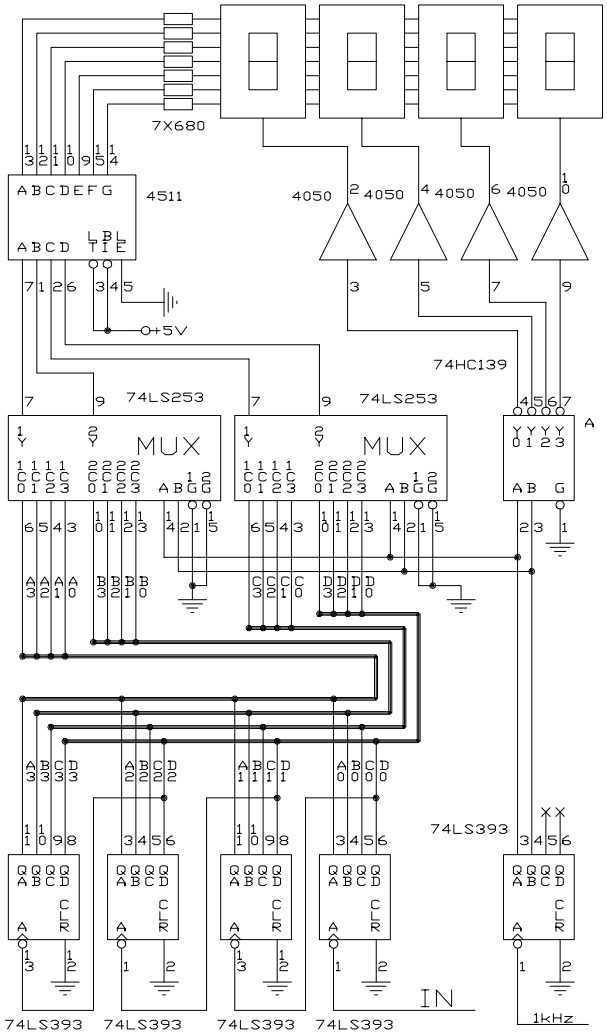



*Figure 4: Multiplexed display 4 digit counter*

Bottom diagram gives the data between multiplexer and seven segment decoder chip 4511. While lines A and B (inputs to decoder HC139) are both low lines between multiplexer and decoder 4511 contain data for the most significant display from leftmost counter LS393, and in the same time through decoder HC139 leftmost display's common electrode is pulled low by buffer 4050. Data from counter comes to the correct display. Note that buffer 4050 can sink up to several tens of mA, and this suffices for display. In the next step lines A and B change, and content of the second counter is applied to decoder, simultaneously second display is on...

Similar approach with multiplexing can be used with micro controller based instruments. Most of the hardware from previous block diagram is implemented in software of microprocessor. One port of microprocessor can be used for outputting data for display. Four bits of one port are used for BCD numbers, and other four bits are used to select one of displays. Outputs for the selection of display can be binary coded; in this case a one-of-many decoder is added externally to select one of many LED displays.

In this experiment only four LED displays are used. Upper four bits of port P4 are used to output BCD numbers, and lower four bits are used as LED display selects. Note

that since only four displays are used external one-of-many decoder for selecting displays is not needed. Four outputs are decoded by software in microprocessor already.

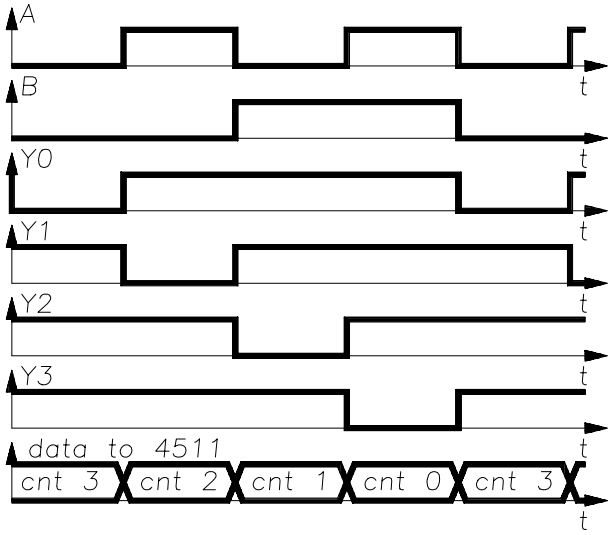



*Figure 5: Timing diagram for multiplexed display*

**EXPERIMENT**

Assemble the circuit as shown at fig. 6. Transistors for driving common cathodes of LED displays can be soldered to microprocessor test board, see stuffing diagram given in foreword to this chapter. Seven segment decoder, resistors and LED displays can be added externally on a test board. Run test program and observe signals at connections to the display.

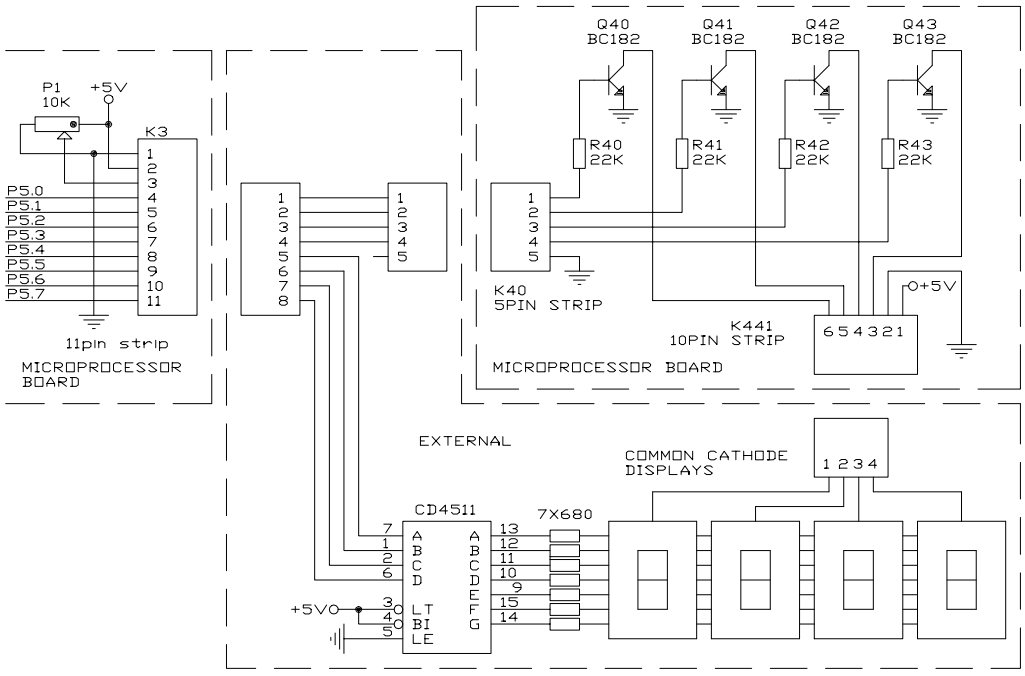



*Figure 6: The connection of LED display to microprocessor system*

The software is given below. It periodically (about once per millisecond) selects another display by setting one of lower four lines of port P4, and simultaneously sends out BCD data for that display through upper four lines of the same port. The data for displaying is taken from ports P1 (units and tens) and P5 (hundreds and thousands). These inputs default to high so appropriate number is set by shorting some of these inputs to ground.

Program listing:

```
$MOD515

        ORG     0H              ;arbitrary start
start:  mov     a,P4            ;read P4
        anl     a,#0Fh          ;lower four bits only

        cjne    a,#8,n_d1       ;pointer to 4. digit ?
        mov     a,#1            ;Y: change to 1. digit
        xch     a,r1            ;tmp: store ptr to R1
        mov     a,P1            ;read input, 1. byte
        jmp     send            ;send it out

n_d1:   cjne    a,#1,n_d2       ;ptr to 1. digit ?
        rl      a               ;Y: change to 2. digit
        xch     a,r1            ;tmp: store ptr to R1
        mov     a,P1            ;read input, 2. byte
        swap    a               ;
        jmp     send            ;send it out

n_d2:   cjne    a,#2,n_d3       ;ptr to 2. digit ?
        rl      a               ;Y: change to 3. digit
        xch     a,r1            ;tmp: store ptr to R1
        mov     a,P5            ;read input, 3. byte
        jmp     send            ;send it out

n_d3:   rl      a               ;change to 4. digit
        xch     a,r1            ;tmp: store ptr to R1
        mov     a,P5            ;read input, 4. byte
        swap    a               ;

send:   anl     a,#0fH          ;send out; take upper
        swap    a               ;nibble
        orl     a,r1            ;join it with ptr
        mov     P4,a            ;to port P4
        call    dl_1ms          ;wait 1 milli second
        jmp     start           ;and repeat


dl_1ms: mov     r1,#10          ;this is delay routine
o_loop: mov     r2,#49
i_loop: djnz    r2,i_loop
```

```
        djnz    r1,o_loop
        ret

    END
```

## Fault identification

By observing the figure 6 one can see, that the segment of the individual displays are organized in a grid. Vertical connections (common cathodes of individual displays) take care for each individual display, while horizontal connections (common anodes) take care for each individual segment.

- If therefore, a fault appears in Q41 for instance, the leftmost display will work correctly, and so will the two rightmost. The second display will be blank.
- If one of the lines from the segment decoder faults, then all displays will be affected: the same segment on all displays will fail.
- If only one segment at one display is faulty, then the driving might be good, but the display needs a replacement.

# 6. MICROPROCESSOR BASED COUNTER

**OBJECTIVE**

Some micro controllers for embedded applications have counters built in. Those counters can be used either for event counting or timing. The purpose of this experiment is to demonstrate a simple microprocessor based counter.

**REVIEW**

Microprocessor from 8032 family has three 16 bit counters/timers built in (T0, T1 and T2). Counters can be used in different modes: to count external pulses, to count external pulses gated with external signal, or as timers. The mode of operation is defined by appropriate flags in microprocessor.

Timer/counter T0 will be used to count external pulses. Its behaviour depends on five flags located in two special function registers (SFR) named TMOD and TCON.

Four flags define what the counter will count. Those are located in SFR named ***TMOD***. Following table summarises the location of flags in SFR TMOD:

| 89 hex | GATE | C/T | M1 | M0 | GATE | C/T | M1 | M0 |
|---|---|---|---|---|---|---|---|---|
| | timer/counter 1 | | | | timer/counter 0 | | | |

Lower four bits of SFR TMOD are used for timer/counter T0 as summarised in the table:

| bit | | Action |
|---|---|---|
| GATE | | Gating control<br>When set, timer/counter 0 is enabled only while “INT0” pin is high and “TR0” control bit is set<br>When cleared timer 0 is enabled whenever “TR0” control bit is set |
| C/T | | Counter/timer select bit<br>Set for counter operation (input from “T0” input pin)<br>Cleared for timer operation (input from internal system clock) |
| M1 | M0 | |
| 0 | 0 | 8 bit timer/counter<br>“TH0” operates as 8 bit timer/counter, “TL0” operates as 5 bit prescaler |
| 0 | 1 | 16 bit timer/counter<br>“TH0” and “TL0” are cascaded; there is no prescaler |

| | | |
|---|---|---|
| 1 | 0 | 8 bit auto reload timer/counter<br>"TH0" holds a value which is to be reloaded into "TL0" each time it overflows |
| 1 | 1 | Timer 0:<br>"TL0 is an 8 bit timer/counter controlled by standard timer 0 control bits. TH0 is an 8 bit timer only controlled by timer 1 control bits. |
| 1 | 1 | Timer 1:<br>Timer 1 stops |

Similarly upper four flags of the same SFR define timer/counter T1.

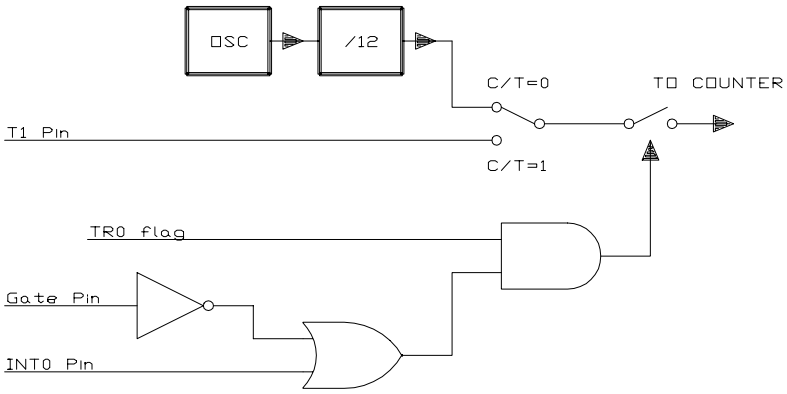



*Figure 1: Logic diagram representing the input to timer/counter T0*

The fifth flag enables timer/counter T0, and is located in SFR named TCON. Complete layout of flags in SFR TCON is given below:

| 8F hex | 8E hex | 8D hex | 8C hex | 8B hex | 8A hex | 89 hex | 88 hex |
|---|---|---|---|---|---|---|---|
| TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |

Counting is enabled by setting flag TR0. Flag TF0 is set by the counter when it overflows. Similarly flags TF1 and TR1 are responsible for counter/timer T1.

A wiring diagram representing the operation of flags is shown at fig.1.

Flag TR0 is used to enable counting. To use timer/counter 0 as an event counter for external pulses flag GATE should be cleared, flag C/T set, and flags M1 and M0 should have value of 01 (16 bit counter).

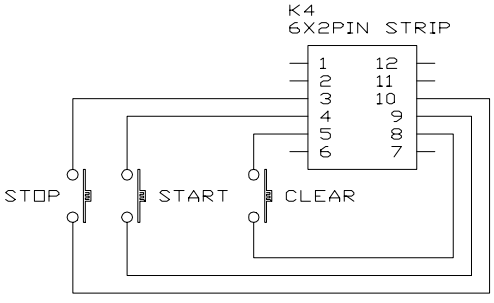



*Figure 2: The connection of switches*

Three switches are used to control microprocessor based counter. Those are CLEAR, START and STOP. Wiring diagram of switches for controlling counter is given at fig.2. When switch START is momentarily closed the counter is enabled, and it is disabled when switch STOP is momentarily closed. When switch CLEAR is closed the content of the counter is zeroed.

Input signal to counter should be TTL compatible and connected to line T0 (connector K7, pin 6).

Program for counter is written in assembly language. Flags are initialised as stated above at the beginning of program; after this microprocessor enters a program loop. Once in the loop it reads switches and sets flags and content of the counter accordingly. Finally it reads the content of counter/timer T0 and copies it onto ports P1 and P4. Additionally, it converts the content of counter into BCD format and displays it on LCD.

**EXPERIMENT**

Assemble the circuit of a counter. Burn EPROM with prepared program and test the circuit. What is the shortest pulse, that is still registered by the counter? What is the maximum input frequency? (Hint: if pulses are random, but short, they can be extended by a univibrator. Two univibrators are available at microprocessor board, see wiring diagram at fig. 3 and stuffing diagram in experiment μP.0).

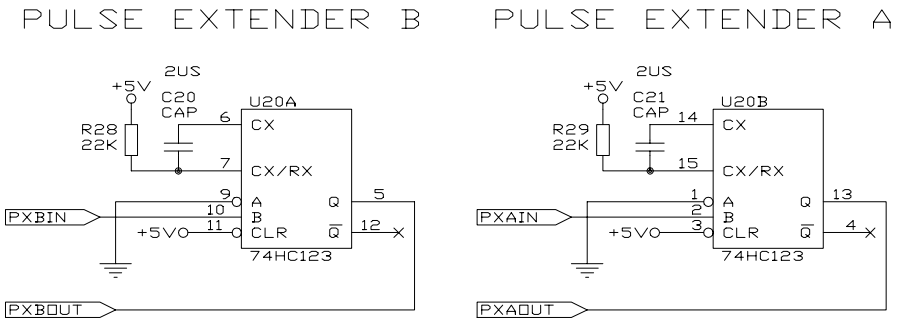



*Figure 3: One of the two univibrators can be used to extend pulses*

```
$MOD515
        ORG     0H              ;arbitrary start
        call    lcd_ini         ;initialise LCD     **
        call    lcd_coff        ;hide cursor        **
        mov     tl0,#0          ;init counter T0, low
        mov     th0,#0          ;init counter T0, high
        mov     tcon,#0         ;init TCON
        mov     tmod,#00000101b ;init TMOD

start:  mov     a,0DBh          ;RD switches, port AN
        jb      acc.7,n_clr     ;switch clear closed ?
        mov     tl0,#0          ;Y: clear counter T0
        mov     th0,#0          ;
n_clr:  jb      acc.6,n_sta     ;switch start closed ?
        setb    tr0             ;Y: enable counting
n_sta:  jb      acc.5,n_sto     ;switch stop closed ?
        clr     tr0             ;Y: disable counting
n_sto:  mov     p1,tl0          ;copy T0 to P1
        mov     p4,th0          ;and P4
        mov     r4,tl0          ;set call bin2bcd **
```

```
        mov     r3,th0          ;
        call    bin2bcd         ;and call bin2bcd **
        mov     r2,#4           ;set call display **
        clr     LZB             ;lead. 0 blank ON **
        call    lcd_4bcd        ;display at LCD   **
        call    del_5ms         ;wait 5 mili seconds
        jmp     start           ;and repeat

$include(535lcd.asm)            ;include LCD routines
        end
```

Note: the line with ** can be omitted if LCD module is not attached. The result from counter T0 is also available at ports P1 (hi) and P4 (lo). Routines for writing to LCD module will be available for use, they are included in this program by statement $include.

## Fault identification

There is not much to check and repair with this counter, if it is not counting but the LCD display is working and providing the program is stored in the ROM correctly. The counter migt not count because there are no pulses coming at the appropriate input, or if the counter is not started. But if these things are properly set, and the counter is still not counting, the reason might be a faulty microprocessor. Try replacing it.

In real life the counter consists of a microprocessor and several stages before it. These might cause additional troubles. In real life always check first, if the pulses to be counted, are coming to the pin of the microprocessor.

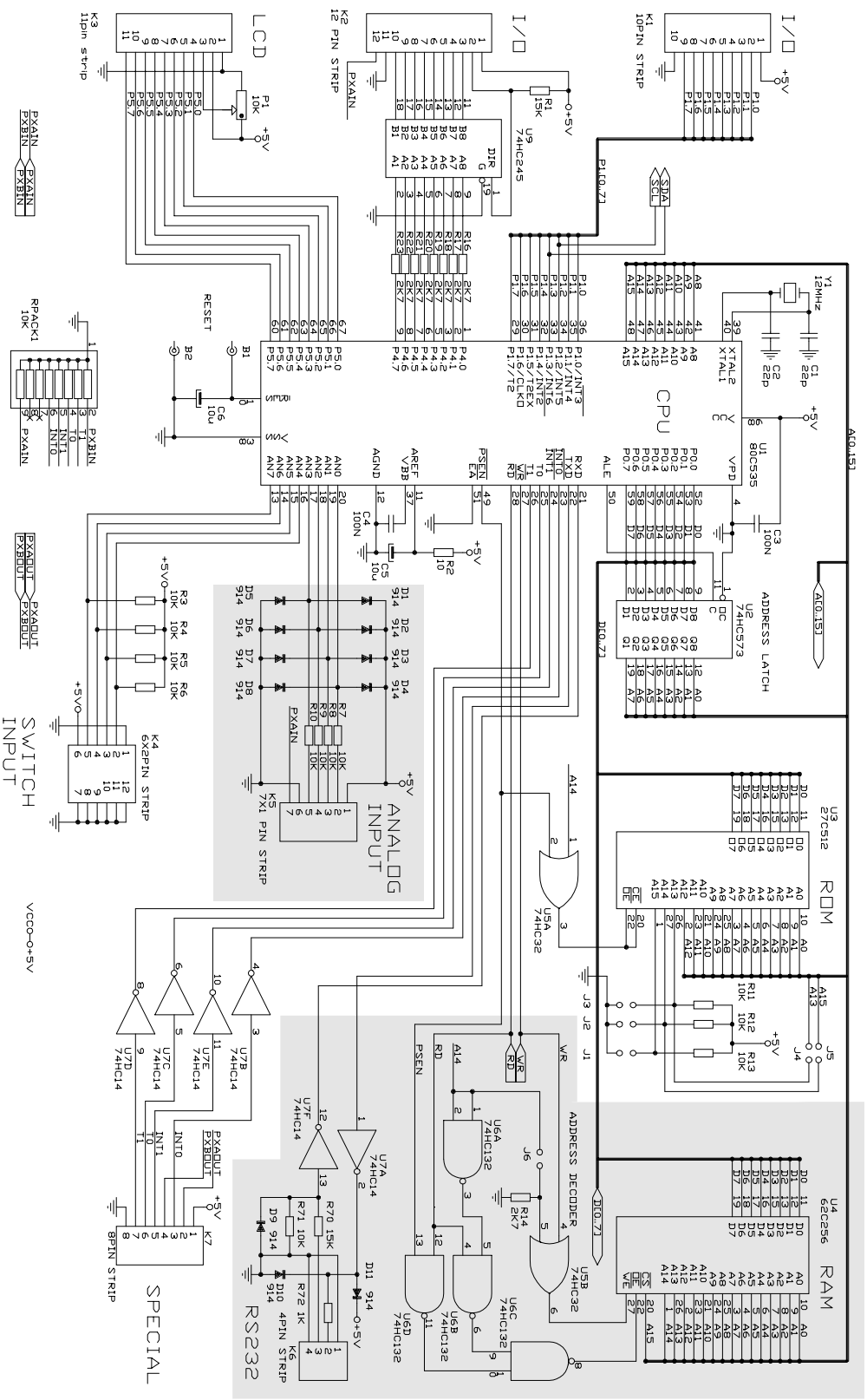



*Figure 1: Highlighted parts are used in microprocessor based counter*

# 7. RS 233 CONNECTION

**OBJECTIVE**

The RS232 line can be used to transfer data between personal computer and microprocessor based instrument. The purpose of this experiment is to demonstrate the use of RS232 connection, to practice determining of the RS232 transmission parameters, and use of terminal program on PC.

**REVIEW**

The RS232 connection in widely used for transferring data between microcomputer based instruments and PCs. Basically three wires are used to transfer data between units A and B: first wire carries data from unit A to unit B, second wire carries data from unit B to A, and third wire connects grounds of both units together. Connections to units have names as stated in fig. 1. Signal output with name transmit data (TXD) from one unit must be connected to signal input called receive data (RXD) of the other unit.

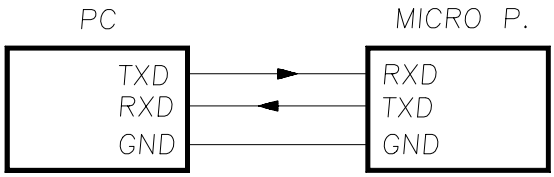



*Figure 1: RS232 connection*

Signals on these two wires can have one of two possible values: logic high is represented by voltage between -5V and -15V, and logic low is represented by voltage between +5V and +15V. These voltages can be measured at RS232 output from the PC. In practice PC recognises any input voltage above about 1V as logic low, the rest is treated as logic high. It transmits high and low as stated before.

Data is sent out of a unit as a string of bits. Usually a string is eight bits (a byte) long, but other lengths are also possible. Logic level for every bit is present for some time; faster the transmit speed less time one bit is available. Transfer speed can be for instance 2400 bits per second (2400 Baud); this means that one bit is present on a wire for 1/2400s, or about 0.4ms. Typical transfer speeds are 300 Baud, 1200 Baud, 2400 Baud, 4800 Baud, 9600 Baud…

A string of bits has a head and a tail. The head is called start bit. It has the same duration as any other bit and is logic low. The tail is composed of parity bit and stop bit(s). Parity bit is sometimes added to ensure correct data transmission. It is one bit in duration, and its value depends on other bits in a string. Stop bit(s) signalise the end of string; it can be from one bit to two bits, and is logic high. Typical sequence of bits at TXD is given at fig.2. Note that start bit is always represented

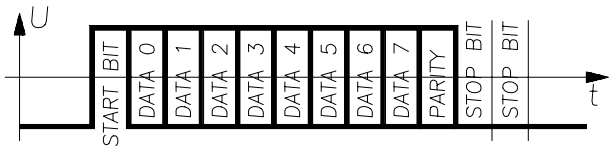



*Figure 2: Signal at TXD line*

by positive voltage, and stop bit(s) by negative voltage.

Transfer speed, number of bits in a string, parity, and number of stop bits are called communication parameters, and this can usually be set at unit. Both units must use the same communication parameters otherwise data is not transferred correctly.

Microprocessor system will be used to test RS232 communication. Prepared programs will be run, and serial connection between PC and microprocessor system will be established. Fig. 3 gives details on microprocessor system used.

The microprocessor 80C535 includes complete serial port for receiving and transmitting bytes through RS232 line. Serial port is full duplex; it can simultaneously transmit and receive data. The byte received is stored into a buffer, and reception of second byte can start even before the first has been read out of the buffer. Serial channel at 80C535 is compatible with the one in 8031 family of microprocessors.

*Details on programming the 80C535.*

A set of flags determine the behaviour of serial port. These flags are located at four special function registers: SCON, SBUF, PCON and ADCON. Baud rate can be either fixed or derived from Timer 1.

The layout of SFR named SCON is given below.

| ***SCON*** | 9F hex | 9E hex | 9D hex | 9C hex | 9B hex | 9A hex | 99 hex | 98 hex |
|---|---|---|---|---|---|---|---|---|
| 98 hex | SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI |

The content of flags SM1 and SM0 define the mode of serial interface:

| SM0 | SM1 | mode |
|---|---|---|
| 0 | 0 | serial mode 0: shift register mode, fixed baud rate |
| 0 | 1 | serial mode 1: 8-bit UART, variable baud rate |
| 1 | 0 | serial mode 2: 9-bit UART, fixed baud rate |
| 1 | 1 | serial mode 3: 9-bit UART, variable baud rate |

Serial mode 1 is used here, therefore flag SM0 is cleared and flag SM1 is set.

Flag SM2 enables special feature of 8032 series microprocessors, a multiprocessor communication facility; this feature is not used here so flag is cleared.

Flag RI is set by hardware when a byte has been received through serial channel. It must be cleared by software after using the received byte.

Register ***SBUF*** has double function: it contains the byte received through serial channel, and a byte is transmitted through serial port after being written into this register.

Flag located at D7 of SFR ***PCON*** is named SMOD. When set baud rate is doubled. For the experiment this flag is cleared.

Flag located at D7 of SFR ***ADCON*** is named BD. When set, baud rate is fixed to 4800 Baud (9600 Baud if flag SMOD if set). When cleared, baud rate if defined by Timer 1. The following table gives proper setting for the desired Baud rate with Timer 1:

| Baud rate | fosc[MHz] | Timer 1 | | |
|---|---|---|---|---|
| | | C/T | Mode | Reload value |
| Mode 1,3 9600 Baud | 11.059 | 0 | 2 | FD hex |

| | | | | |
|---|---|---|---|---|
| 4800 Baud | 11.059 | 0 | 2 | FA hex |
| 2400 Baud | 11.059 | 0 | 2 | F4 hex |
| 1200 Baud | 11.059 | 0 | 2 | E8 hex |
| 110 Baud | 12.0 | 0 | 1 | FEEB hex |

This flag is cleared in out experiment, since Timer 1 is used to define the Baud rate. Note that quartz crystal must have frequency of 11.059MHz for variable baud rate.

Microprocessor unit will be used to transmit a single byte periodically. Microprocessor reads byte from port P1, and sends it out through RS232 line every millisecond. Transmit speed is defined by logic levels at switch inputs AN4 and AN5:

| input AN5 | input AN6 | speed |
|---|---|---|
| 0 | 0 | 1200 Baud |
| 0 | 1 | 2400 Baud |
| 1 | 0 | 4800 Baud |
| 1 | 1 | 9600 Baud |

Those switches are read only once after the reset of microprocessor.

Observe signals at the output TXD of microprocessor board. Can you recognise the pattern sent as byte from port P1? What is the transmission speed? Connect microprocessor board to PC and run terminal program to visualise received characters. The connection of RS232 cable is given at fig.4.

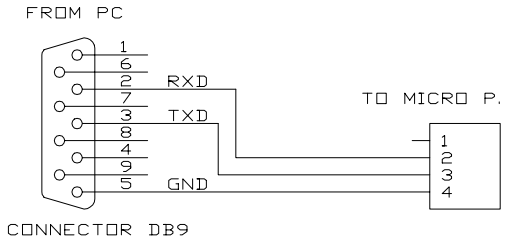



*Figure 3: The RS232 cable*

```
$MOD515
        ORG     0               ;start at 0hex
        call    lcd_ini         ;lcd: initialise     *
        call    lcd_coff        ;lcd: cursor off     *
        mov     a,0dbh          ;read switches

        clr     SM0             ;select serial mode 1
        setb    SM1             ;
        clr     SM2             ;
        clr     TI              ;clear TI flag
        anl     PCON,#01111111b
        mov     TMOD,#00100000b ;T1 defines baud rate

        anl     a,#01100000b    ;use only AN5 and AN6
```

```
        mov     P4,a
        mov     tl1,#0FDh       ;9600 Bd by defoult
        mov     th1,#0FDh       ;
        cjne    a,#0,n1200      ;1200 Bd ?
        mov     tl1,#0E8h       ;yes -> reload for T1
        mov     th1,#0E8h       ;
n1200:  cjne    a,#20h,n2400    ;2400 Bd ?
        mov     tl1,#0F4h       ;yes -> reload for T1
        mov     th1,#0F4h       ;
n2400:  cjne    a,#40h,n4800    ;4800 Bd ?
        mov     tl1,#0FAh       ;yes -> reload for T1
        mov     th1,#0FAh       ;
n4800:  setb    TR1             ;enable timer T1

loop:   mov     a,P1            ;read char from P1
        mov     SBUF,a          ;send it through RS232
        mov     r1,a            ;prepare write LCD **
mov     r2,#5           ;                       **
        call    lcd_charpos1    ;write on LCD      **
w_td:   jnb     TI,w_td         ;wait until sent
        clr     TI              ;acknowledge sending
        call    del_15ms        ;wait 15ms
        jmp     loop            ;and repeat

$include(5351cd.asm)
        end
```

Microprocessor receives a byte through RS232 line, and writes it to port P1. The same byte is echoed back through RS232 line. Transfer speed is defined by logic levels at switch inputs AN4 and AN5 as specified above. A terminal program is used at PC to send bytes and receive echoed bytes. Check maximum length of the line between PC and microprocessor system for different transmit speeds.

```
$MOD515
        ORG     0               ;start at 0hex
        call    lcd_ini         ;lcd: initialise     *
        call    lcd_coff        ;lcd: cursor off     *

        clr     SM0             ;select serial mode 1
        setb    SM1             ;
        clr     SM2             ;
        clr     TI              ;clear TI flag
        anl     PCON,#01111111b
        mov     TMOD,#00100000b ;T1 defines baud rate

        mov     a,0dbh          ;read switches
        anl     a,#01100000b    ;use only AN5 and AN6
        mov     P4,a
        mov     tl1,#0FDh       ;9600 Bd by defoult
```

```
        mov     th1,#0FDh       ;
        cjne    a,#0,n1200      ;1200 Bd ?
        mov     tl1,#0E8h       ;yes -> reload for T1
        mov     th1,#0E8h       ;
n1200:  cjne    a,#20h,n2400    ;2400 Bd ?
        mov     tl1,#0F4h       ;yes -> reload for T1
        mov     th1,#0F4h       ;
n2400:  cjne    a,#40h,n4800    ;4800 Bd ?
        mov     tl1,#0FAh       ;yes -> reload for T1
        mov     th1,#0FAh       ;
n4800:  setb    TR1             ;enable timer T1
        clr     RI              ;clear receive flag
        setb    REN             ;enable serial receive

w_rd:   jnb     RI,w_rd         ;wait for character
        clr     RI              ;clear receive flag
        mov     r1,SBUF         ;prepare write on LCD*
        mov     r2,#5           ;                    *
        call    lcd_charpos1    ;write on LCD        *
        mov     SBUF,r1         ;send character back
w_td:   jnb     TI,w_td         ;wait until sent
        clr     TI              ;acknowledge sending
        call    del_15ms        ;wait 15ms
        jmp     w_rd            ;and repeat

$include(535lcd.asm)
   end
```

## Fault identification

How do you find the correct lines to be connected together? Which pin is TD, and which is RD on a unit? The simplest way is to use the oscilloscope, or at least a multimeter. The TD line from a PC is always at approx. –12V, when characters are not sent out of the PC. Check pins 2 and 3 at RS232 connector of the PC. One of them is –12V, and this is the transmit data TD. The same is truth for other units, like printers, modems, and alike. The pin with the negative voltage is the TD. Be sure to connect the pin with –12V from the PC with the pin (pin 2 or 3), which is not –12V from the other unit, and you have made the proper connection.

At the PC side we need some software for communication. This software transmits every character pressed on the keyboard, to the COMM port of the PC. If we plug our unit to COM2, then the software should be set to communicate through COM2. Instead of figuring it out through software settings of the PC, we will rather use the oscilloscope, observe the voltage at TD pin of the connector selected, and press a key on the keyboard. If the setting is correct, the we will see some toggling of the observed line from –12V to +12V every time a key is pressed. If we see nothing, then we do not observe the correct COMM port; characters are coming out of the PC through another connector.

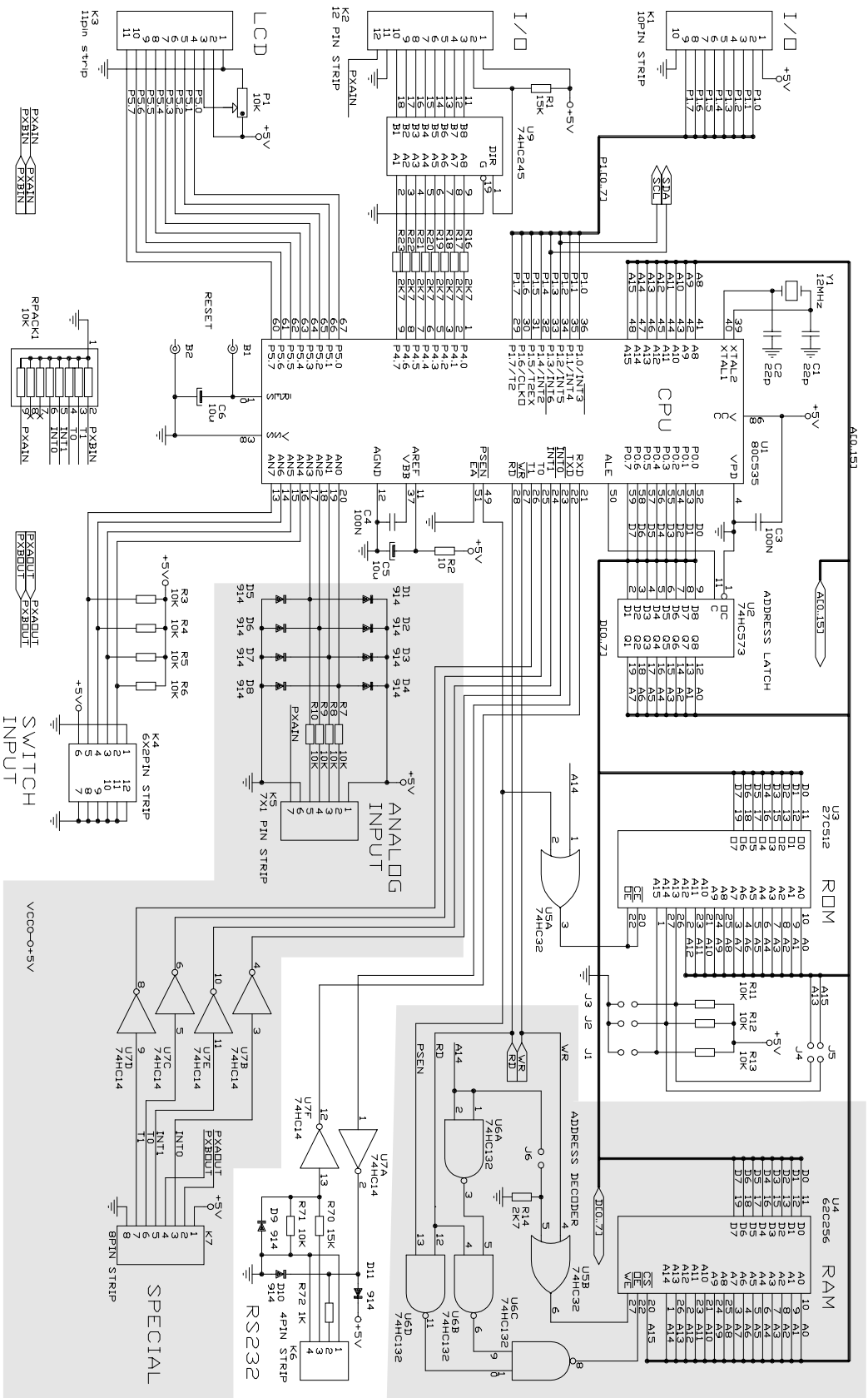



*Figure 4: Wiring diagram of microprocessor system*

Our microprocessor unit does not support the full ±12V range, as specified for the RS232. It relies on the fact, that the PC interprets 0V at the RS232 RD line the same way as –12V, and that it interprets +5V at this pin the same way as +12V. This simplifies out interfacing at the microprocessor board, but is not considered a professional solution. To do it properly, we should use one of the RS232 driver chips, like MAX232.

Our design, however, is safe even against improper connection of the unit to the PC. If the TD and RD lines are not connected properly, the connection will not work, but no chips will be destroyed.

The micro controller unit can communicate with the PC at different speeds and settings. Some of them are mentioned above. Please note that the speed and other communication parameters must be set the same way at the PC and at the microprocessor unit, otherwise the communication will not be successful.

Please note the table with appropriate reload values for the timer T1. It says that we **must use the quartz crystal of 11.059MHz** in order to obtain baud rates of 9600, 4800 and 2400 Baud. With the 12MHz crystal this is not possible! Resolder a suitable quartz crystal into the PCB. These two quartz crystal frequencies seem similar, but the timer T1, which is used to derive a clock signal for the serial communication circuit inside the microprocessor, can not make the correct frequency from the 12MHz crystal. The frequency is about 5% wrong, and this is enough, that the communication is not possible. This odd resonant frequency of the quartz crystal has some consequences. With the 12MHz crystal the timing made with the microprocessor is accurate, since each clock tick takes exactly one μs. With 11.059MHz crystal the former μs becomes a little longer. Timing circuits made with the microprocessor must be adjusted to compensate for the changed crystal. In some cases this requires very strange coding, and the timing can not even be successful.

Some recent micro controllers avoid this problem by another, specially developed, timer, which is included in the chip as a baud rate generator. This can derive a suitable clock frequency for the serial communication even from the 12MHz crystal. Such timer is included in the C515 microprocessor, for instance.

# 8. MICROPROCESSOR WITH BUILT IN ADC

**OBJECTIVE**

Modern microprocessors for embedded applications have ADCs built in. The purpose of this experiment is to get familiarised with such ADC.

**REVIEW**

Microprocessor 80C535 has an 8 bit ADC built in. Its conversion time is about 13μs. The ADC has eight multiplexed inputs AN0 to AN7. Four of them (AN0 to AN3) can be used on the microprocessor system presented. See fig. 1 for wiring diagram.

### *Details on programming ADC in SAB80C535*

The behaviour of the built-in ADC is determined by flags from three SFRs: ADCON, DAPR in ADDAT.

The layout of register ***ADCON*** is given below. Five flags from this register are used.

| | 0DF h | 0DE h | 0DD h | 0DC h | 0DB h | 0DA h | 0D9 h | 0D8 h |
|---|---|---|---|---|---|---|---|---|
| 0D8 h | BD | CLK | - | BSY | ADM | MX2 | MX1 | MX0 |

Flags MX2 to MX0 are used to select one of eight inputs to ADC. An example: to select input 2, flag MX1 must be set, and flags MX0 and MX2 must be cleared. Input 0 is used in experiment, therefore all three flags are cleared.

Flag ADM defines conversion mode of ADC; when cleared a converter is stopped after a conversion, and has to be restarted for next conversion. When this flag is set the converter repeats conversions continuously. Flag ADM is cleared in for experiment.

Flag BSY is set during conversion. This flag is cleared by hardware when conversion is finished.

Conversion is started by writing a value to register ***DAPR***. The value written specifies input voltage range. The range is determined by lower limit and upper limit. Both limits are coded by four bits as follows. Upper four bits of DAPR are for the upper limit, and lower four bits for the lower limit.

| Step | code | lower limit | upper limit |
|---|---|---|---|
| 0 | 0000 | 0.0 | 5.0 |
| 1 | 0001 | 0.3125 | - |
| 2 | 0010 | 0.625 | - |
| 3 | 0011 | 0.9375 | - |

| | | | |
|---|---|---|---|
| 4 | 0100 | 1.25 | 1.25 |
| 5 | 0101 | 1.5625 | 1.5625 |
| 6 | 0110 | 1.875 | 1.875 |
| 7 | 0111 | 2.1875 | 2.1875 |
| 8 | 1000 | 2.5 | 2.5 |
| 9 | 1001 | 2.8125 | 2.8125 |
| 10 | 1010 | 3.125 | 3.125 |
| 11 | 1011 | 3.4375 | 3.4375 |
| 12 | 1100 | 3.75 | 3.75 |
| 13 | 1101 | - | 4.0625 |
| 14 | 1110 | - | 4.375 |
| 15 | 1111 | - | 4.68754 |

Thus writing value 0 to DAPR specifies input voltage range of 0V to 5V.

The result of conversion is available in register ***ADDAT*** after flag BSY is cleared by hardware.

**EXPERIMENT**

Run prepared program. It initiates conversions and copies results to port P1. Simultaneously the result is shown at LCD module. Observe the resolution of the ADC.

The program:

```
$MOD515

        ORG     0                   ;start at 0
        call    lcd_ini             ;init LCD          *
        call    lcd_coff            ;cursor off        *
        mov     dptr,#mystr         ;display aux text  *
        mov     r2,#3h              ;at position 3     *
        call    lcd_string          ;display it        *

mai:    call    del_15ms            ;wait 15ms
        anl     adcon,#11111000b    ;select IN0
        mov     dapr,#0             ;set 0-5V range
bsy1:   jb      0dch,bsy1           ;still busy ?
        mov     P1,0d9h             ;N: copy to P1
        mov     r4,0d9h             ;result of conver.
        mov     r5,#0               ;prepare bin2bcd   *
        call    bin2bcd             ;and call bin2bcd  *
        mov     r2,#7               ;prepare for write *
        clr     LZB                 ;no leading zeros  *
        call    lcd_4bcd            ;display 4 digits  *
        jmp     mai                 ;and repeat
```

```
mystr:  db      'U1:',0             ;                   *
$include(5351cd.asm)

        end
```

Lines with asterix can be omitted if LCD module is not used. In this case the result of conversion is available at port P1.

Note that inputs to the ADC are protected against over voltage by diodes and resistors. However, do not abuse this protection by applying high voltage at the input.

## Fault identification

Take care not to apply a voltage, which is below 0V or above +5V at any input of the microprocessor, since this may damage the microprocessor. At the board these inputs of the microprocessor are protected with resistors and diodes, but it is better to be careful.

If the read-out from the ADC is stucked to either 255 or 0, then the problem might be in the wrong range. This ADC can work either from 0V to 5V, or it can be sub ranged, see the first table in this chapter. Check the programme in the ROM.

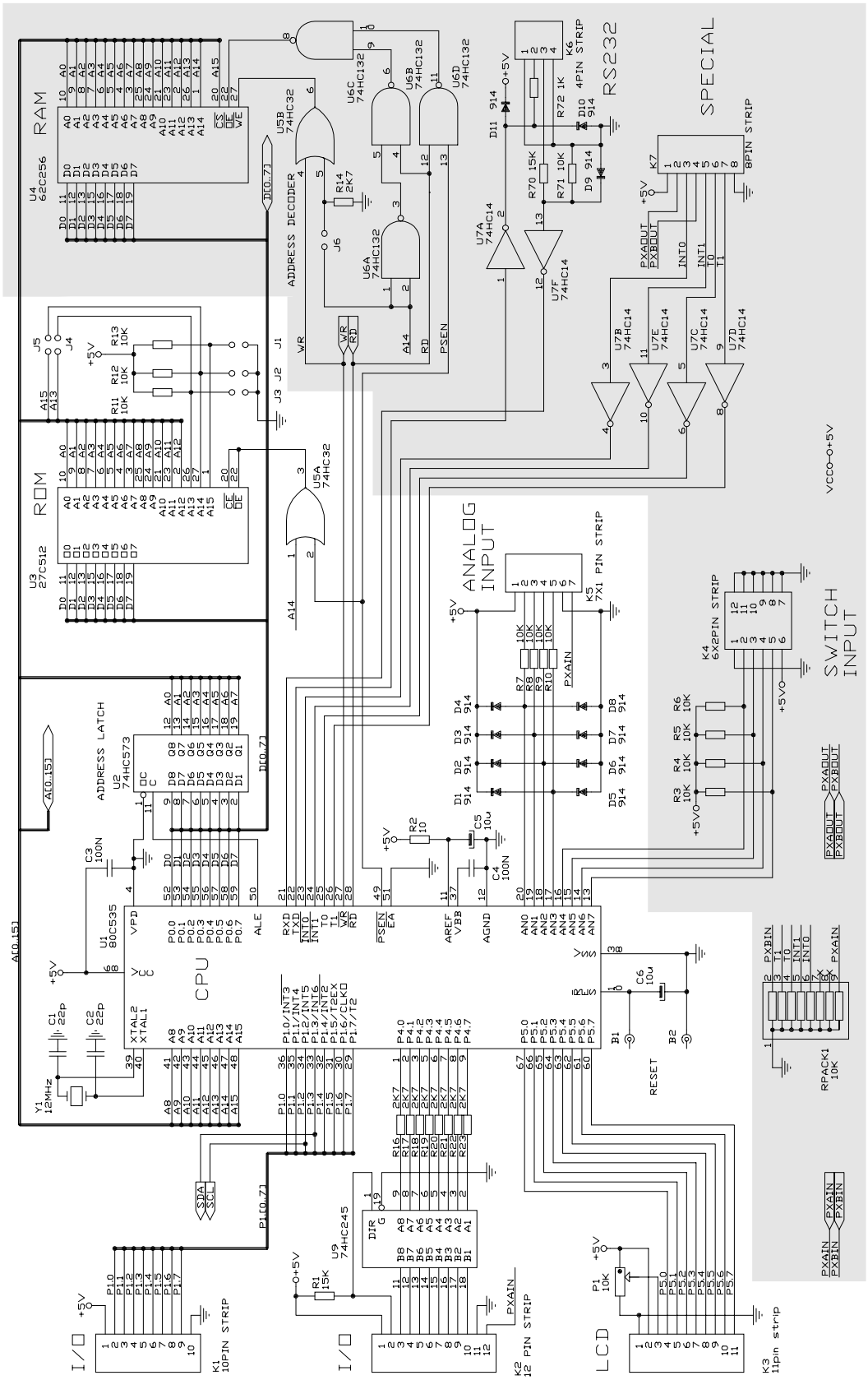



*Figure 2: The Microprocessor system*

# 9. MICROPROCESSOR INTERRUPTS

**OBJECTIVE**

The objective of this experiment is to get familiarised by microprocessor interrupts.

**REVIEW**

Usually a microprocessor is running program instruction by instruction as written in EPROM. But there are times when it must react quickly because something special has happened. This something might be a power fail, or only a pulse that should be counted. Microprocessor must stop executing normal program and service the unexpected event immediately. This action of breaking the normal program and servicing the unexpected event is called an interrupt. After the interrupt microprocessor jumps to a prepared part of the program called interrupt service routine. This part holds instructions to be executed after interrupt. Once microprocessor finishes executing these instructions it jumps back to the same place it was before the interrupt.

In the experiment presented the interrupt input of microprocessor is used for counting pulses. Microprocessor is mainly executing its program; it writes successive numbers about every 15ms to port P1 (and also to the LCD screen, first line). On external interrupt microprocessor increments the number at port P4 (and also to the LCD screen, second line).

Microprocessor SAB80C535 has many external interrupt inputs; interrupt input INT0 is used here. This input is buffered and available at connector K7, pin 3, see wiring diagram at fig.1. An interrupt is caused by bringing input INT0 high. input must stay high for at least 12 clock cycles (1μs), before it is recognised as an interrupt.

*Details on programming interrupts with SAB 80C535*

The response to an interrupt is defined by set of flags in four SFR of microprocessor: TCON, IEN0, IP0 in IP1.

The layout of the SFR named ***TCON*** is given below.

| TCON | 8F h | 8E h | 8D h | 8C h | 8B h | 8A h | 89 h | 88 h |
|---|---|---|---|---|---|---|---|---|
| 88 hex | TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |

An interrupt can be started by high to low transition at pin of microprocessor, or it can be started by logic level low at the same pin. Microprocessor interrupt input is transition sensitive when flag IT0 is set otherwise it is state sensitive. Transition sensitive option is used so flag IT0 is set.

Flag IE0 is set when interrupt is received by microprocessor, and cleared when interrupt is processed (once instruction IRET has been executed).

The layout of SFR named ***IEN0*** is given below.

| IEN0 | 0AF h | 0AE h | 0AD h | 0AC h | 0AB h | 0AA h | 0A9 h | 0A8 h |
|---|---|---|---|---|---|---|---|---|
| 0A8 h | EAL | WDT | T2 | ES | ET1 | EX1 | ET0 | EX0 |

Microprocessor will respond to interrupts at those lines only which are enabled. Setting flag EX0 enables external interrupt line INT0.

Flag EAL is master enable for interrupts. When set all enabled interrupts can break the execution of a program.

Many interrupts can happen simultaneously. The interrupt with the highest priority is served first. The priority for an interrupt is defined by flags in SFRs named ***IP0*** and ***IP1***. Their layout of both is given below.

IP0

| 0A9 h | - | WDTS | IP0.5 | IP0.4 | IP0.3 | IP0.2 | IP0.1 | IP0.0 |
|---|---|---|---|---|---|---|---|---|

IP1

| 0B9 h | - | - | IP1.5 | IP1.4 | IP1.3 | IP1.2 | IP1.1 | IP1.0 |
|---|---|---|---|---|---|---|---|---|

Flags IP0.0 and IP1.0 define priority for interrupt input INT0:

| IP1.0 | IP0.0 | Priority |
|---|---|---|
| 0 | 0 | 0 - lowest |
| 0 | 1 | 1 |
| 1 | 0 | 2 |
| 1 | 1 | 3 - highest |

The highest priority is used, so both flags are set for this experiment.

Different interrupts are possible on microprocessor SAB80C535. An interrupt service routine for each interrupt must start at following address:

| Interrupt flag | Address of the servicing routine | Interrupt source |
|---|---|---|
| IE0 | 0003 hex | external interrupt INT0 |
| TF0 | 000B hex | Counter/timer T0 |
| IE1 | 0013 hex | external interrupt INT1 |
| TF1 | 001B hex | Counter/timer T1 |
| RI/TI | 0023 hex | RS232 |
| TF2/EXF2 | 002B hex | Counter/timer T2 |
| IADC | 0043 hex | ADC |

**EXPERIMENT**

Run prepared program. Observe signals on microprocessor bus. How frequent can interrupts be?

Software for this experiment is given below.

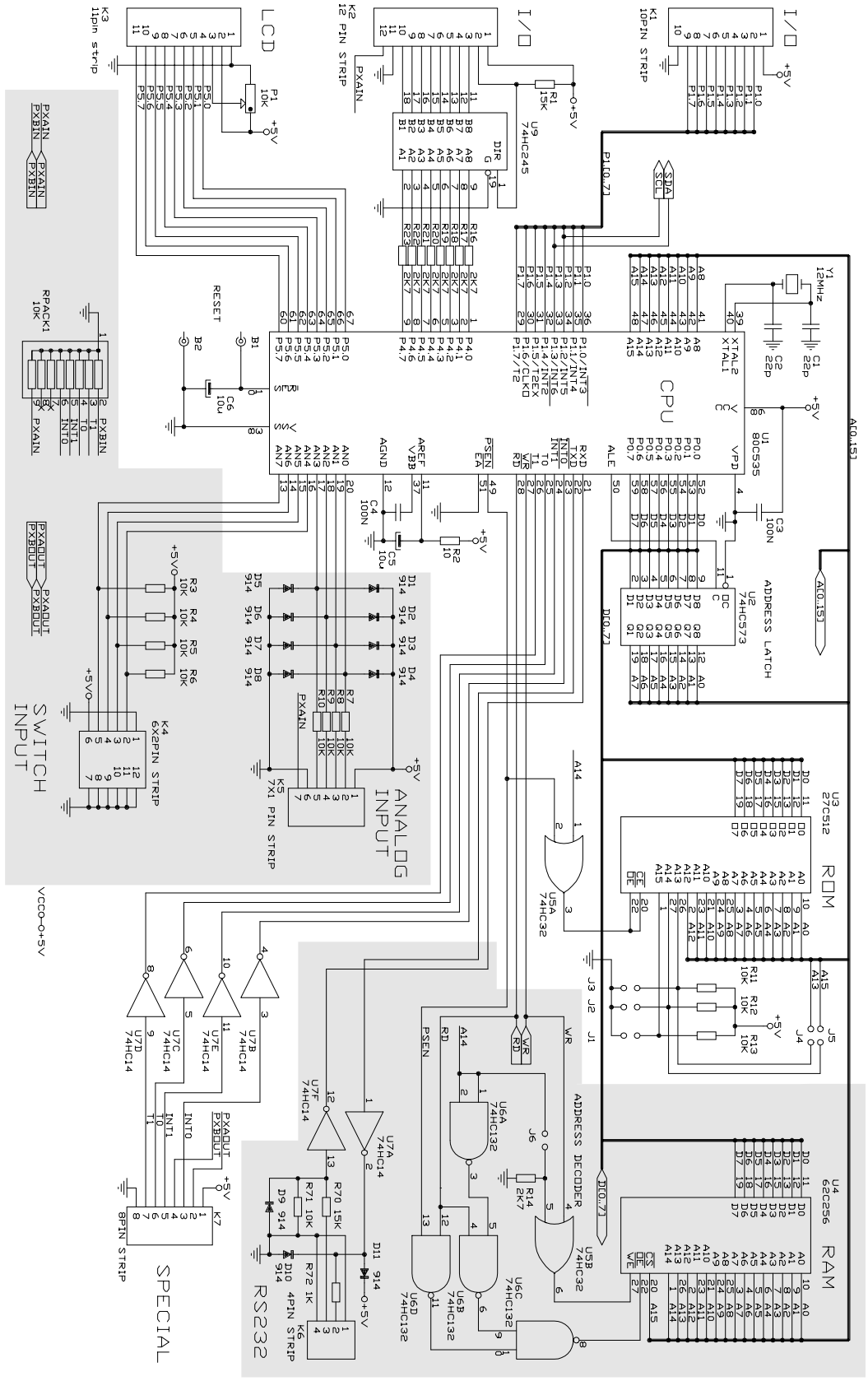



*Figure 1: Shematic diagram*

```
$MOD515
        ORG     0h          ;start at location 0 hex
        jmp     main        ;go to main program

        ORG     3h          ;start for INT0 routine
        inc     P4          ;do the actual work
        reti                ;and return from interrupt

        ORG     100h        ;main program starts here
MAIN:   mov     P4,#0
        setb    IT0         ;set flag IT0 - edge
        setb    EX0         ;set flag EX0 - enable
        clr     IE0         ;clear pending interrupts
        setb    EAL         ;enable all interrupts

        call    lcd_ini             ;init lcd              *
        call    lcd_coff            ;cursor off            *

loop:   inc     P1                  ;increment P1
        mov     r4,P1               ;take result from P1   *
        mov     r3,#0               ;                      *
        call    bin2bcd             ;convert it to BCD     *
        mov     r2,#5h              ;position on screen    *
        call    lcd_4bcd            ;display result        *

        mov     r4,P4               ;take result from P4   *
        mov     r3,#0               ;                      *
        call    bin2bcd             ;convert it to BCD     *
        mov     r2,#45h             ;position on screen    *
        call    lcd_4bcd            ;display result        *
        call    del_15ms            ;wait 15ms
        jmp     loop

$include(535lcd.asm)
   END
```

# 10. MICROPROCESSOR BASED MULTICHANNEL ANALYZER

**OBJECTIVE**

The objective of this experiment is to see how a microprocessor can be used for a simple multichannel analysis.

**REVIEW**

Microprocessor can be used to built a multichannel analyser. All functions for a complete (but moderate speed) MCA can be programmed into a microprocessor; in this case it is programmed into SAB 80C535.

The use of microprocessor has some advantages over the use of hardware; it keeps the electronics simple since complete MCA is built by microprocessor and software. Unfortunately such solution also has disadvantages: mainly it is the speed. The time needed to save a pulse in a spectrum is in the range of 10μs.

The MCA presented has 256 channels, two bytes per channel. The external random access memory (RAM) is used, and spectrum is stored in the low part of this memory. First 256 locations (address range from 0hex to 0FF hex) are used for low bytes of 256 channels, next 256 bytes (from address 100 hex to 1FF hex) are used for high bytes of 256 channels. External memory can be accessed through register by the name of DPTR.

*Connection to an ADC*: The MCA presented is on one side connected to ADC. The end-of-conversion (EOC) signal from ADC issues an interrupt to microprocessor by asserting logic high to pin INT0. Microprocessor breaks execution of the main program on interrupt, reads the result of conversion from the ADC through port P1, increments the content of corresponding channel, and returns to main program. Interrupt associated with signal EOC has the highest priority (3). This interrupt is served only when MCA is running (acquiring pulses). Pulse EOC must be at least 2μs in duration otherwise microprocessor can not recognise it as a valid interrupt. If pulse is shorter then on-board univibrators can be used to extend it.

The MCA should increment content of the channel only once per EOC pulse, therefore interrupt INT0 must be edge sensitive; flag IT0 must be set. Interrupt input INT0 is enabled by setting flag EX0 and flag IE0 must be cleared initially. Priority for this interrupt is the highest, therefore flags IP0.0 and IP1.0 are set.

When the result of conversion is safely transferred to microprocessor an output pin P4.0 gets toggled. This signal could be used to enable next acquisition at ADC.

*Connection to a PC*: The MCA presented can not show spectrum by itself since it does not have graphical display. Also, the MCA does not have any switches to control its operation. Instead the MCA is connected to a PC through RS232 serial link. All commands are issued from PC, and the spectrum to be shown is transferred to PC and

displayed at its monitor. PC controls microprocessor based MCA by sending ASCII characters to it. A list of acceptable characters follows (capitals only ! ):

**COMMANDS**

| Character | Action |
|---|---|
| C | clear spectrum and time |
| A | start acquisition |
| S | stop acquisition |
| R | send spectrum to PC; ASCII format, five digits, MSD, ch.0 first |
| T | send real time to PC; ASCII format, five digits, MSD first; 50ms increment |
| L | send live time to PC; ASCII format, five digits, MSD first; 50ms increment |

Microprocessor loops in the main part of program and periodically checks serial receiving flag RI as it was demonstrated already in experiment μP.6. If a valid character is received then microprocessor does the job requested as listed in table above, otherwise it echoes back to PC the received character followed by a question mark. Fixed transmission speed of 9600 Baud is selected. Use 12MHz crystal!

*Timing*: MCA incorporated two timers: true time counter and live time counter. Counter/timers T0 and T1 from microprocessor are used.

True time counter (counter/timer T0) is programmed to count internal microprocessor clock (1MHz), and divide it by 200. This counter interrupts the execution of main program on overflow; this happens every 200μs. At interrupt from T0 microprocessor increments the content of variable called TrueTime; this variable is two bytes in length and is located in registers of microprocessor. Timer T0 is running only when MCA is running (acquiring pulses). The content of variable TrueTime can be initialised at the beginning of the acquisition and read through a RS232 link by a command. To put timer/counter T0 in this mode of operation flags C/T0 and GATE0 must be cleared. The timer is enabled by setting flag TR0. The interrupt from true time counter is not as important as the one for serving ADC, and is assigned lower priority of 1.

Live time counter (counter/timer T1) acts almost the same as counter/timer T0, except that the counter can be temporarily stopped by external gating signal connected to pin INT1 of microprocessor. Therefore timer/counter T1 (live time counter) is running only during acquisition and when gating signal at external connection K7/pin 5 is low (which is true by default if nothing is connected to this input). Such operation is achieved by clearing flag C/T1, and setting flag GATE1. The timer is enabled by setting flag TR1. Corresponding variable to be incremented on overflow of counter T1 is named LiveTime, and is placed beside the aforementioned variable TrueTime in internal microprocessor registers.

The MCA presented has no facility for pre-setting the time of measurement. This facility could be implemented; preset value should be sent from PC and stored in memory of microprocessor, content of corresponding counter should be periodically compared with preset value, and acquisition stopped when the two are equal. This option is not programmed in the example presented, since this would prolong the program.

All interrupts are enabled globally by setting the flag EAL. Setting this flag also enables acquisition.

**EXPERIMENT**

Connect ADC from one of the previous experiments to microprocessor system, and microprocessor system to PC through RS232 channel. Use the cable assembled for experiment μP.6. Run prepared program. Observe signals in system and analyse quality of such MCA.

The software for this experiment is given below.

```
$MOD515

        ORG     0               ;start at 0hex
        jmp     init            ;jump to main prog.

        ORG     3h              ;external interrupt 0
        jmp     ADC_EOC         ;jump to ADC_EOC

        ORG     0Bh             ;timer/counter T0
        jmp     Timer_T0        ;jump to Timer_T0

        ORG     1Bh             ;timer/counter T1
        jmp     Timer_T1        ;jump to Timer_T1

        ORG     100h            ;the start of program
init:   call    I_232           ;initialise RS232
        call    I_CT01          ;initialise T0, T1
        call    I_INT0          ;initialise INT0

;-------------------------------------------------------
; Main part of the program
;-------------------------------------------------------
Main:   jnb     RI,Main         ;wait for character
        mov     a,SBUF          ;take char. from buffer
        clr     RI              ;clear receive flag

        cjne    a,#'C',no_C     ;character = "C" ?
        call    Spect_clr       ;yes -> clear spectrum
        mov     a,#'C'          ;"C"
        call    Ch_back         ;send char back
        call    CRLF            ;send CR & LF
        jmp     Main

no_C:   cjne    a,#'A',no_A     ;character = "A" ?
        setb    EAL             ;yes: acquire, en. int.
        setb    TR0             ;enable RT counter
        setb    TR1             ;enable LT counter
        call    Ch_back         ;send char back
        call    CRLF            ;send CR & LF
        jmp     Main            ;next message
```

```
no_A:   cjne    a,#'S',no_S     ;character = "S" ?
        clr     EAL             ;yes -> stop, dis. int.
        clr     TR0             ;dis. real time counter
        clr     TR1             ;dis. live time counter
        call    Ch_back         ;send char back
        call    CRLF            ;send CR & LF
        jmp     Main            ;next message

no_S:   cjne    a,#'R',no_R     ;character = "R" ?
        call    Send_Results    ;yes -> results to PC
        jmp     Main            ;next message

no_R:   cjne    a,#'T',no_T     ;character = "T" ?
        call    Send_Real       ;yes -> real time to PC
        jmp     Main            ;next message

no_T:   cjne    a,#'L',no_L     ;character = "L" ?
        call    Send_Live       ;yes -> live time to PC
        jmp     Main            ;next message

no_L:   call    Ch_back         ;send char back
        mov     a,#'?'          ;what is this ?
        call    Ch_back         ;send "?"
        call    CRLF            ;send CR & LF
        jmp     Main            ;next message

;-----------------------------------------------------
; Rutine handles interrupt from ADC
; read result from P1 and increment channel
; 29 micros min./34 micros max., release after 7 micros
;-----------------------------------------------------
ADC_EOC:
        push    dpl             ;2-save context
        mov     dpl,P1          ;2-take result from P1
        setb    P4.0            ;1-toggle P4.0/free ADC
        clr     P4.0            ;1-
        push    dph             ;2-save context
        push    acc             ;2-
        mov     dph,#0          ;2-low byte
        movx    a,@dptr         ;2-channel content lo
        add     a,#1            ;1-increment it
        movx    @dptr,a         ;2-and store back
        jnc     adc_f           ;2-carry ?
        mov     dph,#1          ;2-Y: hi byte
        movx    a,@dptr         ;2-channel content hi
        add     a,#1            ;1-increment it
        movx    @dptr,a         ;2-and store back
adc_f:  pop     acc             ;2-restore context
        pop     dph             ;2-
```

```
        pop     dpl             ;2-
        reti                    ;2-and return

;---------------------------------------------------
; Routine handles interrupt from timer T0
; increment variable RealTime
;---------------------------------------------------
Timer_T0:
        push    acc             ;save processor context
        push    psw             ;
        setb    rs0             ;use register bank 1
        djnz    r4,no_re        ;every 250th time :50ms
        mov     r4,#250         ;init counter again
        xch     a,r2            ;take Real_lo
        add     a,#1            ;increment lo
        xch     a,r2            ;store back
        xch     a,r3            ;take Real_hi
        addc    a,#0            ;carry to hi
        xch     a,r3            ;store back
no_re:  pop     psw             ;restore context
        pop     acc             ;
        reti                    ;return

;---------------------------------------------------
; Rutine handles interrupt from timer T1
; increment variable LiveTime
;---------------------------------------------------
Timer_T1:
        push    acc             ;save processor context
        push    psw             ;
        setb    rs0             ;use register bank 1
        djnz    r7,no_li        ;every 250th time :50ms
        mov     r7,#250         ;init counter again
        xch     a,r5            ;1-take Live_lo
        add     a,#1            ;1-increment lo
        xch     a,r5            ;1-store back
        xch     a,r6            ;1-take Live_hi
        addc    a,#0            ;1-carry to hi
        xch     a,r6            ;1-store back
no_li:  pop     psw             ;restore context
        pop     acc             ;
        reti                    ;2-return

;---------------------------------------------------
; Routine sends spectrum through RS232 line to PC
; starts with channel 0: high byte, low byte; ...
;---------------------------------------------------
Send_results:
        mov     dpl,#0          ;start with ch. 0
send_l: mov     dph,#1          ;take hi byte
        movx    a,@dptr         ;from memory
```

```
        mov     r3,a            ;and put it into r3
        mov     dph,#0          ;take low byte
        movx    a,@dptr         ;from memory
        mov     r4,a            ;and put it into r4
        call    send6dig        ;then send it to PC
        call    CRLF            ;send Cr & LF
        mov     a,dpl           ;next channel
        add     a,#1            ;
        mov     dpl,a           ;
        jnc     send_1          ;last channel ?
        call    CRLF            ;send Cr & LF
        ret                     ;yes -> return

;-------------------------------------------------------
; Routine sends TrueTime through RS232 line to PC
; order: hi byte, mid byte, lo byte
;-------------------------------------------------------
Send_Real:
        mov     r3,11
        mov     r4,10
        call    send6dig
        call    CRLF            ;send Cr & LF
        ret                     ;return

;-------------------------------------------------------
; Routine sends LiveTime through RS232 line to PC
; order: hi byte, mid byte, lo byte
;-------------------------------------------------------
Send_Live:
        mov     r3,14
        mov     r4,13
        call    send6dig
        call    CRLF            ;send Cr & LF
        ret                     ;return

;-------------------------------------------------------
; Routine clears spectrum and time from memory
;-------------------------------------------------------
Spect_clr:
        mov     a,#0            ;value to store
        mov     dph,#0          ;init pointer dptr
        mov     dpl,#0          ;
clr_lo: movx    @dptr,a         ;clear channel lo
        djnz    dpl,clr_lo      ;next channel ?
        mov     dph,#1          ;all done -> init ptr
        mov     dpl,#0          ;
clr_hi: movx    @dptr,a         ;clear channel hi
        djnz    dpl,clr_hi      ;next channel ?
        setb    rs0             ;use register bank 1
        mov     r2,#0           ;init RealTime variable
```

```
        mov     r3,#0           ;
        mov     r4,#250         ;
        mov     tl0,#0          ;and real time counter
        mov     r5,#0           ;init LiveTime variable
        mov     r6,#0           ;
        mov     r7,#250         ;
        mov     tl1,#0          ;and counter
        clr     rs0             ;use register bank 0
        ret

;-------------------------------------------------------
; initialise RS232 link - 9600 Baud
;-------------------------------------------------------
I_232:  mov     sp,#128
        clr     SM0             ;select serial mode 1
        setb    SM1             ;
        clr     SM2             ;
        clr     TI              ;clear transmitt TI flag
        clr     RI              ;clear receive RI flag
        orl     PCON,#80h       ;use 9600 Baud, fixed
        orl     ADCON,#80h      ;
        setb    REN             ;enable ser. reception
        ret                     ;and return

;-------------------------------------------------------
; initialise counter/timer T0 and T1
;-------------------------------------------------------
I_CT01: mov     TMOD,#0A2h      ;select mode for T0,1
        mov     th0,#56         ;reload value 200 us
        mov     th1,#56         ;reload value 200 us
        mov     tl0,#56         ;
        mov     tl1,#56         ;
        anl     ip1,#0F5h       ; priority 1
        orl     ip0,#00Ah       ;set bits 1 and 3
        setb    ET0             ;enable T0 interrupts
        setb    ET1             ;enable T1 interrupts
        ret

;-------------------------------------------------------
; initialise external interrupt INT0
;-------------------------------------------------------
I_INT0: setb    IT0             ;edge sensitive INT0
        orl     ip1,#1          ;setb ip1.0
        orl     ip0,#1          ;setb ip0.0: priority 3
        clr     IE0             ;clear pending int
        setb    EX0             ;enable INT0
        ret

;-------------------------------------------------------
```

```
; Routine sends six digits through RS232 line to PC
;-------------------------------------------------------
send6dig:
        call    bin2bcd
        mov     a,r7            ;take most sig. digits
        anl     a,#0fh          ;5th place
        add     a,#'0'          ;add char "0"
        call    Ch_back         ;send it
        mov     a,r6            ;take mid sig. digits
        swap    a               ;4th please
        anl     a,#0fh          ;
        add     a,#'0'          ;add char "0"
        call    Ch_back         ;send it
        mov     a,r6            ;take mid sig. digits
        anl     a,#0fh          ;3rd please
        add     a,#'0'          ;add char "0"
        call    Ch_back         ;send it
        mov     a,r5            ;take least sig. digits
        swap    a               ;2nd please
        anl     a,#0fh          ;
        add     a,#'0'          ;add char "0"
        call    Ch_back         ;send it
        mov     a,r5            ;take least sig. digits
        anl     a,#0fh          ;1st please
        add     a,#'0'          ;add char "0"
        call    Ch_back         ;send it
        ret

;-------------------------------------------------------
; Routine sends CR and LF through RS232 line to PC
;-------------------------------------------------------
CRLF:   mov     a,#10           ;line feed
        call    Ch_back         ;send
        mov     a,#13           ;carriage return
        call    Ch_back         ;send
        ret

;-------------------------------------------------------
; Routine sends character through RS232 line to PC
;-------------------------------------------------------
Ch_back:
        mov     SBUF,a          ;send character back
        clr     TI
wt_txd: jnb     TI,wt_txd       ;wait until sent
        clr     TI              ;acknowledge sending
        call    del_100
        ret                     ;and return

$include(535lcd.asm)
   end
```

## Fault identification

Complex microprocessor based instruments can be checked part by part. Each part should be checked following the statements given in former chapters. If the experiments were made from the beginning, then it is clear that the data and address bus are working properly, that the control lines are working, that the interrupt is OK, and that the serial communication port is OK. The only remaining thing that can go wrong is the software. Please check that the correct software was programmed into the ROM.

In general, it is not possible to write your own testing programs for the faulty microprocessor system, and the fault can not be easily isolated and corrected. Additionally, we usually do not know how the software is written. The only thing that remains to check is to use oscilloscope and observe data bus, address bus and control lines. As stated at the beginning, the data and address lines should toggle, and have either value of logic low or high. Anything in between is bad.

It helps a lot to understand what the faulty unit is supposed to do. For the MCA we know: ADC converts a pulse, sends EOC, the microprocessor gets an interrupt, it reads the result from the ADC, then reads the current content of the memory, increments it, and stores new content back into the memory. We should be therefore able to detect a chip select signal and a read signal for the memory soon after the EOC, and a little later a chip select and write signal for the RAM again. In more complicated MCAs there might be several of read and write signals, since the content of the channel may be written in more than one memory location. Similarly, for our design, after a command for Clear spectrum, there should be a series of write signals for the RAM, since the content of all channels should be erased. And every command R for send spectrum should be followed by a series of read signals for the RAM.

Sometimes it is difficult to observe some chip select signals, because they come every so often. In this case a storage oscilloscope might be useful, since the event is stored and can be observed later. However, analog oscilloscope can be used to detect even such rare occurrences of a signal. Just put the oscilloscope to a relatively slow horizontal deflection (1ms/div will do), and select normal triggering. Then adjust the level of the triggering to a value just below the present value of the chip select signal (chip select signal is usually active low: it is at +5V most of the time, and drops to 0V for maybe a μs) and wait. If from time to time you see a trace running at the screen, then there are pulses coming at the input of the oscilloscope, and they trigger it. For every triggering there is one drawing of the trace at the screen. Do not try to see the shape of such rare CS pulse with the analog scope, you will not succeed, but at least you know, it is there.