USING AVR MICROCONTROLLERS FOR PROJECTS

■ **K. PADMANABHAN, P. SWAMINATHAN & S. ANANTHI**

The AVR 8535 microcontroller and its new version ATmega8535 are versatile, high-performance but low-cost chips. This article series covers typical applications of this processor illustrating its power and cost-effectiveness in an embedded system.

The AVR family comprises several chips, all with almost the same instruction set. Of them, the 90S8515, 90S8535 and ATmega8535 chips are low-cost and readily available with the complete set of port pins. The ATmega8535-16 is more powerful and available for around Rs 250. Capable of running at 16 MHz and achieving almost 16 million instructions per second (MIPS), it is one of the fastest devices available in the market today.

Using ATmega8535, you can build a microcontroller-based project with following features:

1. Four ports, of which one of them has eight analogue-to-digital converter (ADC) channels

2. ADC conversion time is as little as 60 microseconds. Imagine adding an external ADC to 8051 or any other microcontroller chip—that would have taken the cost to over four digits. And mind you, it is a 10-bit ADC, not just 8-bit.

3. If an 8MHz crystal is connected, each instruction executes in 1/8th of a microsecond. The 89C51 at 12MHz clock had its internal division by twelve, so it ran at just one microsecond. Thus, ATmega8535 chip is eight times faster with an 8MHz crystal. However, you can also use a higher-frequency crystal. The chip is basically a RISC processor that executes most instructions in one clock cycle itself.

4. The chip has RS-232 transmit and receive terminals much like the 8051 family, but it can support even higher baud rates.

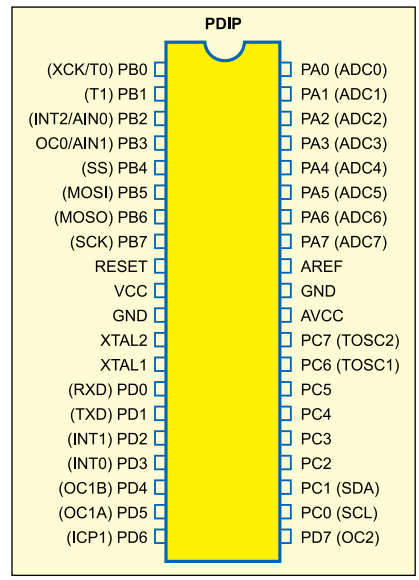

Fig. 1: Pin configuration of ATmega8535

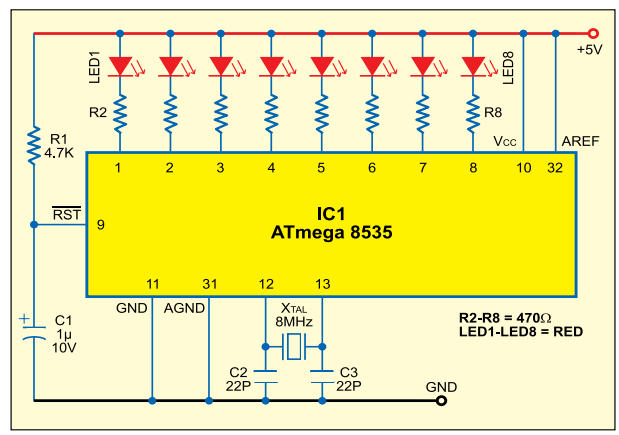

Fig. 2: A simple LED display circuit using ATmega8535

5. It has quite a few internal registers, RAM, EEPROM and CODE memory (flash memory in excess of 4kB).

6. The instruction set is versatile, complete with several arithmetic, logic and transfer instructions and related jump instructions, etc.

7. An analogue comparator pin, which can compare an external analogue voltage and take control action.

8. Reset is possible through the software, and a watchdog is provided. Power-down or sleep modes are available.

9. An additional serial interface, known as the SPI bus, with three wires: data (2) and clock (1). These pins can be used for programming or loading the code from a PC through the printer port or serial port. For programming the internal flash memory locations, just 5V supply is enough.

10. Two PWM output pins, which are useful for power control applications.

11. Several timers as in other members of the 8051 family, but with much better time resolution.

12. Additional features like input capture and output compare.

Here, we shall delve into the chip's operations with typical programs and circuits. All the development tools including 'C' compiler are available for free from the Internet.

The features of ATmega8535 make it the right candidate for various embedded control applications. Even a digital filter can be implemented on the device, provided you are fully conversant with its hardware and software features. You can download the databook of ATmega8535 from the 'ATMEL.com' Website to understand its features and work out simple applications.

The sample programs given here can be used to yield a powerful con-

troller for many applications like a filter or motor controller.

Programming the chip

The AVR source code file with '.asm' extension can be written using either the EDIT, Wordpad or notebook programs.

As with all microprocessor or microcontroller programs, for the source code, one has to enter the program by mnemonics and assembler directives and then convert the same into a code list for the program. (Directives are assembler commands used to control the input, output and data allocation of the assembler. These are, however, not translated into op-codes directly.) This is done using the cross-assembler software 'avrasm.exe.'

To describe the modus of writing of an Assembly language program, a simple program (LED.ASM) for AVR processors is given below:

LED.ASM

```
.NOLIST
.INCLUDE "m8535def.inc"
.LIST
.DEF      mp = R16
.org $0000 ; Reset address
rjmp      main
main:
ldi R16,low(RAMEND); Load low byte
         address of the end of the RAM
         into register R16
out SPL,R16 ; Initialise stack
         pointer to the end of the
         internal RAM
ldi R16,high(RAMEND) ; Load high byte
         address of the end of the RAM
         into register R16
out       SPH, R16  ; Initialise high
          byte of stack pointer ; to the
          end of the internal RAM
ldi       mp,0b11111111
          out       DDRB,mp
loop: ldi mp,0x00
          out       PORTB,mp
          Rcall delay
          ldi       mp,0xFF
          out       PORTB,mp
          Rcall delay
          ldi       mp,0xFF
          out       PORTB,mp
          rjmp      loop
delay: clr r19
          ldi r17,$ff
          loop1: inc r17
          brne loop1
          inc r19
          brne r19,loop1
          ret
```

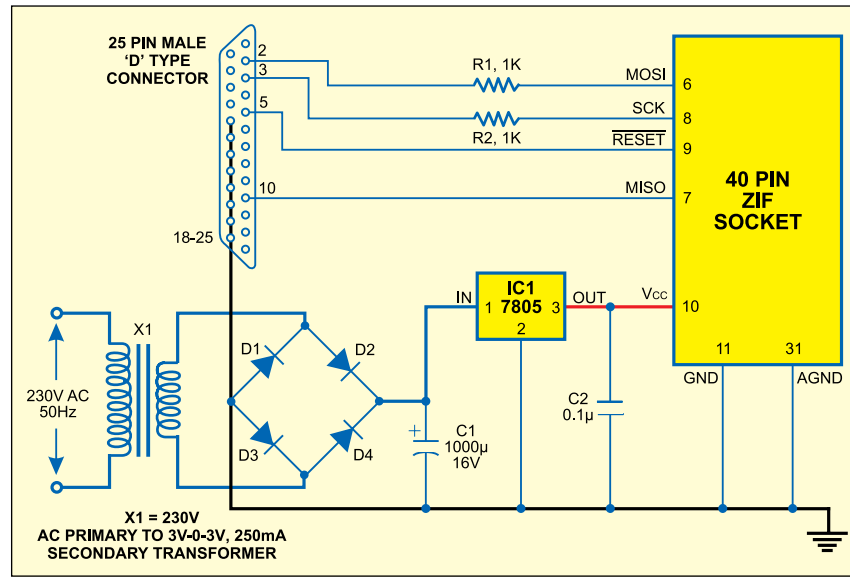

Fig. 3: Circuit diagram of AVR programmer (Pod)

This program helps you understand:

1. Access to the output port (here port B, where LEDs are connected)
2. The different parts of a typical assembler program
3. Different conventions like use of semicolon, upper-/lower-case letters, etc

Explanatory notes for LED.ASM

1. In Assembly language, all the text on a line after a semicolon (;) is treated by the cross-assembler as comments and it does not use it for code formation.

2. Including the m8535def.inc processor-specific file in Assembly program means all the I/O register names, I/O register bit names, etc appearing in the datasheet can be used. Failure to include this file may result in a number of error messages. Ensure that this file is placed in the same directory as your source code file (LED.asm in this case). Else, give complete path for the m8535def.inc file.

3. Following conventions have been used in the program:

(a) Words in upper-case letters are used for command directive words of the Assembly language or predefined ports of the processor.

PARTS LIST

Parts list for LED display circuit (Fig. 2)

Semiconductors:
IC1 - ATmega8535
LED1-LED8 - Red LED

Resistors (all ¼-watt, ±5% carbon):
R1 - 4.7-kilo-ohm
R2-R8 - 470-ohm

Capacitors:
C1 - 1µF, 10V electrolytic
C2, C3 - 22pF ceramic disk

Miscellaneous:
X_{TAL} - 8MHz

Parts list for AVR Programmer (Fig. 3)

Semiconductors:
IC1 - 7805 5V regulator
D1-D4 - 1N4007 rectifier diode

Resistors (all ¼-watt, ±5% carbon):
R1, R2 - 1-kilo-ohm

Capacitors:
C1 - 1000µF, 16V electrolytic
C2 - 0.1µF ceramic disk

Miscellaneous:
X1 - 230V AC primary to 6V, 250mA secondary transformer
- 40-pin ZIF socket
- 25-pin D-type male connector

Parts list for message display on the LCD (Fig. 6)

Semiconductors:
IC1 - ATmega8535

Resistors (all ¼-watt, ±5% carbon):
R1 - 4.7-kilo-ohm
VR1 - 10-kilo-ohm preset

Capacitors:
C1 - 1µF, 10V electrolytic
C2, C2 - 22pF ceramic disk

Miscellaneous:
X_{TAL} - 8MHz
- 16x1-character Hitachi make LCD or 16×2-character LCD

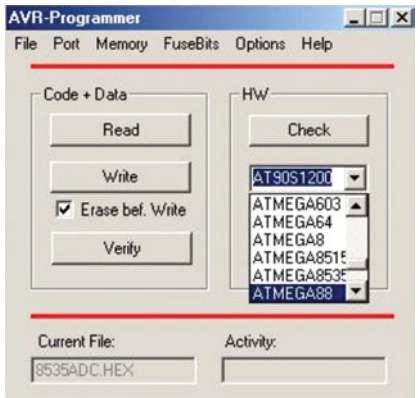

Fig. 4: Screenshot of AVR-Programmer

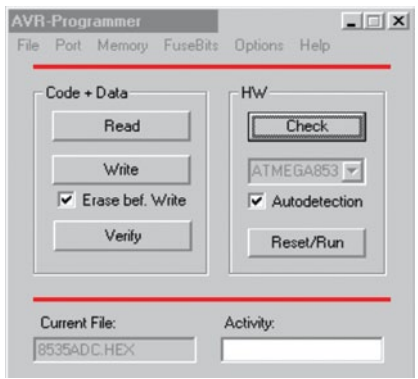

Fig. 5: Screenshot of AVR-Programmer showing activity window

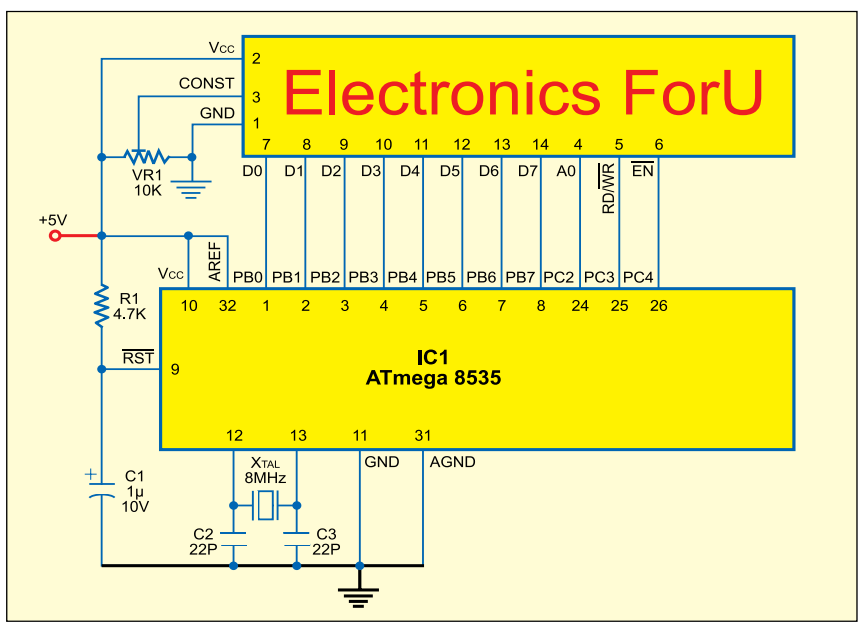

Fig. 6: Circuit for message display on the LCD

(b) Mnemonic words are written in lower case.

4. LIST directive turns on the listing output if it had been previously turned off. Similarly, NOLIST directive, if used, will turn off the listing output.

5. DEF directive is used to define a text-substitution label for a string. A label/name is easy to remember. Here, register R16 is replaced with 'mp' name. Thus whenever 'mp' is encountered in the source code, it will be automatically replaced with 'R16.'

6. '.org $0000' defines the reset address. When power is switched on, the program starts from this location. A restart from the reset address can be activated by resetting the respective hardware pin of the chip (pin 9) or upon watchdog timer reaching its zero count. A relative jump command (rjump) at this reset location directs the program execution to label (main)—as long as the label is within 2k locations from the reset address (0000). Incidentally, 'rjmp main' is the first code-generating instruction.

7. It is essential to set up the stack pointer before being able to call any subroutine, since stack is required for saving the return address, where the next program execution is to start from. The program lines starting with 'ldi R16,low(RAMEND)' and ending with 'out SPH, R16' do just that.

8. The 'ldi mp, 0b11111111' and 'out DDRB, mp' lines set port-B pins as the output. The first line, interpreted as 'load immediate (ldi) into register 'mp'', loads binary value '11111111' into the 'mp' register. The second line transfers the contents of 'mp' (11111111) to the data direction register of port B (DDRB). DDRB is already defined in the m8535def.inc file. (If you want to set port-B pins as input, load binary '00000000' into 'mp' and output it to DDRB.) Incidentally, '0b' precedes a binary number. Similarly '0x' precedes a hex number. Numbers without these prefixes denote decimal numbers by default. Hence you may replace '0b11111111' with either '0xFF' or simply '255' to achieve the same results.

9. The rest of the program starting at label 'loop:' and ending with 'rjmp loop' achieves switching on and off of the LEDs with a delay. The delay subroutine starting at label 'delay:' and ending with return instruction 'ret' is called from within the loop.

Initially, 'mp' is loaded with hex value '00' and output through port-B pins, making them low. Since the cathodes of all the eight LEDs are connected to these port pins via current-limiting resistors, the LEDs light up. Thereafter, the delay subroutine (Rcall delay) is called and 'mp' is loaded with hex value 'FF' and transferred to the port-B output to turn off the LEDs. The loop is repeated as long as the power is switched on.

10. The internal R-C clock of ATmega8535 is 1 MHz by default. In the absence of 'Rcall delay' instruction, each of 'ldi' and 'out' instructions requires 1000 ns, while 'rjmp' instruction requires 2000 ns. Thus loop execution would take 4000 ns. This amounts to LED switching rate of 250 kHz.

Introduction of delay between switching on and off reduces this frequency to around 0.5 Hz by decrementing registers 'r19' and 'r17' from '255' to '0,' thereby making the elapsed time slower by 256×256 (which works out to around 0.5Hz rate).

After assembling the LED.asm source file, the program will have eight words. The LED.LST file stores the result of the assembly process in the form of a listing.

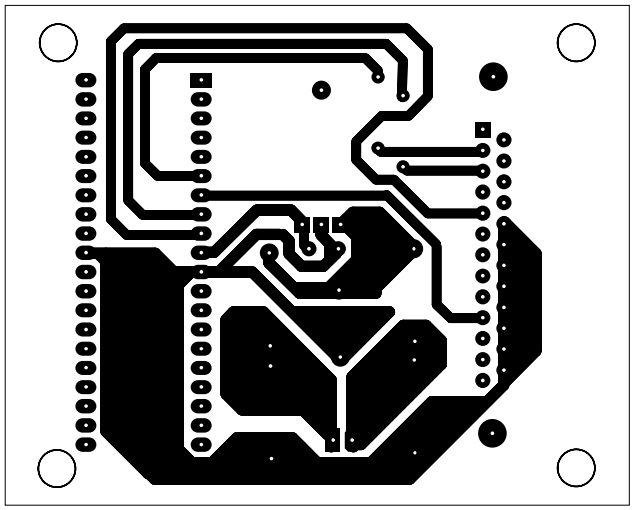

Fig. 7: Actual-size, single-side PCB layout for AVR programmer (Pod)

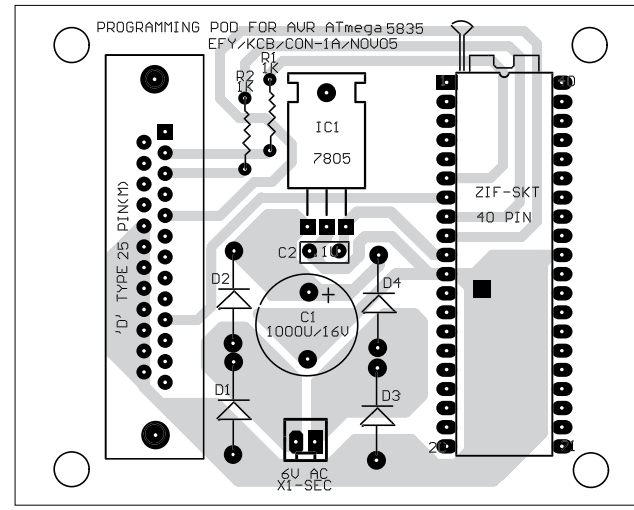

Fig. 8: Component layout for the PCB in Fig. 7

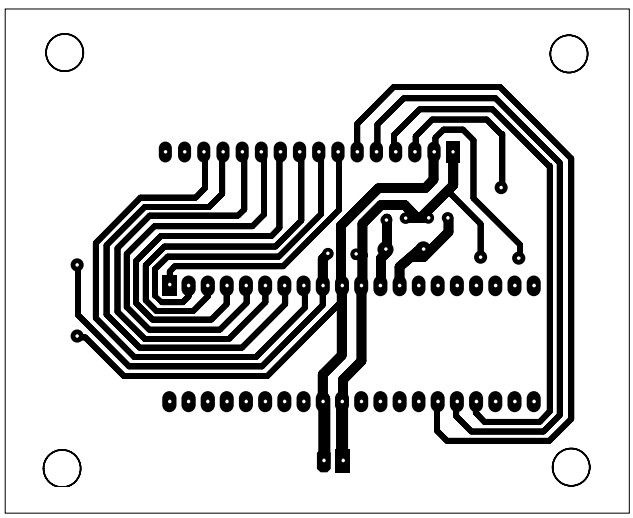

Fig. 9: Actual-size, single-side PCB layout for message display on LCD

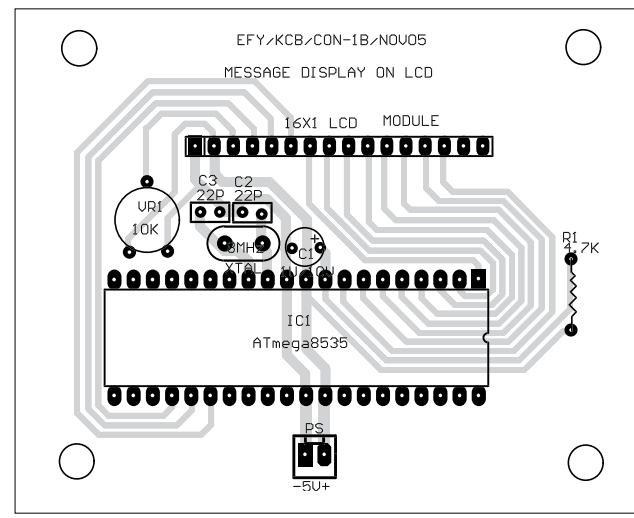

Fig. 10: Component layout for the PCB in Fig. 9

Once a program has been written using any editor, wordpad or notepad, it is assembled using the avrasm.exe AVR assembler, available on the download link given at the end of this article. Of course, the AVRSTUDIO 4.0 integrated development environment (IDE) is more versatile and user-friendly software for development, but the avrasm.exe assembler is simpler and direct.

Simply typing 'avrasm -i LED.asm LED.lst LED.hex' under the DOS prompt makes the cross-assembler generate code for the LED.hex file and also provide a text file giving both the code and the program together in LED.lst. Thus, you get the LED.lst listing file and the LED.hex Intel hex code file.

Alternatively, you can prepare a batch file as follows:

Upon DOS prompt, enter 'copy con avr.bat.' In the following line, type 'Avrasm -i %1.asm %1.lst %1.hex.' Pressing 'F6' key in the following line displays 'Control-Z.' Now pressing the 'Enter' key displays "1 file copied."

Now the avr.bat file has been prepared. This simple batch file is invoked to assemble this (or any) program by typing 'Avr LED' upon the DOS prompt and pressing the 'Enter' key.

This assembles the program, and forms both the list file (that contains the code-cum-Assembly listing) and the hex file (the actual Intel-format hex file for use by the programmer).

Likewise, any other assembly program 'xxx.asm' can be coded into the hex file by simply typing 'avr xxx' on DOS prompt. 'xxx' denotes the name of the program. The '.asm' is not to be typed.

In our LED.asm program, we have included the m8535def.inc file. This file is required along with the avrasm.exe cross-assembler. For other AVR processors like 90S8515, 90S8535 and at-Tiny 26, the files to be included are 8515def.inc, 8535def.inc and tn26def.inc, respectively.

The next task is to burn the code into the chip. Note that a chip previously programmed or erased is automatically erased when a new program is burnt into it using the device programmer as described below.

The AVR device programmer

The AT-PROG programmer software is used for programming ATmega8535.

This menu-drive programming software is simple to use and invoked from command prompt.

The software uses a simple pod connected to the printer port of a computer. The circuit of the pod (shown in Fig. 3) is very simple. It just connects the IC to be programmed to the pins of the PC's printer port.

This circuit is assembled on a small PCB with a D25 male-female plug at one end. The IC base is a 40-pin zero-insertion-force socket (ZIF). This enables easy insertion and removal of the IC to be programmed.

The AT_PROG.exe is a simple programming software that can be run under DOS prompt by typing AT-PROG. The files At-prog-hlp.htm, At-prog.exe, At-prog.cfg and At-prog.ini should be placed in one directory before running the AT-PROG. These files have been included in this month's EFY-CD as part of this article.

The menu-driven window of the AT-PROG programmer has the following menu items:

1. File menu. This menu is used to select or open the LED.hex file, or whatever, which is to be programmed into the device.

Pull down the menu by clicking it. Under 'Open' option, enter the file name as 'LED.hex' and press 'Enter.' The IC to be programmed is selected from the AVR-Programmer window by clicking the edge of the small rectangular window and choosing the IC as shown in Fig. 4. Now connect the printer-port connector to the programming pod, whose circuit is shown in Fig. 3.

2. ***Write menu.*** On clicking the 'Write' menu, the 'Activity' window at the bottom whitens and shows 'Connecting' (refer Fig. 5). Then, the data is transferred to the IC and verified after programming, showing 'ok' in the same window.

3. Check menu. This menu is used to find out whether the IC is inserted in the socket and whether the connector connections are okay. It will indicate an error if the IC is not there or not responding.

In this mode of programming, the serial-peripheral interface (SPI) of the AVR chip is used. This interface has three wire connections:

(i) Master output and slave input (MOSI)

(ii) Master input and slave output (MISO)

(iii) Serial clock (SCLK)

Using these wires, the SPI interface does the serial transfer of data (i.e., our program codes) into the chip, which is configured as a slave. The data and clock are connected via MOSI and SCLK pins of the chip, respectively. Upon reception of each byte, the chip acknowledges it by sending a byte (53hex).

In 'Check' mode, the IC is enquired about its name by the computer (Master), which it replies with its signature code embedded in the chip memory by the manufacturer. Each IC has its specific signature code. Thus, by noting the code itself, what IC is being programmed will be known to the computer. So the small window under the device-select rectangular window can be clicked to show 'autodetect' the IC.

4. Options menu. In this menu, the speed of the clock used for transferring data from the computer can be selected as 'slow,' 'normal' or 'fast.' With present high-speed PCs, choose 'normal' or 'slow.' In the same menu, the 'read signature bytes' option is to be enabled and it is so by default.

5. Port menu. The port menu, which is next to the file menu, is useful if a different printer port is available. The program automatically selects the available printer port.

When the 'Activity' window shows 'ok' after clicking the 'Write' menu, remove the programmed chip from the programmer circuit board and fix it onto the target circuit for the LED.asm program (shown in Fig. 2). Now apply 5V and press the switch connected to Reset pin, if needed. (The circuit resets at power-on.) The LEDs start blinking fast and the waveform can be observed on the CRO for any of the pins at the output to the LEDs. It will be around 600 Hz.

Message display on the LCD module

Method I. Given below is the source code for message display on the LCD module along with suitable comments wherever needed.

LCD_CHAR.ASM

```
;*************************************
; * This program writes a message on to
the LCD *
;*************************************
.NOLIST
.INCLUDE "m8535def.inc"
;device =ATmega8535
.LIST
;
; Constants
;
; Used registers
;
.DEF rmpr = R16
.DEF temp = R14
.DEF result=R12
.DEF mpr =R16
; Code starts here
;
.CSEG
.ORG $0000
;
; Reset-vector
rjmp Start ; Reset-vector

;******** various subroutines for LCD
display********
;cmd is the LCD module's command entry
subroutine. Command value in R16
         cmd:  cbi portc,2
                    cbi portc,3
                    cbi portc,4
                    out portb,r16
                    sbi portc,4
                    nop
          nop
          nop
          nop
          nop
          cbi portc,4
          rcall delay1
          ret
;lcdwr is the LCD module's data entry
subroutine.
ASCI code value in R16
         lcdwr:  cbi portc,2
                      cbi portc,3
                      cbi portc,4
                      sbi portc,2
                      out portb,r16
```

```
                    sbi portc,4
                    nop
                    nop
                    nop
                    nop
                    nop
                    cbi portc,4
                    rcall delay1
                    ret
;init_lcd is the LCD module's initialise
LCD routine for cursor, etc.
init_lcd:
ldi R16,$38 ;function_set command for
8-bit data
         drive
rcall cmd ;write this in the command
register for LCD
rcall delay1 ; wait since this takes some
         milliseconds for LCD
rcall delay1
ldi R16,$0e ; command for entry mode set
rcall cmd ; cursor active, display on,
          no blink.
rcall delay1
ldi R16,6 ; command for cursor shift
         right after each write
rcall cmd
ldi r16,1 ;command for clear display
rcall cmd
rcall delay1
ret

delay1:clr result
ldi R16,$a0 ; a suitable number for the
           required delay
loop2: inc R16 ; increments from 160
          ($a0) to 256 brne loop2
inc result ; increments result register
          from 0 to 255 brne loop2
ret ;got 256 times 95 for loop

; ********* End of the subroutine section
********
;
; ********** Main program **********
;
; Main program routine starts here
;
Start:    ldi       R16,low(RAMEND)
; Load the low byte address of the end
   of the RAM into register R16
         out       SPL,R16
; Initialise the stack pointer to the end
of the internal RAM
         ldi       R16,high(RAMEND)
; Load the high byte address of the end
  of the RAM into register R16
         out       SPH, R16
; Initialise the high byte address of the
  stack pointer to the end of the
  internal RAM

LCD:
 ldi r16,$ff
 out ddrb,r16 ;ff makes all bits as the
 output on port B
 out ddrc,r16 ;PORT C BITS USED FOR
            LCD WIRING
 ldi r16,$55
 out portb,r16;then alternate bits are
 low and high
($01010101)
          ;the above is a test which one
           can find if the program works!

 rcall init_lcd
 ldi R16,$80
 rcall cmd
; simply observe pins 1-8 for alternate
  high and low outputs!
 ldi R16,$45 ;"E"

 rcall lcdwr

 ldi R16,$6c ;"l"
 rcall lcdwr
 ldi R16,$65 ;"e"
 rcall lcdwr
 ldi R16,$63 ;"c"
 rcall lcdwr
 ldi R16,$74 ;"t"
 rcall lcdwr
 ldi R16,$72 ;"r"
 rcall lcdwr

 ldi R16,$6f ;"o"
 rcall lcdwr
 ldi R16,$6e ;"n"
 rcall lcdwr

 ldi R16,$c0 ; this command is to set to
            the next half of the LCD
 rcall cmd ;because 8 characters have
           filled the first half
;omit the above two lines if a two-row
          LCD display or a Hitachi 1-row
; display is used.
 ldi R16,$69 ;'i'
 rcall lcdwr
 ldi R16,$63 ;"c"

 rcall lcdwr

 ldi R16,$73 ;"s"
 rcall lcdwr
 ldi R16,$20 ;" "
 rcall lcdwr
 ldi R16,$46 ;"F"
 rcall lcdwr
 ldi R16,$6f ;"o"
 rcall lcdwr
 ldi R16,$72 ;"r"
 rcall lcdwr

 ldi R16,$55 ;"U"
 rcall lcdwr
here: RJMP HERE; Test of the serial
interface
```

This program displays 'Electronics ForU' on the LCD module (Fig. 6). The message may be displayed on the LCD in a single or two rows depending on the LCD module. In some LCD modules, the first eight characters are written consecutively, while for display of the next eight characters, the program needs to restart the cursor at address $C0. But Hitachi-make single-row types do not need to restart the cursor's address after the eighth entry; the characters can be written consecutively up to '16,' i.e., in a single row.

The program is named as 'LCD_CHAR.asm' and assembled into the '.hex' file by typing 'avr lcd_char' and invoking the cross-assembler AVR. Now the lcd_char.hex file is generated. The AT-PROG programmer burns this code into the flash memory of the ATmega8535.

Note that while assembling this program using 'avr lcd_char' command, the definition file for IC ATmega8535 (m8535def.inc) should be in the same directory.

Method II. This message display program uses look-up table. In the message display program described in Method I, 'Call lcdwr' instruction was written for each character. Here, instead, if we enter all the bytes for 'Electronics ForU' in a table, they can be picked up one by one until the end and shown on the LCD screen. For the

purpose, there is an instruction called load program memory (LPM).

The table, as also the name, is stored in the program memory. Here is the program along with necessary comments.

LCD TABLE.ASM

```
;————————————————————
;
.INCLUDE "m8535def.inc"
 ;device =ATmega8535
.LIST
;
; Constants
;
; Used registers
;
.DEF rmpr = R16
.DEF temp = R14
.DEF result=R12
.DEF mpr =R16
; Code starts here
;
.CSEG
.ORG $0000
;
; Reset-vector
rjmp Start ; Reset-vector

;********* various subroutines for LCD
display*********
;cmd is the LCD module's command entry
;subroutine.Command Value in R16
 cmd:     cbi portc,2
          cbi portc,3
          cbi portc,4

          out portb,r16
          sbi portc,4
          nop
          nop
          nop
          nop
          nop
          cbi portc,4
          rcall delay1
          ret
;lcdwr is the LCD module's data entry
subroutine.
Asci codeValue in R16
lcdwr:    cbi portc,2
          cbi portc,3
          cbi portc,4
          sbi portc,2

          out portb,r16
          sbi portc,4
          nop
          nop
          nop
          nop
          cbi portc,4
          rcall delay1
          ret
;init_lcd is the LCD module's initialize
LCD routine for cursor etc.
init_lcd:
ldi R16,$38 ;function_set command for
8 bit data drive
rcall cmd ;write this in the command
register for LCD
rcall delay1 ; wait since this takes some
          milliseconds for LCD
rcall delay1
ldi R16,$0e ; command for entry mode set
rcall cmd ; cursor active, display on,
no blink.
rcall delay1
ldi R16,6 ; command for cursor shift
           right after each write
 rcall cmd
 ldi r16,1 ;command for clear display
 rcall cmd
 rcall delay1
 ret

 delay1:
 clr result
 ldi R16,$a0 ; a suitable number for the
               required delay

loop2: inc R16 ; increments from 160
          ($a0) to 256
           brne loop2
           inc result ; increments result
           register from 0 to
           255
           brne loop2
           ret ;got 256 times 95 for loop

;************ End of the subroutine
section ***********
;
; ********** Main program ************
;
; Main program routine starts here
;
Start:    ldi       R16,low(RAMEND)

; Load low byte address of end of RAM
       into register R16
         out       SPL,R16
; Initialize stack pointer to end of
  internal RAM
         ldi       R16,high(RAMEND)
; Load high byte address of end of RAM
  into register R16
         out       SPH, R16
; Initialize high byte of stack pointer
  to end of internal RAM
 LDI R16,$FF
 OUT DDRB,R16
 OUT DDRC,R16 ; MAKE PORTS B AND C AS
OUTPUT PORTS (WIRED TO lcd)
 rcall init_lcd
 clr r17
 clr r18 ; required in the table fetch
 routine LCD:
 ldi ZH, high(table*2) ; Set up Z to
  point to the beginning of table
          ldi ZL, low(table*2)
          add ZL, r17
          ; Offset Z by r18:r17
          adc ZH, r18
          lpm
          ; Load

 mov r16,r0 ;get loaded value into r16
 cpi r16,$ff ;table end?
 breq idle
 rcall lcdwr ;write on lcd display
 inc r17
 rjmp LCD
; use Hitachi LCD display module if 1
row type is used; or else use ; ;any
two-row type LCD.
Otherwise, this above program sequence
will ;work only for 8 characters. The
rest will not be seen:
"Electron " ;only will be visible.

idle:
          ldi r16, (1<<SE) ; Enable sleep
          out MCUCR, r16
          sleep
          rjmp idle
table:
.db $45,$6c,$65,$63,$74,$72,$6F,
$6e,$69,$63,$73,$72,$6f,$72,$55,$
```

The actual-size PCB for programming and LCD message display are given in Figs 7 and 9, while their component layouts are shown in Figs 8 and 10, respectively.

In the first part of this article, we had described the main features of the AVR microcontroller and the hardware/software required for an AT-PROG programmer board interfaced to the printer port of a PC. Further, we explained the methods for message display on a liquid crystal display (LCD).

This part dwells on the architecture of ATmega8535 along with application programs exploiting its important features for embedded control.

Architecture of ATmega8535

Pin configuration of ATmega8535 was shown in Fig. 1 of Part 1. The device has ports for input/output, interrupts, serial communication and various others functions. There are a total of 32 pins, which are arranged as 'A,' 'B,' 'C' and 'D' ports for various functions as shown in Table I.

A crystal of maximum 16MHz or 8MHz frequency can be connected across pins 12 and 13 of ATmega8535 or its low-voltage version ATmega8535(L), respectively. Pin 9 serves as the active-low reset pin.

The non-volatile program and data memories built into ATmega8535 are:

1. 8 kB of self-programmable flash for storing the software code of the application program.

2. 512 bytes of SRAM, which is a read/write memory.

3. 512 bytes of EEPROM for storing the data. Unlike the flash memory, it can be accessed in a program for writing and reading.

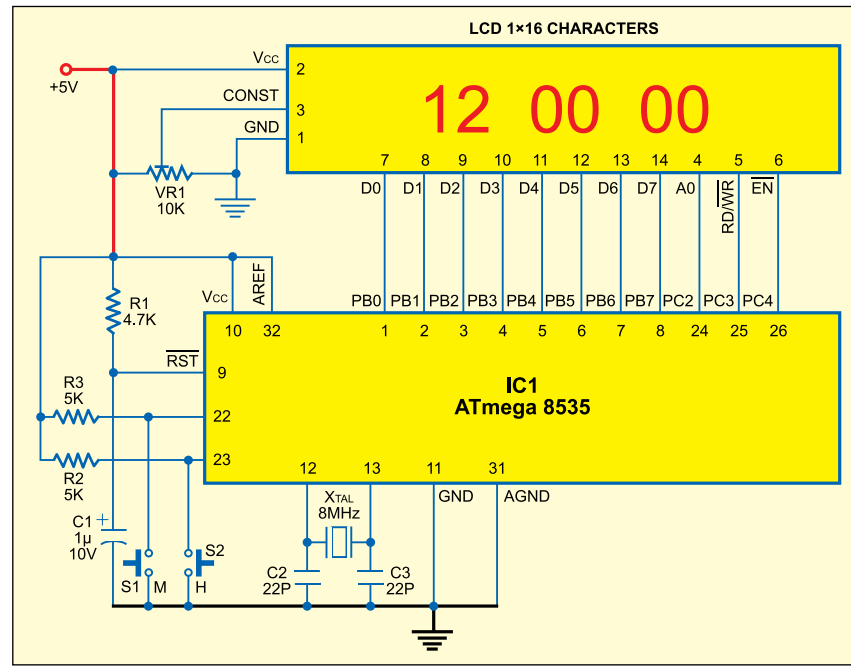

Fig. 11: Circuit diagarm of real-time clock

Programming the on-chip code/program memory

The on-chip flash memory is programmed by pulling up the reset pin and sending data through pins 6 (MOSI) and 7 (MISO), and pin 8 (SCK), which is used for clocking the the code data into the flash memory. This is accomplished by the host computer by sending appropriate instructions and the code bytes; data verification is done by reading the flash memory and comparing it with the original code data. Writing the lock bits to prevent reading of the code in the chip is accomplished through the instructions and the relevant data.

For using the AVR device, these instructions are built into the AT-PROG program (explained in Part I), which is run on the host PC.

Selection of clock. There are some additional fuse bits, which can be programmed for some extra operational functions. Note that the AVR device, as shipped, is preset to work at 1 MHz with its internal oscillator. If you want to use an external crystal, say, of 8MHz frequency, you have to exercise this option by programming the fuse bits accordingly. A fuse bit is just like a flash code memory location.

The CKSEL fuse bits can be programmed to select the desired crystal. The device clocking options are selectable by Flash Fuse bits as shown in Table II. The clock from the selected source is input to the AVR clock generator and routed to the appropriate modules.

Since the default oscillator is 1MHz, unless we set the CKSEL bits to an appropriate value, the external crystal on pins 12 and 13 will not function for ATmega8535.

Programming the fuse bits. The fuse bit programming option is available on the screen when the AT-PROG is run on the PC. When this option is

TABLE I
Port Description

Port description	Pin Nos.	Usage
Port A (PA0-PA7)	40 to 33	Bidirectional I/O pins with 20mA sink capability and active internal pull-ups; alternately used as ADC input as well as data lines to external RAM
Port B (PB0-PB7)	1 to 8	Input or output port, also used for additional functions as T0, T1, AIN0, AIN1, SS, MOSI, MISO and SCK pins
Port C (PC0-PC7)	22 to 29	Used for address output if external RAM is attached; four pins are alternately used as SCL, SDA for I^2C, TOSC1 and TOSC2, respectively
Port D (PD0-PD7)	14 to 21	Bidirectional, as for port A. Also serve as pins for serial communication, interrupts 0 and 1, and PWM 1 and 2 output comparison, etc.

TABLE II
Device Clocking Options Select*

Device clocking option	CKSEL3.0
External crystal/ceramic resonator	1111-1010
External low-frequency crystal	1001
External RC oscillator	1000-0101
Calibrated internal RC oscillator	0100-001
External clock	0000

**For all fuses, '1' means unprogrammed, while '0' means programmed*

Instruction set for ATmega8535

The instruction set comprises several arithmetic, logical, branch and bit-test type instructions. You can download a 150-page user manual for the AVR instruction set from Atmel's site 'www.atmel.com/dyn/resources/prod_documents/doc0856.pdf.' A summary of the instruction set is given on pages 299 through 301 of the Atmega8535(L) datasheet.

Some of the important instructions are given in Table IV.

Points to be noted

1. When the relative call or jump instruction is executed, the entire memory address space can be accessed.

2. During interrupts and subroutine calls, the return address value is stored in the stack space, which is to be defined by the user at the beginning of every program in SRAM space.

3. The 16-bit stack pointer is read-/write-accessible in the I/O space.

4. The 512-byte data RAM is easily accessed through five different addressing modes supported.

5. A flexible interrupt module has its control registers in the I/O space with an additional global interrupt enable bit in the status register. Every interrupt has a separate address for vectoring, where the instruction causing it to jump to the memory area of that particular interrupt has to be kept stored by the programmer.

There are many interrupts available in ATmega8535. In order to use any interrupt, you need to place the address of the program of the respective interrupt service routine at the vector address.

From location '001H' to '014H,' there are 20 such interrupt vector locations in the order of their priority. Address '000H' is used for the reset vector. A reset may be caused by power-on reset, brownout reset and watchdog reset, or externally by making pin 9 low. Table 19 on page 45 of the datasheet lists the details of reset and interrupt vectors.

Note that here we are dealing with word addresses, so each location is actually two bytes long. In this two-byte location, if you place a RETI (return from interrupt) instruction, nothing will be done upon that interrupt. For example, if you place a jump instruction to the required routine, you can

TABLE III
Some Registers in the I/O Space of ATmega8535

DDRB data direction reg. of port B $37	DDRA $3A	DDRC $34	DDRD $31	UDR UART data reg. $2C
PINB input reg. of port B $36	PINA $39	PINC $33	PIND $30	UCSR UART control reg. A=$2B B=$2A
PORTB output reg. port B $38	PORTA#3B	PORTC $35	PORTD$32	UBRR UART baud rate reg.
ADMUX ADC channel sel. $27	ADCSRA ADC control/ status register* $26	ADCH ADC value high reg. $25	ADCL: ADC value low-byte reg. $24	ACSR analogue comparator control/ status reg. ($28)

**ADC in ATmega8535 is named 'ADCSRA'*

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 12: Bit description for status register; I-global interrupt-enable bit, T-T bit copy storage, H-half carry flag, S-sign bit, V-overflow in 2's complement arithmetic, N-negative number flag (2's complement arith.), Z-zero flag, C-carry flag

selected, it pops up a menu of its own. On this menu, you can write the necessary code for CKSEL programming.

Internal registers. Six of the 32 registers can be used as three 16-bit indirect address register pointers for data space addressing, enabling efficient calculations. One of three address pointers (X, Y and Z registers, described under 'register operations' section) is also used for table look-up.

The I/O memory space contains 64 addresses for CPU peripheral functions like control registers, timers/counters and analogue-to-digital converter (ADC). It can be accessed directly or as the data space locations following those of the register files, i.e., after '20H' and up to '5FH.'

The memory space contains important registers for use in interrupt selection, timer control, UART, SPI interface, watchdog and reset selection modes, etc. Table III shows the exact addresses of these I/O registers.

The bit description for the status register (SREG) is shown in Fig. 12.

PARTS LIST

Parts list for real-time clock (Fig. 11)

Semiconductors:
IC1 - ATmega8535

Resistors (all ¼-watt, ±5% carbon):
R1 - 4.7-kilo-ohm
R2, R3 - 5-kilo-ohm
VR1 - 10-kilo-ohm preset

Capacitors:
C1 - 1µF, 10V electrolytic
C2, C3 - 22pF ceramic disk

Miscellaneous:
X_{TAL} - 8MHz
S1, S2 - Push-to-on switch
- 16x1-character Hitachi make LCD or 16×2-character LCD

TABLE IV
Instruction Set for ATmega8535

1. Arithmetic and logical instructions

ADD rd, rs	SUB (rd-rs)	AND rd, rs	OR rd, rs	EOR rd, rs
ADC Add with carry	SBC Subtract with carry	ANDI AND with immediate data	ORI rd, K	COM rd One's complement
ADIW (Word) Add immediate to word	SBCI rd=rd-Carry-K	INC rd Increment reg.	CLR rd Clear register	NEG rd ; 2's complement
	SBIW Subtract immediate from word	DEC rd decrement reg.	SER rd Sets register	SBR rd,n ; sets *nth* bit in rd

Note: *K-immediate data, rd-destination register, rs-source register*

2. Data movement instructions

MOV rd,rs	LDI rd, K Immediate load	ST X, rs Stores rs into [X]	LPM R0 ← [Z] Load from prog. mem.	PUSH rs Pushes to stack the value of register Rs
	LD rd, X* Load indirectly	ST X+, rs ; store indirectly and incr. X	IN rd, port# As PORTB for reg.	POP rs Pops into Rs from stack
	LD rd, X+ Indirect, post incr. address	ST –X, rs Decr. after storing	Out Port, rd	
	LD rd, -X Dec. X by 1 and then read indirectly	STD Z+disp, rs Stores indirect with added displacement	STS k, rs Stores direct to SRAM addr. k	

**X denotes register pair R26-R27. Likewise, Y and Z are also usable for these instructions. # Port B should be entered as 'PINB' for inputs for the assembler; The notation PORTB used for output.*

3. Boolean logic BIT based instructions

SBI P, b Sets bit P=addr, bit no.	CBI P, b Clear bit in I/O register	LSL rd Logical left shift	ROL rd Rotate left through carry	SEC Sets carry flag
		LSR rd Logical right shift	ROR rd Rotate right through carry	CLC Clears carry
	CLI Interrupts disabled.	ASR Arithmetic shift right	SWAP rd Swaps nibbles in Rd	SEZ, CLZ Set/clear zero flag
	SEI Global interrupt enable		Bst/BLd reg, b Stores loads a bit in SREG(b) (status register)	NOP No operation

Note. *Bits can be in any I/O register or bits of any register Rd. P means an I/O address register.*

4. Frequently used test and skip as well as jump and call instructions

RJMP k Jumps k (-2047 to +2048)	RCALL k Calls relative	CP Rd, Rs Compares Rd-Rs	BREQ K Branches to K relative if zero flag is set	
LJMP PC ← Z	ICALL Calls to [Z] indirectly	CPSE Rd, Rs Compares, skips the next instruction if equal	BRNE K Does the opposite of the above	
RET Subroutine return		CPI Rd, K Immediate data compared	BRCS k Branch if carry is set	
RETI Interrupt service return			BRCC k Branch if carry not set	

Note. Most of these instructions are relative branching, except some, which need a full address of destination for jumping.

5. Additional multiply instructions

MUL	Rd, Rr	Multiply unsigned	R1:R0←Rd×Rr
MULS	Rd, Rr	Multiply signed	R1:R0←Rd×Rr
MULSU	Rd, Rr	Multiply signed with unsigned	R1:R0←Rd×Rr
FMUL	Rd, Rr	Fractional multiply unsigned	R1:R0←(Rd×Rr)<<1
FMULS	Rd, Rr	Fractional multiply signed	R1:R0←(Rd×Rr)<<1
FMULSU	Rd, Rr	Fractional multiply signed with unsigned	R1:R0←(Rd×Rr)<<1

write:

```
rjmp timer_routine
```

Then you can use that interrupt to jump to the timer_routine.

As mentioned above, the interrupt vectors follow the reset address at '000H,' wherein a jump instruction to the corresponding actual memory addresses, defined by the labels, is placed. For example, the instruction:

```
RJMP Int0
```

It means that the external interrupt

Bit	7	6	5	4	3	2	1	0
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Fig. 13: Bit details for TCCR0 register; bits 0, 1 and 2 are defined in Table IV reproduced from the original datasheet

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 14: Bit details for TIMSK register

TABLE V
Clock-Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (timer/counter stopped)
0	0	1	$clk_{I/O}$/(no prescalling)
0	1	0	$clk_{I/O}$/8 (from prescaler)
0	1	1	$clk_{I/O}$/64 (from prescaler)
1	0	0	$clk_{I/O}$/256 (from prescaler)
1	0	1	$clk_{I/O}$/1024 (from prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge
1	1	1	External clock source on T0 pin. Clock on rising edge

routine has the label 'Int0,' to which the processor jumps upon pin 17 getting a high logic signal. Also, at the label 'Int0,' if a simple return instruction is entered as:

```
Int0:   reti
```

This instruction simply ignores such an interrupt and returns to the main program. In case you need to process the interrupt, enter the necessary code starting at label 'Int0.'

After the interrupt processing instructions, the various subroutines are entered. Then comes the main program. In the main program, the first thing to write is the stack initialisation instructions. Here, the stack pointer is set to the highest end of the internal RAM, for which a temporary register is used, and the high and low addresses are written to the stack pointer using an instruction at the Ext_Int0 vector address (0x001) such as:

```
ldi temp,low(RAMEND)
out spl,temp
ldi temp,high(RAMEND)
out sph,temp
```

Register operations. Each register is assigned a data memory address, mapping it directly into the first 32 locations of the data space. Register pairs R26-R27, R28-R29 and R30-R31 serve as 16-bit registers, which are used for indirect addressing of the data memory space. These three 16-bit registers are known as 'X' (R27:R26), 'Y' (R29:R28) and 'Z' (R31:R30) registers, respectively. The last 16 registers in the register file (R16 through R31) cannot be used with the first 16 registers (R0 through R15).

The operating instructions for registers have direct and single-cycle access to the registers. The following instructions—constant arithmetic instructions—use the second half of the registers in the register file and cannot be used with the first half:

```
sbci, subi, cpi, andi, ori and ldi
```

The following general instructions that use two registers or only a single register can use the entire register file:

```
Sbc, sub, cp and & or
```

Embedded control functions and their applications

Here we'll use the following four functions of ATMega8535 for typical control applications:

1. Timers; two 8-bit and one 16-bit with add-on features
2. Pulse-width modulated output
3. Analogue-to-digital converter
4. Serial RS-232 interface

Timers and their applications. Both timer 0 and timer 1 are 8-bit timers, while timer 1 is a 16-bit timer. The clock inputs to the timers can have a variety of selections. The CPU clock itself, divided by a prescaling divider with divisors of 8, 64, 256 and 1024, can be chosen. Further, it can also count an externally applied clock at T1 pin (for timer 1).

We shall use these timers for developing a real-time clock with time display on the LCD (see Fig. 1). For the purpose, the registers to be used in timer 0 are:

1. TCCR0: Timer counter control register 0
2. TIMSK: Timer interrupt mask register

TCCR0 register bits. Fig. 13 shows the bit details for TCCR0 register. Bits WGM01, COM00, COM01, WGM00 and FOC0 (bits 3 through 7) of TCCRO register are used with the timer-based comparators for waveform and pulsewidth-modulated output generation. Since these bits are not required for the normal timing operation of the timer, they have not been used here.

The timer clock is selected by using the remaining three bits (CS00, CS01 and CS02). We will set these bits to '0,' '1' and '1,' respectively, for dividing the 1MHz default internal clock of ATmega8535 k (with no external crystal) by '64.' This division gives 65 microseconds per clock. Then we accumulate the counts for getting one second and divide it by '60' to get minutes and again by '60' to get hours, which are counted up to '12' and the process is repeated.

TIMSK register bits. Fig. 14 shows the bit details for TIMSK register. Bit 0 refers to 'timer-overflow interrupt enable.' It must be set to enable the interrupt action on overflow. The TIMO_OVF interrupt ($0009 address) is used to direct a vector at this ad-

dress to the respective interrupt service routine, where we will perform the relevant action that is needed upon timer-0 overflowing, i.e., when the number in its TCNT0 register (timer counter 0) crosses '255' (decimal). So in the software program for real-time clock, we initialise TIMSK to '01.'

Software program for real-time clock (8535clk.asm). The 8535clk.asm program with suitable explanations and comments is given at the end of this article. The programmed IC can be fixed to the RTC circuit board to show real-time clock on the LCD.

8535CLK.ASM

```
; Things to learn here:
; - Timer in interrupt mode
; - Interrupts, Interrupt-vector
; - BCD-arithmetic
.LIST
.NOLIST
.INCLUDE "m8535def.inc"
;device =ATMEga8535
.LIST
; Universal register definition
.DEF        mp = R16
.DEF    result=R18
; Counter for timer timeouts, MSB timer driven by software
.DEF        z1 = R0
; Working register for the Interrupt-Service-Routine
; Note that any registers used during an interrupt, including the status-register with all the flags must either be reserved for that purpose
; or they have to be reset to their initial
; value at the end of the service routine! Otherwise
; nearly unpredictable results will occur.
.DEF        ri = R1
; Register for counting the seconds as packed BCD
.DEF        sec = R2
.DEF    min = R3
.DEF    hour=R4
.DEF   count=R5
.DEF count1= R6
.CSEG
.ORG $0000
; Reset- and Interrupt-vectors
      rjmp Start ; Reset-vector
.org ovf0Addr
      rjmp tc0i
; Reset-vector to address 0000
.org $30
start:  rjmp                main
; Be sure that the jump
; to the interrupt service routine tc0i is exactly at the adress "ovf0",otherwise the interrupt fails.
; The following  sequence takes place : If the timer overflows
; (transition from 255 to 0) the program run is interrupted, the current address in the program counter
; is pushed to the stack, the instruction at address ovf0
; is executed (the  jump instruction). After finishing execution of the interrupt service routine
; the program counter value is restored from the
; stack and program execution proceeds from that point.
tc0i:
       in  ri,SREG ; save the content of the flag register
   inc z1 ; increment the software counter
      out    SREG,ri ; restore the initial value of the flag register
                reti ; Return from interrupt
.org $50
; The main program starts here
main:
  ldi mp,LOW(RAMEND) ;Initiate Stackpointer
  out SPL,mp ; for the use by interrupts and subroutines
  ldi mp,HIGH(RAMEND)
  out SPH,mp       ; Port b (pin 1-8) output-port, port c all output except bit 0,1
  ldi mp,0xFF ; all bits are output
  out DDRb,mp ; to data direction register
  ldi mp,0xFc  ;bits 0,1 input;  pin22 for minutes set; pin23 hour set;
  out  DDRC,mp
  LDI MP,03
  OUT PORTC,MP   ;pull up port C bits 0- 1
                internally itself
  rcall init_lcd          ;initialise LCD module
  ldi R16,$80
  rcall cmd
; Software-Counter-Register reset to zero
  ldi mp,0 ; z1 cannot be set to a constant value, so we set mp
  mov z1,mp ; to zero and copy that to R0=z1
  mov sec,mp ; and set the seconds to zero
  mov min,mp    ;and minutes also
  ldi  mp,$12
  mov hour,mp
; Prescaler of the counter/timer = 64, that is 1 MHz/64 = 15625 Hz = $3D09
  ldi mp,0x03 ;Initiate Timer/Counter 0 Prescaler as /64
  out TCCR0,mp ; to Timer 0 Control Register
; enable interrupts for timer 0
  ldi mp,$01 ; set Bit 0 but for 8515 this was bit  1!
  out TIMSK,mp ; in the Timer Interupt Mask Register
                ; enable all interrupts generally
   sei ; enable all interrupts by setting the flag in the status-register
; The 8-bit counter overflows from time to time and the interrupt service
; routine increments a counter in a register. The main program loop reads this
; counter register and waits until it reaches 3D hex. Then the timer is read until
; it reaches 09 (one second = 15625 (dec)= 3D09(hex) timer pulses). The timer
; and the register are then set to zero and one second is incremented. The seconds
; are handled as packed BCD-digits (one digit = four bits, one byte represents
; two digits). The seconds are refreshed. The seconds
; are displayed on the LCD module, as well.
 ldi mp,0x31     ;just show a "1" to begin with
        rcall lcdwr
 loop:
  ldi mp,$3D ; compare value for register counter
 loop1: rcall lookupdate ; check if user adjusts time-minutes
  rcall lookupdatehr ; check if user adjusts time-hours
  cp z1,mp ; compare with the register
 brlt loop1 ; z1 < mp, wait
 loop2:
    in mp,TCNT0 ; read LSB in the hardware counter
  cpi mp,$09 ; compare with the target value
  brlt loop2 ; TCNT0 < 09, wait
  ldi mp,0 ; set register zero and ...
  out TCNT0,mp ; reset hardware-counter LSB
  mov z1,mp ; and software-counter MSB
  rcall IncSec ; call the subroutine to increment the seconds
  rcall Display ; call subroutine to display the seconds
  rjmp loop ; once again the same
; subroutine increment second counter
; in BCD-arithmetic! Lower nibble = Bit 0..3, upper nibble = 4..7
 IncSec:
  sec ; Set Carry-Flag for adding an additional one to the seconds
  ldi mp,6 ; povoke overflow of the lower nibble by adding 6
  adc sec,mp ; add 6 + 1 (Carry)
  brhs Chk60 ; if overflow of the lower nibble occurred go to 60 check
  sub sec,mp ; subtract the additional 6 as no overflow occurred
 Chk60:
  ldi mp,$60 ; 60 seconds already reached?
  cp sec,mp
  brlt SecRet ; jump if less than 60
  ldi mp,256-$60 ; Load mp to add sec to zero
  add sec,mp ; Add mp to reset sec to zero
  rcall incmin
 SecRet:
  ret ; return to the main program loop
 incmin:  ;subroutine for minutes incrementing
  sec ; Setze Carry-Flag for adding an additional one to the seconds
  ldi mp,6 ; provoke overflow of the lower nibble by adding 6
  adc min,mp ; add 6 + 1 (Carry)
  brhs Chk60_m ; if overflow of the lower nibble occurred go to 60 check
  sub min,mp ; subtract the additional 6 as no overflow occurred
 Chk60_m:
  ldi   mp,$60 ; 60 minutes already reached?
  cp min,mp
  brlt minRet ; jump if less than 60
  ldi mp,256-$60 ; Load mp to add min to zero
  add min,mp ; Add mp to reset min to zero
      RCALL INCHOUR
 minRet:
  ret ; return to the main program loop
 INCHOUR:
  sec
  ldi mp,6
  adc hour,mp
  brhs chk12hour
  sub   hour,mp
  chk12hour:
  ldi mp,$13
  cp hour,mp
  brlt houret
  ldi mp,256-$12
  add hour,mp
  houret:      ret
  lookupdate:
  k2:    sbic pinc,0
  ret  ;if key is not closed, return
        ;if closed, wait for key-debounce and check again
  rcall delay1
  Inc count1
  ldi mp,80
  cp count1,mp
  brlt dd
 RCALL incmin
 Ldi mp,0
 Mov count1,mp
 dd:  rcall display
       ret
   lookupdatehr:
 k3:    sbic pinc,1
 ret  ;if key is not closed, return
      ;if closed, wait for key-debounce and check again
  rcall delay1
  inc count
  ldi mp,80
  cp count, mp
  brlt  dd
      RCALL inchour
  Ldi mp,0
  mov count, mp
  rjmp dd
; subroutine for displaying the time on the LCD
 Display: push r16
  ldi r16,$80
  rcall cmd
  pop r16
  mov r16,hour
  andi r16,0xf0
  ror r16
  ror r16
  ror r16
  ror r16
  ori r16,0x30
```

```
rcall lcdwr
mov r16,hour
andi r16,0b00001111
ori r16,0x30
rcall lcdwr
ldi r16,$3A
rcall lcdwr
mov r16,min
andi r16,0xf0
ror r16
ror r16
ror r16
ror r16
ori r16,0x30
rcall lcdwr
mov r16,min
andi r16,0b00001111
ori r16,0x30
rcall lcdwr
ldi r16,$3A ; For : display
rcall lcdwr
mov r16,sec
andi r16,0xf0
ror r16
ror r16
ror r16
ror r16
ori r16,0x30
rcall lcdwr
```

```
mov r16,sec
andi r16,0b00001111
ori r16,0x30
rcall lcdwr
ldi r16,32
rcall lcdwr
ret
 cmd: cbi portc,2 ; command entry to LCD subroutine
cbi portc,3
cbi portc,4
out portb,R16
sbi portc,4
nop
nop
nop
nop
nop
cbi portc,4
rcall delay1
ret
lcdwr: cbi portc,2 ; write to LCD routine
cbi portc,3
cbi portc,4
sbi portc,2
out portb,R16
sbi portc,4
nop
nop
nop
```

```
nop
nop
cbi portc,4
rcall delay1
ret
init_lcd: ; initialise LCD module
ldi R16,$38
rcall cmd
rcall delay1
rcall delay1
ldi R16,$0e
rcall cmd
rcall delay1
ldi R16,6
rcall cmd
ldi r16,1
rcall cmd
rcall delay1
ret
delay10:
ldi R16,$f0
del_lp: inc R16
brne del_lp
ret
delay1:
clr result
loop22: inc result
brne loop22
ret
```

Let's now examine the use of inbuilt functions of AVR ATmega8535 (such as output compare, ADC and UART) for various applications.

PWM operation of ATmega8535

When the AVR is configured for pulse-width modulated (PWM) operation, the PWM outputs become available at output-compare pins 18 (OC1A) and 19 (OC1B) of ATmega8535. PWM, in conjunction with an analogue filter, can be used to generate analogue output signals and thus it serves as a digital-to-analogue converter.

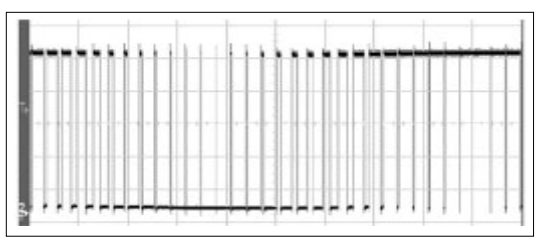

Fig. 15: Variation of pulse width (constant period) with time of a typical PWM wave

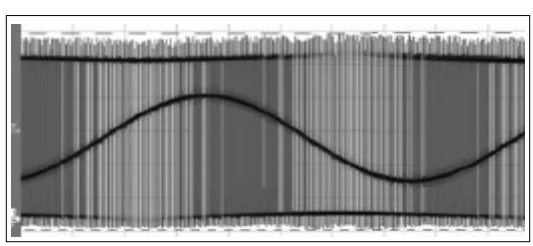

Fig. 16: View of filtered low-frequency sine wave and unfiltered PWM output on an oscilloscope

Principle of pulse-width modulation. To generate different analogue levels, the duty cycle and thereby the pulse-width of the digital signal (base frequency) is changed. If a high analogue level is needed, the pulse width is increased and vice versa (see Figs 15 and 16).

A digital pulse train with a constant period (fixed base frequency) is used as the basis. The base frequency, which can be programmed suitably, should be much higher than the frequency of the output analogue signal obtained after filtering out the base frequency component. For example, to generate a sinewave signal of low frequency (say, 10 Hz, as used for drives or controls), the base frequency of rectangular pulses (with varying duty cycle) may be of the order of 1 kHz or more.

Pulse generation method. The scheme for pulse generation is as follows: Timer/counter 1 is used to count clock ticks. If 8-bit PWM is selected, after the timer counts up to '255,' its count is decremented with each clock tick. Thus, the number increases up to '255' and then decreases, resembling a triangular pattern.

When the number stored in the output-compare register (OCR) matches the loaded count value, pin 19 (output-compare action pin) becomes high or low, as programmed. For example, if the OCR is loaded with a value of '100,' the logic state of OCR pin will be:

Count value	OCR pin
0 to 100	Low logic
100 to 255	High logic
255 to 100	High logic
100 to 0	Low logic

Thus, for the total time taken to count 255×2=510 clock ticks, the output pin (pin 19) will be high for (2×155)/(2×256) or 60.5 per cent of the total triangular wave time of one PWM pulse (or the PWM pulse will have a duty cycle of 60.5 per cent). Thus, effectively the voltage transmitted in this period is 60.5 per cent of the maximum, because the pulse is high only for this period of time.

The following program (AVRSINE.ASM) will generate a 1Hz sine wave (after filtering) on pin 19 using PWM:

```
AVRSINE.ASM
;---------------------
; File: avrsine.asm
; Description: Example of how to use the fast PWM
; of the Avr to generate "sine-wave" signal. The PWM
; output requires filtering to shape the sine wave
; form.
;---------------------
.include "m8535def.inc"
```

```
rjmp init

;Interrupt vector table

.org OVF1Addr ;OC1Aaddr
; Interrupt vector for timer1 output compare match A
rjmp TOF_isr

;Main code
init:
            ldi  R16,low(RAMEND)
; Load low byte address of end
of RAM into register R16
            out SPL,R16
; Initialize stack pointer to end of internal RAM
            ldi  R16,high(RAMEND)
; Load high byte address of end of RAM into
register R16
            out  SPH, R16
; Initialize high byte of stack pointer to end of
internal RAM
                    ldi r16,$ff
                    out ddrb,r16
                    ldi r16,$55
                    out portb,r16

            ldi r16, (1<<PD5) ; Set Pd5 as output
            out DDRd, r16     ;since that is the PWM
                                output pin 19

;SELECT CLOCK SOURCE VIA TCCR1B
       LDI R16,$81


 ;8 BIT PWM NON-INV.
; Set PWM mode: toggle OC1A on compare
            out TCCR1A, r16
; Enable PWM

            ldi r16, 0xFF

; Set PWM top value: OCR1C = 0xFF
            out OCR1AL, r16
             LDI R16,0
             OUT OCR1AH, R16

            ; Enable Timer/Set PWM clock prescaler
            LDI  R16,02
            OUT TCCR1B,R16     ;ck/8 as pwm clock
            (1MHz/8 = 125 kHz)

            ldi r16, (1<<TOIE1)      ; Enable Timer1
Ovrflow interrupt
            out TIMSK, r16
            clr r17
            clr r18
            sei
; Enable global interrupts
idle:
            ldi r16, (1<<SE)         ; Enable sleep
            out MCUCR, r16
            sleep
            rjmp idle

TOF_isr:
            ldi ZH, high(sine_table*2)
; Set up Z to point to the beginning of
sine_table
            ldi ZL, low(sine_table*2)
            add ZL, r17
; Offset Z by r18:r17
            adc ZH, r18
            lpm
            ; Load sine_table[Z] into OCR1A
            out OCR1AL, r0
            inc r17

            reti

sine_table:        ;   256 values
.db 128,131,134,137,140,144,147,150,153,156,159,162,
165,168,171,174
.db 177,179,182,185,188,191,193,196,199,201,204,206,
209,211,213,216
.db 218,220,222,224,226,228,230,232,234,235,237,239,
240,241,243,244
.db 245,246,248,249,250,250,251,252,253,253,254,254,
254,254,254,254
.db 254,254,254,254,254,254,254,253,253,252,251,250,
250,249,248,246
.db 245,244,243,241,240,239,237,235,234,232,230,228,
226,224,222,220
.db 218,216,213,211,209,206,204,201,199,196,193,191,
188,185,182,179
.db 177,174,171,168,165,162,159,156,153,150,147,144,
140,137,134,131
.db 128,125,122,119,116,112,109,106,103,100,97,94,91,
88,85,82
```

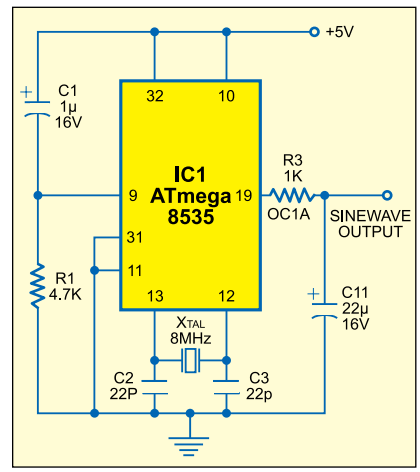

Fig. 17: Circuit for PWM-based sinewave generation

Fig. 18: ADCH and ADCL registers

```
.db 79,77,74,71,68,65,63,60,57,55,52,50,47,45,43,40
.db 38,36,34,32,30,28,26,24,22,21,19,17,16,15,13,12
.db 11,10,8,7,6,6,5,4,3,3,2,2,2,1,1,1
.db 1,1,1,1,2,2,2,3,3,4,5,6,6,7,8,10
.db 11,12,13,15,16,17,19,21,22,24,26,28,30,32,34,36
.db 38,40,43,45,47,50,52,55,57,60,63,65,68,71,74,77
.db 79,82,85,88,91,94,97,100,103,106,109,112,116,119,
122,125
```

For observation of the sine wave on an oscilloscope, use a low-pass filter comprising a 1-kilo-ohm resistor (series element) and a 1µF capacitor (shunt element). However, with an analogue multimeter, the sine wave can be directly observed at pin 19.

In the AVRSINE.ASM program, for each of the triangle wave periods, we read a table of sine values (multiplied by '256') and load these values one by one into the OCR. Since the values vary in a sinusoidal pattern, the pulses that come out are also pulse-width modulated as per these values (see the oscilloscope pattern shown in Fig. 16).

To do the table look-up (as given in the example program 'LCD Table. ASM' of Part 1), the LPM instruction is used. The Z register is used as an indirect indexed register. As stated earlier, the LPM instruction fetches from the table one byte into r_0. The actual loading of the OCR value is done by the instruction:

```
out OCR1AL, r0
```

where OCR1AL refers to pin 19. (OCR1BL refers to pin 18, which is not used here.)

The program contains suitable comments for easy understanding. The table in the program has 256 elements (corresponding to the samples in one complete sinewave period), while each sample period = pulse period (high and low parts) = 510 clock ticks. Thus 256 (samples) × 510 (clock ticks) = 130,560 clock ticks will produce one sinewave cycle. Thus for producing exactly 1Hz frequency, the base frequency should be 130.56 kHz (the nearest value of 125 kHz has been used here).

The circuit for realising the PWM-based sinewave generator is shown in Fig. 17.

The AVRSINE.ASM file and the assembled .HEX file are given in the CD. Using the AT-PROG programmer, load the program into an ATmega8535. Then fix it in a breadboard and make connections as per Fig. 17. Connect the circuit to 5V power supply and observe approximately 1Hz sine wave at pin 19 using an analogue multimeter. The needle on the multimeter will move with the sine wave as a pendulum.

PARTS LIST

Parts list for Figs 17 and 21

Semiconductors:

IC1	-	ATmega8535 microcontroller
IC2	-	LM35 temperature sensor
IC3	-	Max 232, RS-232 level converter

Resistors (all ¼-watt, ±5% carbon):

R1	-	4.7-kilo-ohm
R2	-	100-ohm
R3	-	1-kilo-ohm
VR1, VR3	-	10-kilo-ohm preset
VR2	-	10-kilo-ohm trim potentiometer

Capacitors:

C1, C6	-	1µF, 10V electrolytic
C2, C3	-	22pF ceramic disk
C4, C5, C7-C10	-	10µF, 16V electrolytic
C11	-	22µF, 16V electrolytic

Miscellaneous:

X_{TAL}	-	8MHz crystal
L1	-	100µH inductors
	-	16x1-character Hitachi make LCD or 16×2-character LCD
	-	9-pin female D-connector

Parts list for power supply

Semiconductors:

IC4	-	7805 regulator
D1-D4	-	1N4007 rectifier diode

Capacitors:

C12	-	4700µF, 16V electrolytic
C13	-	0.1µF ceramic disk

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 19: ADMUX register bits

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 20: ADCSRA register bits

Using the ADC

The inbuilt analogue-to-digital converter (ADC) of ATmega8535 is an 8-channel device with 10-bit resolution and maximum conversion time of 65 µs. The reference voltage for the ADC is connected across pins 32 (positive) and 31 (ground). The 5V Vcc supply (either directly or through a potmeter) can be used as reference voltage, but a capacitor at pin 32 is to be used for decoupling.

To access the ADC, you need to select the ADC channel; while the use of ADC interrupt is left to the discretion of the programmer. The ADC is read after conversion of a sample via the ADCH and ADCL registers (8 bits from the ADCL register and only two bits from the ADCH register) as shown in Fig. 18.

ADMUX and ADCSRA are the other registers used in conjunction with the ADC. Functions of various bits of these registers are explained below.

ADMUX register. The ADMUX register bits are shown in Fig. 19.

Bits 4 through 0 of ADMUX select the ADC channels for single-ended or differential operation including channels with gain. (For full selection details, see Table 85 of the ATmega8535(L) datasheet.)

Bit 5 (ADLAR, or AD left adjust result) affects selection of results in ADCH and ADCL registers. If this bit is made '0,' the ADCL contains the least eight bits and the ADCH contains the remaining two high-order bits in its D1:D0 bit positions. When the ADLAR bit is set to '1,' the ADCH contains the most significant eight bits, while the ADCL contains the least two significant bits in bit positions 7 and 6.

Bits 6 and 7 (REFS0 and REFS1) are reference-selection bits. With bit 7 as '0' and bit 6 as '1,' the external reference voltage is applied to pin A_{ref} (32).

We write E0 (1110 0000b) to ADMUX register in the ADC_LCD.ASM program. That means we choose channel-0 (pin 40) for the signal input, ADCH to give us the most significant eight bits and external 5V reference at pin 32 for analogue-to-digital conversion.

ADCSRA register. This is the control-and-status register for the ADC. Its bit positions are shown in Fig. 20.

The bits of the ADCSRA stand for the following signals: ADC enable (bit 7), ADC start (bit 6), ADC auto-trigger enable for free-running (bit 5), ADC interrupt flag set on completion of conversion (bit 4), ADC interrupt enable when set (bit 3) and ADC prescaler for speed (bits 0, 1 and 2). Bits 0, 1 and 2 determine the division factor between the clock frequency and the input clock to the ADC. The division factor can be selected from '2' to '128' as per Table 86 of the datasheet.

Program for displaying the ADC output on the LCD

The following program (ADC_LCD. ASM) takes the ADC data, converts the 10-bit data into five decimal digits and then shows it continuously on the LCD screen:

```
ADC_LCD.ASM
; ************************************************
; *This program uses channel -0 ADC of ATmega8535
; It reads the ADC and outputs the five-digit
; number on LCD.
; Program authored by  Prof. K. Padmanabhan
; ************************************************
.NOLIST
.INCLUDE "m8535def.inc"
 ;device =ATMega8535
.LIST
.EQU xyz = 12345
.EQU   fq=1000000; clock freq. of
internal oscillator
.EQU   baud=9600; Baudrate of SIO comm.
.EQU   bddiv=(fq/(16*baud))-1; Baudrate
divider
.DEF rmpr = R16
.DEF temp = R14
.DEF result=R12
.DEF mpr =R16
.CSEG
.ORG $0000
; Reset- and Interrupt-vectors
rjmp Start ; Reset-vector
.org OVF0Addr  ; timer-0 overflow
interrupt vector
address
rjmp timer0prg
timer0prg:  ;here take ADC sample
at every 64 µs
ldi r16,$cc
      out portc,r16
      push r16
      in r16,SREG
      PUSH R16
here2:in r16,adcsra
      andi r16,0b01000000
      brne here2  ;value got
in  r16,adcl
      in r17,adch
      rcall lcddisp
      POP R16
      out SREG,R16
      POP R16 ;restart adc
      ldi r16,0b11000101 ;prescale /32
(1x32=32 µs)
;adc enable,adc start,adc
freerun,adcflag,adcno int,
adcprescale/32
      out adcsra,r16
      RETI ;End of ISR
 cmd: cbi portc,2 ;command entry to
LCD routine
      cbi portc,3
      cbi portc,4
      out portb,r16
      sbi portc,4
      nop
      nop
      nop
      nop
      nop
      cbi portc,4
      rcall delay1
      ret
lcdwr:cbi portc,2; wrtite to LCD
routine
      cbi portc,3
      cbi portc,4
      sbi portc,2
      out portb,r16
      sbi portc,4
      nop
      nop
      nop
      nop
      nop
      cbi portc,4
      rcall delay1
      ret
busy: cbi portc,2
      sbi portc,3 ;read/write high?
      cbi portc,4 ;chip select low
      nop
      nop
      sbi portc,4      ;chip select high
busy1:lds R16,pinb
      rol R16
      brcs busy1
      cbi portc,4
      ret
init_lcd: ;initialise LCD
      ldi R16,$38
      rcall cmd
      rcall delay1
      rcall delay1
      ldi R16,$0e
      rcall cmd
      rcall delay1
      ldi R16,6
      rcall cmd
      ldi r16,1
      rcall cmd
      rcall delay1
      ret
delay1:clr result
loop22:ldi R16,$f0
loop2:inc R16
```

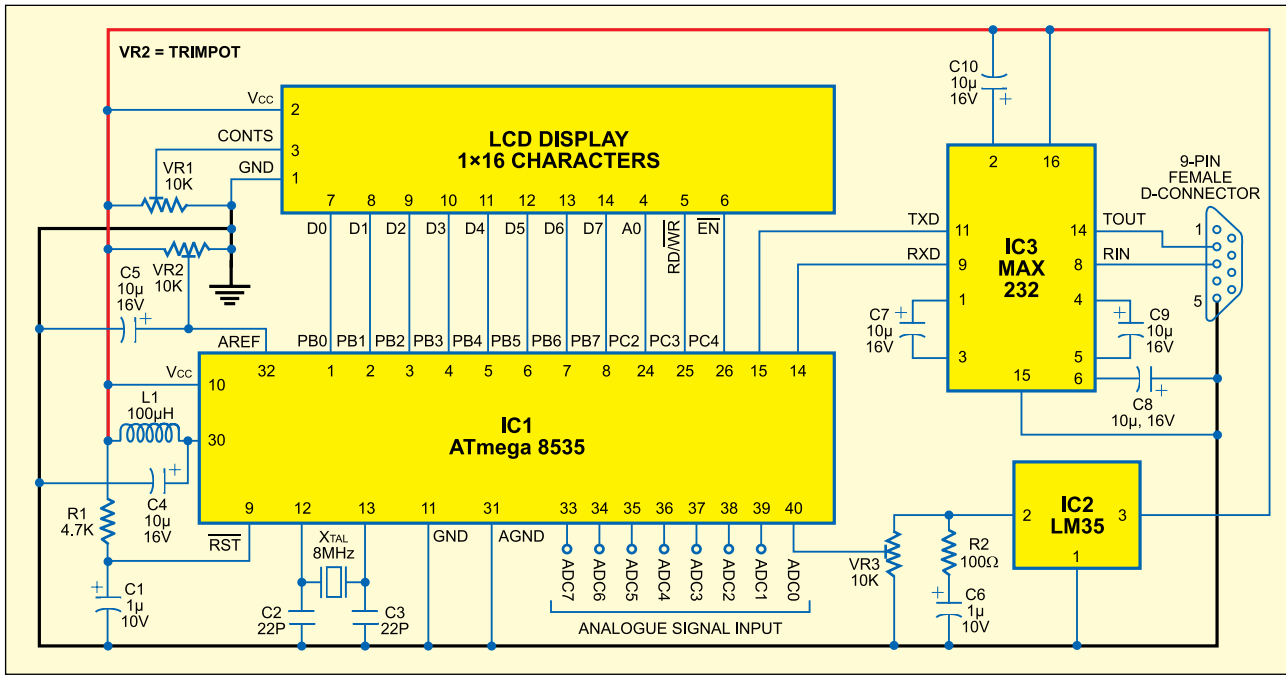

Fig. 21: Circuit for temperature display on either the LCD or the PC

```
      brne loop2
      inc result
      brne loop22
      ret
lcddisp: push r16
      ldi r16,128 ;cursor to left end
      rcall cmd
      pop r16
      rcall binbcd
      mov r16,r15
      andi r16,0x0f
      ori r16,0x30
      rcall lcdwr      ; 1
      mov r16,r14
      andi r16,0b11110000
      ror r16
      ror r16
      ror r16
      ror r16
      ori r16,0x30
      rcall lcdwr      ;2
      mov r16,r14
      andi r16,0x0f
      ori r16,0x30
      rcall lcdwr      ;3
      mov r16,r13
      andi r16,0b11110000
      ror r16
      ror    r16
      ror    r16
      ror      r16
      ori r16,0x30
      rcall lcdwr     ;4
      mov r16,r13
      andi r16,0x0f
      ori r16,0x30
      rcall  lcdwr    ;5
      ret
binbcd:
;* "bin2BCD16" - 16-bit Binary to BCD conversion
;* converts 16-bit number (fbinH:fbinL) to a 5-digit
;* packed BCD number represented by 3 bytes
(tBCD2:tBCD1:tBCD0).
;* MSD of 5-digit number is placed in lowermost
nibble of tBCD2.
;* Number of words       :25
;* Number of cycles      :751/768 (Min/Max)
;* Low registers used    :3 (tBCD0,tBCD1,tBCD2)
;* High registers used  :4(fbinL,fbinH,cnt16a,tmp16a)
;* Pointers used         :Z
Subroutine register variables
.equ        AtBCD0       =13
;address of tBCD0
.equ        AtBCD2       =15
;address of tBCD1
.def        tBCD0        =r13
;BCD value digits 1 and 0
```

```
.def        tBCD1        =r14
;BCD value digits 3 and 2
.def        tBCD2        =r15
;BCD value digit 4
.def        fbinL        =r16
;binary value Low byte
.def        fbinH        =r17
;binary value High byte
.def        cnt16a       =r18
;loop counter
.def        tmp16a       =r19
;temporary value
bin2BCD16:
ldi cnt16a,16            ;Init loop counter
            clr tBCD2
;clear result (3 bytes)
            clr tBCD1
            clr tBCD0
            clr ZH
;clear ZH (not needed for AT90Sxx0x)
bBCDx_1:lsl fbinL ;shift input value
            rol fbinH
;through all bytes
            rol tBCD0                ;
            rol tBCD1
            rol tBCD2
            dec cnt16a
;decrement loop counter
            brne bBCDx_2 ;if counter not zero
            ret                      ;   return

bBCDx_2:ldi  r30,AtBCD2+1
;Z points to result MSB + 1
bBCDx_3: ld tmp16a,-Z    ;get (Z) with
                                      pre-decrement
            subi  tmp16a,-$03        ;add 0x03
            sbrc  tmp16a,3           ;if bit 3 not
                                     clear
            st  Z,tmp16a;store back
            ld  tmp16a,Z;get (Z)
            subi  tmp16a,-$30        ;add 0x30
            sbrc  tmp16a,7           ;if bit 7 not
                                     clear
            st  Z,tmp16a;store back
            cpi  ZL,AtBCD0           ;done all three?
            brne  bBCDx_3            ;loop again if
                                     not
            rjmp  bBCDx_1

; Main program routine starts here
Start:ldi R16,low(RAMEND);Load low byte address
of end of RAM into register R16
            out  SPL,R16; Initialize stack
pointer to end of internal RAM
            ldi  R16,high(RAMEND);Load
high byte address of end of
    RAM into register R16
```

```
out  SPH, R16; Initialize high byte of stack
pointer to end of internal RAM
ldi rmpr,0b00000001;TIMER 0 INTERRUPT ENABLE
out TIMSK,rmpr
    ldi rmpr,05 ; So, we get once 1x10^6/1024=1000 Hz
out TCCR0,rmpr ;prescalar 1024 so that timer
interrupt occurs at 1KHz rate
ldi  r16,$c0      ;c0 for int. ref, e0 with adch
                  alone used.
      out admux,r16  ;channel 0 is selected
      ldi r16,0b11000101 ;prescale /32 (1x32=33 usec)
;adc enable,adc start,adc freerun,adcflag,adcno int,
adcprescale/32
      out adcsra,r16
      ldi r16,0
      out sfior,r16 ;write 0-0-0 to bits d7-d5 for
                  free run
adc
here1:in r16,adcsra
      andi r16,0b01000000
      breq here1  ;value got
      ldi R16,255
      out ddrb,R16   ; port b is all bits output
      out ddrc,R16   ; so is port c
      ldi r16,0
      out ddra,r16         ;port a input
init: sei ;enable global interrupt
LCD:  rcall init_lcd
      ldi R16,$80
      rcall cmd
here3:in r16,adcsra
      andi r16,0b01000000
      brne here3  ;value got
      in  r16,adcl
      in r17,adch
      rcall lcddisp
idle: ldi r16,(1<<SE)
      out mcucr,r16
      sleep
      rjmp idle
restrt:ldi r16,$80    ;point to first cursor
       rcall cmd  ; command to lcd to position cursor
       rcall delay1
       ldi r16,0b11000101 ;prescale /32 (4.43/32=138
usec)=7.2Khz
;adc enable,adc start,adc freerun,adcflag,adcno int,
adcprescale/32
       out adcsra,r16
here4: in r16,adcsra
       andi r16,0b01000000
       brne here4  ;value got
       in  r16,adcl
       in r17,adch
       sbi adcsra,6  ;restart adc
hh:    rcall lcddisp
       RJMP restrt ; Test of the serial interface
```

Note. The ADC_LCD.ASM program together with the .hex file, for directly programming into the chip, is provided in the EFY-CD.

Fig. 21 shows the circuit for viewing the analogue temperature (°C) output of an LM35 temperature sensor IC connected to ADC Ch.0 (pin 40) of the AVR on the LCD screen in 5-digit decimal format after analogue-to-digital conversion using the ATmega8535 chip with the ADC_LCD program. The same circuit with addition of MAX232 chip and ATmega8535 can be used for interfacing to a PC for viewing the temperature data on the PC screen. However, for that you have to program the AVR with the firmware as described in the succeeding paragraphs.

Using the UART in the ATmega8535

Serial communication between the microcontroller and a PC is essential for data transfer to the microcontroller and reading of its ADC output by the PC. The universal asynchronous receiver transmitter (UART) built into the microcontroller can be programmed to operate at certain baud rates.

The ADC_CH.ASM sample program given below is useful for UART applications:

```
ADC_CH.ASM
; ************************************************
; This program read one of ADC channels (0  to 7).
The Channel can be selected by sending Channel
number.
; ATmega8535 receives the Channel no. and outputs
the five digit ADC value
; on RS232 port for reading by a PC's XTALK
program or a VB project.
; Software features:  It is possible to read the ADC
value and also
; transmit to the PC for data logging.
; ************************************************
.NOLIST
.INCLUDE "m8535def.inc"
;device =ATMega8535
.LIST
.EQU xyz = 12345
; Constants for Sio properties
.EQU fq=1000000;  clock frequency of m8535 with
internal oscillator
.EQU baud=4800; Baudrate for SIO communication
.EQU bddiv=(fq/(16*baud))-1; Baudrate divider
.DEF rmpr = R16
.DEF temp = R14
.DEF result=R12
.DEF mpr =R16
.CSEG
.ORG $0000
rjmp Start ; Reset-vector
.org $000b
            rjmp USART_RXC
.org $0100
InitSio:
            LDI rmpr,bddiv ; Init baud generator
            OUT UBRRL,rmpr ; set divider in UART
baud rate register
            ldi rmpr, 0
            out ubrrh,rmpr
            LDI rmpr,(1<<Rxen)|(1<<Txen)|
(1<<RXCIE)
            out UCsRB,rmpr
            LDI ZL,0 ; Wait some time
            LDI ZH,0
InitSio1:
            SBIW ZL,1
            BRNE InitSio1
            ldi r16,(1<<Ursel)|(1<<USBS)|
(3<<UCSZ0)
            out ucsrc,r16
            RET
USART_RXC:
push r16
      in R16,udr
      andi R16,07
      mov r1,r16
      ori r16,$c0
      out admux,r16
      ldi r16,$43
      rcall tout      ; intimate new channel to host
      ldi r16,$48
      rcall tout
      mov r16,r1
      ori r16,$30
      rcall tout
      pop r16
      reti
 cmd: cbi portc,2
      cbi portc,3
      cbi portc,4
      out portb,r16
      sbi portc,4
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      cbi portc,4
      rcall delay1
      rcall delay1
      rcall delay1
      ret
lcdwr:cbi portc,2
      cbi portc,3
      cbi portc,4
      sbi portc,2
      out portb,r16
      sbi portc,4
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      cbi portc,4
      rcall delay1
      ret
init_lcd:
      ldi R16,$38
      rcall cmd
      rcall delay1
      rcall delay1
      ldi R16,$0e
      rcall cmd
      rcall delay1
      ldi R16,6
      rcall cmd
      ldi r16,1
      rcall cmd
      rcall delay2
ret
tout: sbis UcSRa,UDRE ;TX COMPLETE check
      RJMP tout
      OUT UDR,R16
      Ret
delay1:push r16
       clr result
loop22:ldi R16,$f0
loop2:inc R16
      brne loop2
      inc result
      brne loop22
      pop r16
      ret
delay2:push r16
       clr result
loop221:ldi R16,$f0
loop21:inc R16
       brne loop21
       inc result
       brne loop221
       pop r16
       ret
delay: clr result
ld:    inc result
       brne ld
       ret
lcddisp:push r16
       ldi r16,128 ;cursor to left end
       rcall cmd
       pop r16
       rcall delay1
       rcall delay1
       rcall binbcd
       mov r16,r15
       andi r16,0x0f
       ori r16,0x30
       rcall tout
       rcall lcdwr      ; 1
       mov r16,r14
       andi r16,0b11110000
       ror r16
       ror r16
       ror r16
       ror r16
       ori r16,0x30
       rcall tout
       rcall lcdwr      ;2
       mov r16,r14
       andi r16,0x0f
       ori r16,0x30
       rcall tout
       rcall lcdwr      ;3
       mov r16,r13
       andi r16,0b11110000
       ror r16
       ror    r16
       ror    r16
       ror      r16
      ori r16,0x30
      rcall tout
      rcall lcdwr     ;4
      mov r16,r13
      andi r16,0x0f
      ori r16,0x30
      rcall tout
      rcall  lcdwr    ;5
      ldi r16,$0a
      rcall tout
      ldi r16,$0d
      rcall tout
      ret
binbcd:
;* "bin2BCD16" - 16-bit Binary to BCD conversion
;* convert 16-bit number (fbinH:fbinL) to a 5-digit
;* packed BCD number represented by 3 bytes
(tBCD2:tBCD1:tBCD0).
;* MSD of 5-digit number is placed in the lowermost
nibble of tBCD2.
;* Number of words      :25
;* Number of cycles     :751/768 (Min/Max)
;* Low registers used   :3 (tBCD0,tBCD1,tBCD2)
;* High registers used  :4(fbinL,fbinH,cnt16a,tmp16a)
;* Pointers used        :Z

; Subroutine Register Variables
.equ        AtBCD0 =13          ;address of tBCD0
.equ        AtBCD2 =15          ;address of tBCD1
.def        tBCD0   =r13   ;BCD value digits 1 and 0
.def        tBCD1  =r14    ;BCD value digits 3 and 2
.def        tBCD2  =r15     ;BCD value digit 4
.def        fbinL =r16     ;binary value Low byte
.def        fbinH  =r17    ;binary value High byte
.def        cnt16a =r18    ;loop counter
.def        tmp16a =r19   ;temporary value

bin2BCD16:
            ldi  cnt16a,16 ;Init loop counter
            clr  tBCD2  ;clear result (3 bytes)
            clr  tBCD1
            clr  tBCD0
            clr  ZH     ;clear ZH (not
                        needed for AT90Sxx0x)
bBCDx_1:lsl  fbinL      ;shift input value
            rol  fbinH  ;through all bytes
            rol  tBCD0
            rol  tBCD1
            rol  tBCD2
            dec  cnt16a   ;decrement loop counter
            brne  bBCDx_2 ;if counter not zero
            ret  ;   return
bBCDx_2:ldi  r30,AtBCD2+1 ;Z points to
result MSB + 1
bBCDx_3:
            ld tmp16a,-Z ;get (Z) with pre-decrement
```

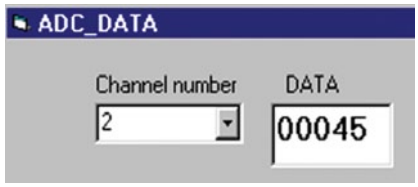

Fig. 22: Screenshot of ADC_CHSEL application

```
          subi  tmp16a,-$03       ;add 0x03
          sbrc  tmp16a,3          ;if bit 3 not
                                  clear
          st  Z,tmp16a;store back

          ld  tmp16a,Z;get (Z)
          subi  tmp16a,-$30       ;add 0x30
          sbrc  tmp16a,7 ;if bit 7 not clear
          st  Z,tmp16a ;          store back
          cpi  ZL,AtBCD0 ;done all three?
          brne  bBCDx_3 ;loop again if not
          rjmp  bBCDx_1 ;End of the subroutine
                              section
```

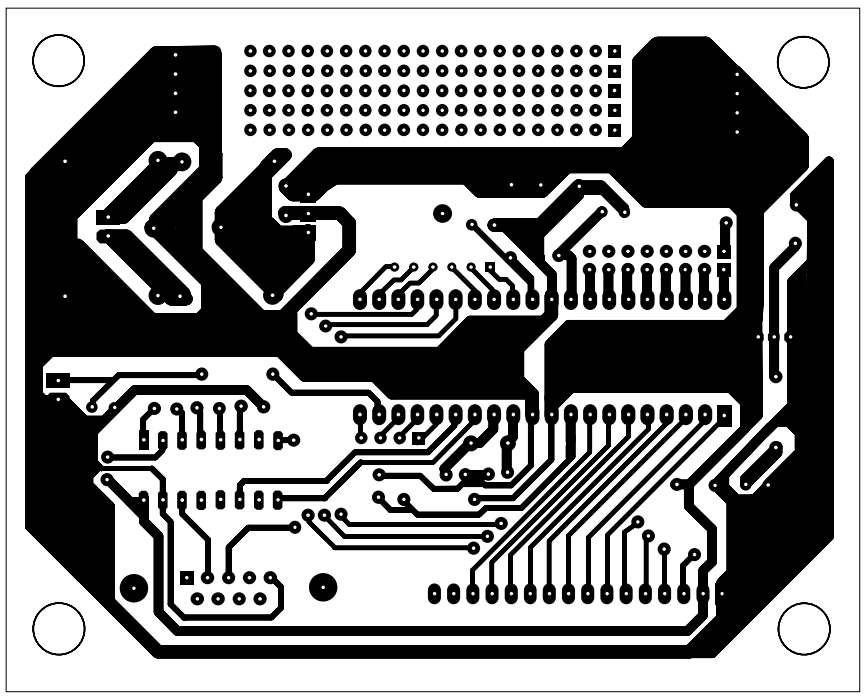

Fig. 23: Integrated actual-size PCB layout (including the 5V power supply circuit given in Part 1) for all the applications described in this 3-part article

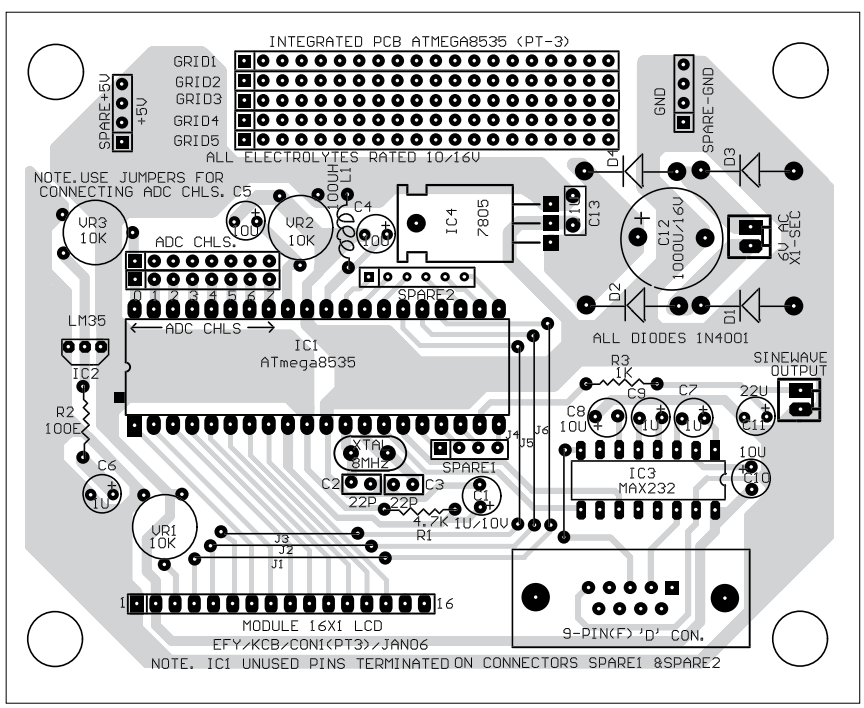

Fig. 24: Component layout for the PCB

```
; Main program routine starts here
Start:ldi R16,low(RAMEND);Load low byte address
                                of end of RAM into
                                 register R16
            out SPL,R16 ; Initialize stack pointer to
                        end of internal RAM
            ldi R16,high(RAMEND);Load high byte
                                      address of end
                                      of RAM into
                                      register R16
            out         SPH, R16 ;Initialize high
                                 byte of stack pointer
                        to end of internal RAM
  ldi  r16,$c0 ; c0 for int. ref, e0 with adch alone
                        used.
      out admux,r16  ; channel 0 is selected
      ldi r16,0b11000101 ;prescale /32 (1x32=33 usec)
    ;adc enable,adc start,adc freerun,adcflag,adcno
            int,
adcprescale/32
      out adcsra,r16
      ldi r16,0
      out sfior,r16 ; write 0-0-0 to bits d7-d5 for
                        free
run adc
here1:in r16,adcsra
      andi r16,0b01000000
      breq here1  ;value got
      ldi R16,255
      out ddrb,R16 ; port b is all bits output
      out ddrc,R16 ; so is port c
      ldi r16,0
      out ddra,r16 ;port a input
init: rcall initsio
      sei  ;enable global interrupt
LCD:  rcall init_lcd
lcd1: ldi R16,$80
      rcall cmd
      rcall delay1
      rcall delay1
      rcall delay1
      rcall delay1
here3:in r16,adcsra
      andi r16,0b01000000
      brne here3  ;value got
      in  r16,adcl
      IN R17,adch
      push r16
       ldi r16,0b11000101 ;prescale /32 (1x32=32
usec)
;adc enable,adc start,adc freerun,adcflag,adcno int,
adcprescale/32
      out adcsra,r16
      pop r16
      rcall lcddisp
        rjmp lcd1
      in r16,udr
      andi r16,07
      mov r14,r16
       ori r16,$c0
      out admux,r16
      ldi r16,$43
      rcall tout      ; intimate new channel to host
      ldi r16,$48
      rcall tout
      mov r16,r14
      ori r16,$30
      rcall tout
      rjmp lcd1
```

The above program reads the ADC output data, whose decimal value is output to pin 15 (TX) of ATmega8535 at 4800 bauds in 8-bit ASCII data format. If a MAX232 is wired to pin 15, it can be directly connected to the receive pin of the RS-232 com port of a PC. Then, by using any terminal program (such as XTALK), it can be received by the PC.

The program (ADC_CH.ASM) may be tested as follows:

1. Wire ATmega8535 to the LCD and the serial port through a MAX232 IC as shown in Fig. 21.

2. Connect an analogue signal (e.g.,

a DC voltage in the 0-5V range tapped from a potmeter or the output of LM35 used in the preceding application) to ADC Ch. 0 (pin 40).

3. Program the adc_ch.asm file into the flash memory of ATmega8535 after compilation.

4. Place the IC on the breadboard and press reset.

5. Connect the RS-232 connector of the PC through a 3-wire cable to the MAX232 pins on the board. The TXD output from ATmega8535 should go to RXD pin of the PC's com port and the PC's TXD output should go to RXD pin of ATmega8535.

6. Run the XTALK program on the PC and set the baud rate as '4800,' data as '8 bits,' parity as 'none,' stop as '1,' and com as '1' or '2,' type 'go low,' then press 'Enter' key.

7. Observe the ADC data continuously on the screen.

The PC terminal program can be used to select one of the eight desired channels. For this, type any number from '0' to '7.' For example, to select channel-3 ADC, type '3.' Remember you need not press Enter key thereafter.

The data from the PC terminal is received by the USART_RX subroutine in the interrupt mode. The main program configures the received data to interrupt the processor. In the interrupt routine, the number sent is used to change (by altering the value of the bits in the ADMUX register of the chip) the ADC channel currently chosen. Thus all the following data will pertain to this channel only and the same will be informed to the PC terminal also by sending CH3 followed by data stream.

The XTALK terminal program is given only for testing purposes. The Visual Basic program (ADC_CHSEL) provided in the EFY-CD of this month does the same. It has two windows, one of which is a Combo box for selecting the channel and the other shows the 5-digit data continuously. Selection of the channel is possible via the Combo box (Fig. 22).

Application notes with programs

You may visit Atmel's Website 'www.atmel.com/dyn/products/app_notes.asp?family_id=607' for the following application notes.

1. AVR100: Accessing the EEPROM. This application note contains assembly routines for accessing the EEPROM for all AVR devices. It includes the code for reading and writing EEPROM addresses sequentially and at random addresses.

2. AVR223: Digital Filters with AVR. This document focuses on the use of the AVR hardware multiplier and general-purpose registers for accumulator functionality, scaling of coefficients when implementing algorithms on fixed-point architectures, actual implementation examples and possible ways to optimise/modify the implementations suggested.

3. AVR240: 4x4 Keypad-Wake Up on Keypress. This application note describes a simple interface to a 4×4 keypad designed for low-power battery operation.

Also there are application notes for interfacing the AVR to an IR detector much like the TV remote. Other topics of interest relating to the AVR are use of watchdog, power idle modes, SPI interfacing for communication, etc. Many have tried out the SPI interface for data communication, but it is found to be more complex compared to the RS-232 protocol. The RS-232 link for ADC data, which is described above, makes a really useful serial data-acquisition system.

An integrated actual-size PCB layout (including the 5V power supply circuit given in Part 1) for all the applications described in this 3-part article is shown in Fig. 23. The component layout for the same is shown in Fig. 24. Suitable pads (not shown in the component layout) have been provided for wiring the components. ●

Download source code: http://www.efymag.com/admin/issuepdf/Application%20AVR%20Part%20II.zip