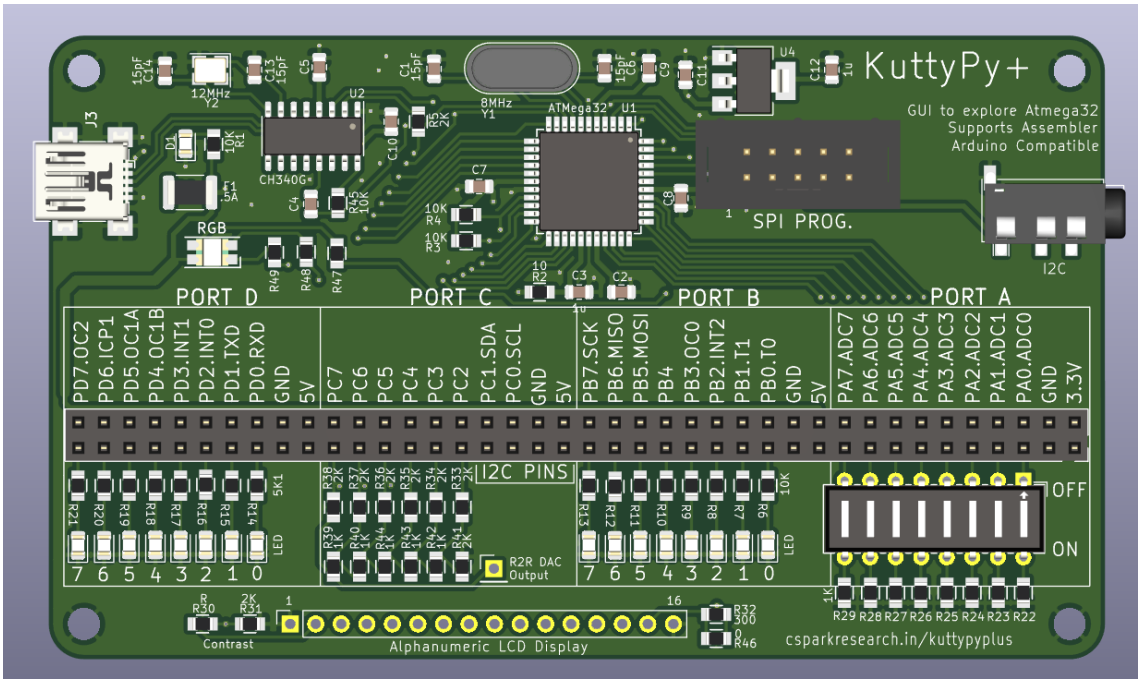
Micro-controller Programming using AtMega32 on KuttyPy



Chapter 1

Introduction

Modern computers perform a variety of tasks like managing our bank accounts, providing entertainment, navigation during our journeys and even playing games like chess. Most of us are amazed by their capabilities and sometimes doubt that they are really intelligent like humans. Are they really intelligent? The only way to answer this question is to explore the internal working of the computers. It appears that computers can represent and manipulate information in the form of text, pictures, audio, video etc. Let us consider the following aspects.

- **Representing Information:** If we consider the storage aspect carefully we can realize that every piece of information can be represented using a sequence of numbers, for example the typed matter you are reading now. Each character is represented by some number (for example in the ASCII code the number 65 represents the character A). In a similar manner a picture can be represented as a collection of pixels, with their positions and colors represented by numbers. So we can come to the conclusion that every type of information can be represented as a sequence of numbers.

- **Storage of Numbers:** We use the decimal system (with base 10) but that is not the only scheme possible. Computers use the binary number system where the base is 2. Under this scheme the possible digits are 0 and 1 only. It is easy to represent them using electronic switches because a switch has two states, closed or open. We can also use two voltage levels to represent 0 and 1. So we can conclude that any piece of information can be stored in the binary format using a sequence of switches. The electronic switches form the memory elements that store the information.

- **Operations:** The third aspect is how to manipulate the information stored as binary numbers. The electronic circuits inside the micro-processor can perform arithmetic and logical operations (adding, subtracting, comparing etc.) only.

- **Sequencing of Operations:** The final requirement is to perform these operations in a predefined sequence, like add two numbers and then do some other operation depending on the result. This sequence of operations also can be stored as numbers, called machine language instructions. The logic circuits inside a microprocessor is capable of bringing the sequence of instructions stored in the memory and perform the specified operations.

To illustrate this process let us consider an example, without using any computer. We need to generate the multiplication table of a number, by using four scratchpads (A,B,C

and D, where we can write one number at a time) and some mechanism to copy or add a number from one scratchpad to another.

- Task: Generate the multiplication table of 5 (1 to 10)
- Resources:
  - Scratchpads A,B,C and D where you can wrtite one number at a time.
  - Ability to Move, Add and Compare numbers.
- Procedure:
  - Write zero to A
  - Write zero to B
  - Write 5 to C
  - Add C to B
  - Send B to the output
  - Increment A
  - Compare A to 10
  - Go to step 4 if A is smaller

The steps given above can be represented using some symbols as shown below.

0	CLR A
1	CLR B
2	MOV C
3	5
4	ADD C,B
5	OUT B
6	INC A
7	CMP A
8	10
9	JLE
10	4

This process can also be done using logic gate circuits. The instructions like CLR, ADD etc. can also be represented using numbers, called Op-Codes. We can see that there are 11 instructions in total. They can be stored in a sequence of memory locations and brought in one by one for execution. The scratchpad D can be used for keeping track of the memory location from which the instruction should be brought, a Program Counter.

1.1 The micro-processor

This stored-program concept was first proposed by John von Neumann and others in 1945, which is known as the von Neumann architecture. All the modern computers still follow the same architecture but the speed is getting increased by processing more instructions per second, by increasing the width of the registers and memory, performing many operations in parallel etc.

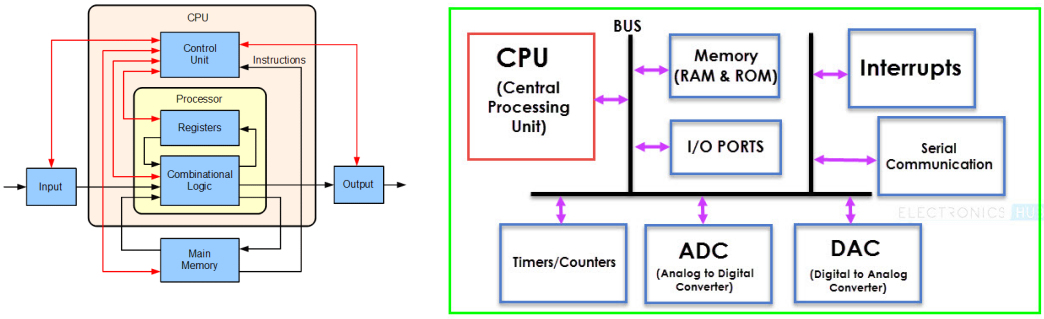


Figure 1.1: a)Micro-processor block diagram. b)Micro-controller block diagram

A micro-processor (also called CPU or Central Processing Unit) requires external memory and Input/Output systems to form a computer. *A micro-controller is a micro-processor combined with program and data memory, peripherals like analog to digital converters, timer/counters, serial communication ports and general purpose Input/Output ports, etc. on a single integrated circuit.* Tasks requiring smaller amounts of processing power and memory are generally implemented using micro-controllers (uC). They are mostly embedded in equipment such as automobiles, telephones, home appliances, and peripherals for computer systems. Block diagrams of a micro-processor and a micro-controller are shown in figure 1.1.

Chapter 2

AVR series Micro-controllers

There are plenty of AVR micro-controller development kits in the market, like Arduino. Most of them focus on explaining the hardware and software of the development kit rather than the micro-controller. They teach programming the I/O pins of the development board using the library functions provided and the user can get things done without understanding anything about the micro-controller. The objective of this work is to help learning uC architecture and programming, not the development board. The focus will be on the features of the micro-controller without hiding its details from the user.

Intel 8051, Atmel AVR, PIC etc. are popular micro controllers available in the market. We have chosen ATmega32 micro-controller from Atmel AVR series, after considering the hardware resources available on it and the support of Free Software tools like GNU assembler and C compiler. A block diagram of an AVR series micro-controller is shown in figure 2.1. The micro-processor sections are outlined in blue.

- R0 to R31, 32 numbers of 8 bit wide General Purpose Registers.
- The Program Counter PC, instruction to execute is brought from the memory location specified by the PC.
- Status and Control Register SREG, updated after every ALU operation.
- The Stack Pointer Register. SPL and SPH are combined to form a 16 bit address. Used by PUSH and POP instructions.

The peripheral devices like Input/output ports, ADCs etc. are controlled using Special Function Registers (SFR). To really understand the working of a microprocessor/microcontroller you need to write programs in Assembly language, by manipulating the contents of various Registers and Memory locations.

2.1 AVR Architecture

A schematic of the AVR architecture is shown in figure 2.1. The 32 General Purpose Registers (R1 to R31, 8 bit wide) are also called the Register File. Data is moved between the Registers and the memory. Addressing memory locations above 255 is done by combining two 8bit registers to form a 16 bit register.

The registers R26 to R31 has some special properties to support indirect addressing requiring a 16 bit address. They are paired to support 16 bit addreessing as shown in the figure 2.3. R26 and R27 combined is the X register, R28 with R29 is the Y register,

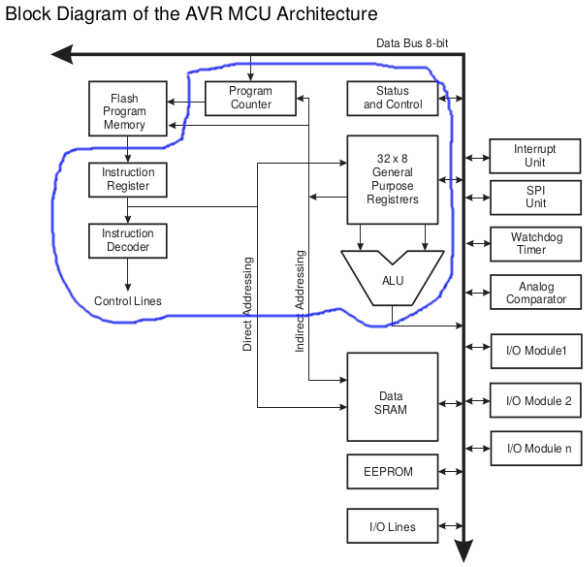


Figure 2.1: AVR series microcontroller

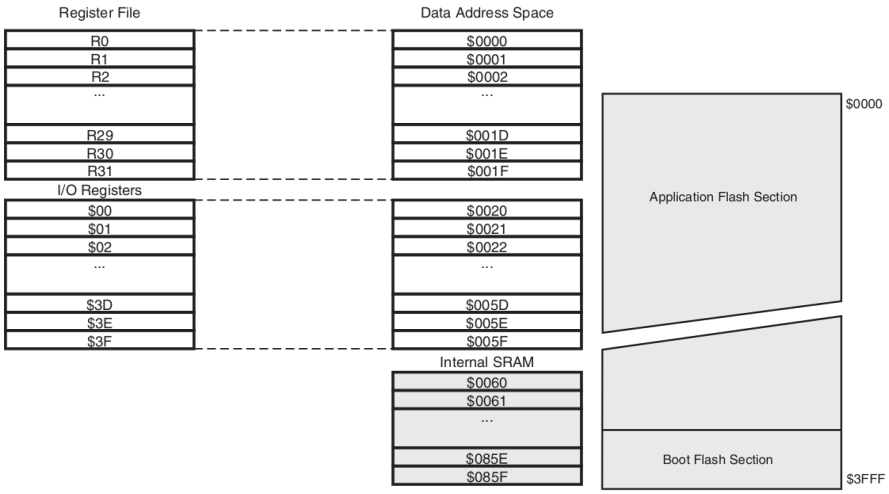


Figure 2.2: AVR memory maps.(a)Data memory.(b) Program memory

and R30 with R31 is the Z register. Different types of addressing modes are defined for transferring data between the Registers and the memory locations, mostly the SRAM.

In the AVR data memory space (figure 2.2), locations 0 to 31 (0x1F)are occupied by the Register File. Generally the assembler refers to them by names R1 to R31, not by the adresses.

2.1.1 Special Function Registers (SFR)

Location 0x20 to 0x5F (32 to 95 decimal) are occupied by the Special Function Registers (SFR), like the Status Register, the Stack Pointer and the control/status registers of the peripherals. *The Special Function Registers can also be accessed using the I/O address space ranging from 0 to 0x3F, using IN and OUT insructions.* The examples in this document will use the memory mapped addresses and not the I/O mapped ones.

The first Register is SREG, the status register that holds the flags resulting from

Figure 2.3: Registers X,Y and Z

the last executed arithmetic or logical instruction. There are several instructions whose results depend on the status of the bits inside SREG. Availability of SREG as a special function register allows us to examine the status of various flags, after arithmetic and logical operations. Stack Pointer is used as a pointer to the data address space. PUSH and POP instructions are used for moving data between the register file and location specified by the stack pointer.

All the peripherals and the general purpose I/O ports are operated by accessing the corresponding SFRs. We need to know about the I/O ports A,B,C and D because our example programs will be using them to display results. The SFRs used often in the example programs are listed below.

Name	I/O Addr.	Mem Addr	Function
SREG	0x3F	0x5F	Status Register
SPH	0x3E	0x5E	Stack pointer high byte
SPL	0x3D	0x5D	Stack pointer low byte
PIND	0x10	0x30	Input from Port D
DDRD	0x11	0x31	Data Direction of Port D
PORTD	0x12	0x32	Output to Port D, controls pull-up register for input bits
PINC	0x13	0x33	Input from Port C
DDRC	0x14	0x34	Data Direction of Port C
PORTC	0x15	0x35	Output to Port C, controls pull-up register for input bits
PINB	0x16	0x36	Input from Port B
DDRB	0x17	0x37	Data Direction of Port B
PORTB	0x18	0x38	Output to Port B, controls pull-up register for input bits
PINA	0x19	0x39	Input from Port A
DDRA	0x1A	0x3A	Data Direction of Port A
PORTA	0x1B	0x3B	Output to Port A

2.2 Writing Source Code and Assembling/Compiling

The programs can be written using any Text Editor. The resulting file is called the Source Code. This need to be translated into the machine language. In the case of Assembly language, the translating program is called an assembler. The translators for high level languages like C are called compilers. We are using the AVR GCC compiler, which can handle both C and Assembly language. The compiler recognizes the type of file from the filename extension, .c for C files and .S or .s for Assembly language. An uppercase extension allows the inclusion of header files where some constants are defined.

The translator program runs on a PC and the output need to be transferred to the target hardware, the AVR micro-controller. Before transferring it is converted into the Intel HEX file format, which is a text file whose format is understood by the uploader programs.

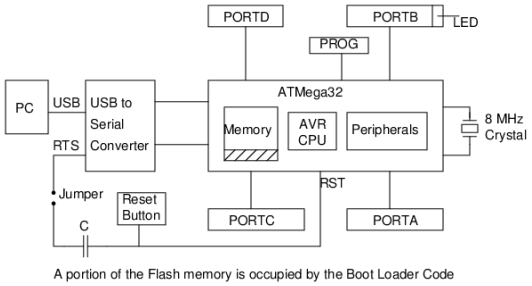


Figure 2.4: Block diagram of KuttyPy

Uploading the HEX file

In order to execute a new program we need to deposit it inside the Program memory of the micro-controller. This can be done in different ways. All the modern micro-controllers provide an interface called SPI (Serial Peripheral Interface) that consists of three terminals MISO, MOSI and SCK. There are uC programmers like USB ASP available supporting this standard.

Another popular method is to use a Boot Loader program. The boot loader program is deposited in the Program memory using the SPI interface. After a reset, the boot loader listens on the Serial Port of the uC for the arrival of any new program. If available, it is loaded into the memory and executed. Otherwise the old program residing there will be executed.

KuttyPyPlus hardware

The block disgram of KuttyPy circuit board is shown in figure 2.4. The complete circuit schematic is shown in figure2.5.

The board has a USB-to-Serial converter chip, to interface it to a PC. To transfer a new HEX file to, the uploader program resets the uC so that it enters the Boot loader code. After that the HEX file is transferred and the uC is reset again to execute the new code. In addition to the program loading capability, the KuttyPy boot loader is capable of accepting commands from the PC to control/monitor all Special Function Registers (SFR). This allows one to explore the micro-controller from a PC, without writing any program.

The KuttyPy Android App supports a Visual Programming Language to access the micro-controller. We can also program the uC using the Android App.

- 8 LEDs are connected from port B to ground, via 10K resistors
- 8 LEDs are connected from port D to ground, via 10K resistors
- 8 switches are connected from Port A to ground via 1K resistors
- A 6 bit R-2R DAC is connected to Port C bits PC2 to PC7, PC0 and PC1 are used by the I2C interface.
- The negative termnals of an RGB LED are connectd to the PWM outputs PB3, PD5 and PD7. The common anode is connected to 5 volts. **Due to this reason these LEDs will glow slightly when left in the Input Mode**.
- The I2C terminals are made available on a 4 pin Audio Jack connector also.

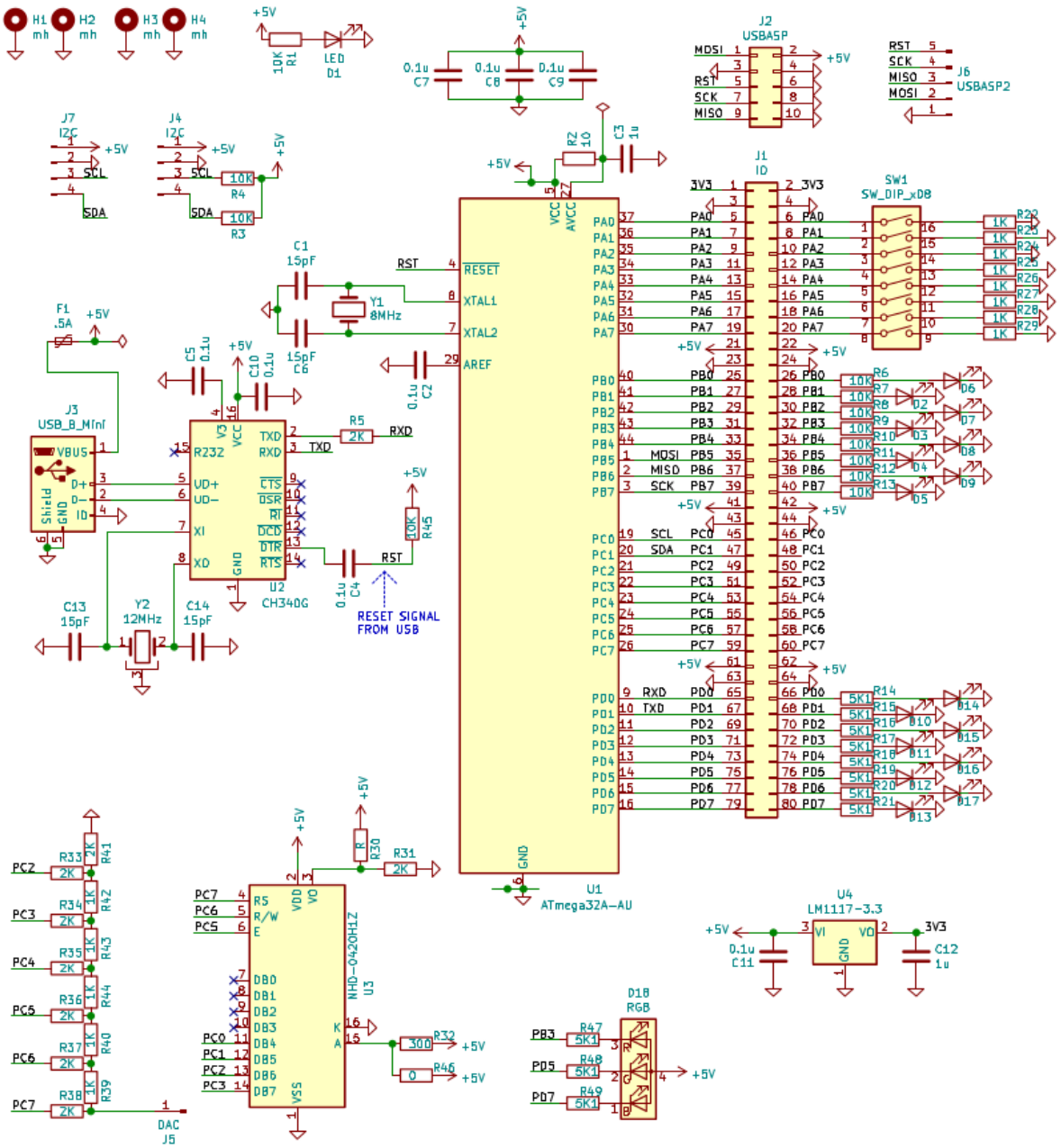


Figure 2.5: The circuit schematic of Kuttypy

Chapter 3

Assembly Language Programming

Main objective of this chapter is to learn the architecture of a micro-controller rather than developing large assembler programs. The main concern with assembly or machine language coding is that the program is specific to the architecture of the selected device. Our approach is to explore the architecture in a generic manner and provide some example programs that are more or less common to most of the processors.[1]

Main components of a micro-controller are shown in figure2.1. After powering up (or Reset) the Program Counter in initialized to zero, so that it points to the beginning of the Program Memory. The instruction stored at that location is brought to the Instruction Decoder and executed. This could be operations like; moving data between the General Purpose Registers (R0 to R31) or from a Register to a memory location, performing some arithmetic and logical operations, changing the content of the program counter, etc. The Special Function Registers are used for accessing the peripheral devices like Input/Output ports, ADCs, Timer/Counter etc. The popular family of micro-controllers like 8051, AVR and PIC follows the same architechture, even though the details may differ. Understanding them from a generic point of view will help switching from one type of controller to another without much effort.

To program in assembly language, we need to have some understanding about the Instruction Set, the Registers and the memory configuration of the micro-controller. We also need to know the syntax supported by the assembler we use, there is usually small differences between various assemblers. Since we are using Atmega32, belonging to the AVR family, and the GNU assember for AVR, further discussions will be restricted to them.

3.1 Format of an Assembler Program

A single line of code may have a

- Label: Always terminated by a colon. A line is labelled only if there is a jump or call to that line
- The instruction: Any valid instruction from the instruction set of the controller
- The operands: There could be 0, 1 or 2 of them, depending on the instruction
- Comment: anything after a semicolon is taken as a comment

[1] http://sourceware.org/binutils/docs/as/

```
lab1: INC  R1  ;increment the content of Register r1
```

The instruction and operand is not case sensitive but the labels are case sensitive, Lab1 is not the same as lab1.

Constants can be defined in two different ways, as shown below.

```
  .equ DDRB, 0x37
  DDRB = 0x37
```

Variables are defined inside the Data Sections as shown below.

```
  .section .data ; the data section
var1:
  .byte 15  ; global variable var1
```

Code is written under the Text Section, as shown below. This section should have a label named main and it should be declared Global.

```
  .section .text  ; The code section
  .global __do_copy_data  ; initialize variables
  .global __do_clear_bss  ; setup stack
  .global main ; declare label main as global
main:

  .end
```

1. .data, starts a data section, initialized RAM variables.
2. .text, starts a text section, code and ROM constants.
3. .byte, allocates single byte constants.
4. .ascii, allocates a non-terminated string.
5. .asciz, allocates a \0-terminated string.
6. .set declares a symbol as a constant expression (identical to .equ)
7. .global, declares a public symbol that is visible to the linker
8. .end, singifies the end of the program

The lines .global __do_copy_data and .global __do_clear_bss will tell the compiler to insert code for initializing variables, which is a must for programs having initialized data. The .text .section declaration is not enforced by GCC assembler. A minimal program that can be assembled without error is shown below.

```
.global main
main:
```

This code does nothing useful but will have some initialization instructions added by the compiler.

Now, let us write an example program to add two numbers (the instructions will be explained soon, first we need to learn howto assemble the code and execute it)

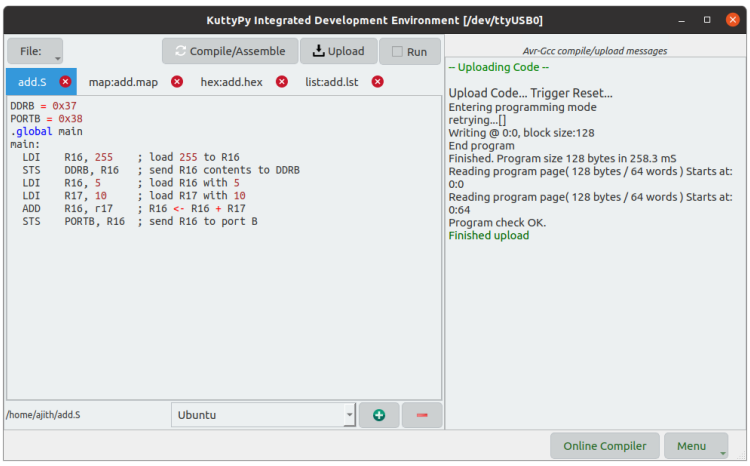


Figure 3.1: KuttyPy IDE

```
.section .text
.global main
main:
  LDI    R16, 5      ; load R16 with 5
  LDI    R17, 10     ; load R17 with 10
  ADD    R16, r17    ; R16 <- R16 + R17
```

3.2 Writing, Assembling and Uploading

The IDE can be used for editing source code, compiling/assembling, uploading it to the AtMega32 micro-controller and executing it. The KuttyPy software can be downloaded from https://csparkresearch.in/kuttypyplus. It contains three programs;

- KuttyPy GUI, provides a GUI to explore Atmega32 on a KuttyPy board
- KuttyPy Plus GUI, provides a GUI to explore Atmega32 on a KuttyPyplus board
- KuttyPy IDE, for editing, assembling/compiling and uploading to Atmega32

For program development, we need to use KuttyPy IDE. A screenshot of it shown in figure 3.1.

The KuttyPy IDE is also capable of accepting source code from a remote machine and send the .hex file back. This feature is used by the KuttyPy Android App to support Assember and C programming on Android phones. The KuttyPy App can be downloaded from Google playstore.

3.2.1 Using command line programs

The source code may be created by using any text editor. For assembling it we are using the AVR GCC compiler, which can translate both Assembly and C language programs. The output of the compiler need to be converted into a format (Intel HEX) understood by the uploader program and the Boot Loader inside the micro-controller. The commands to be used are given below:

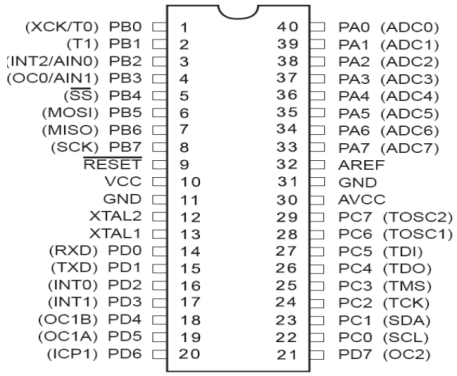


Figure 3.2: Atmega32 Pinout

```
avr-gcc -Wall -O2 -mmcu=atmega32 -o add add.S
avr-objcopy -j .text -j .data -O ihex add add.hex
```

The resulting .hex file can be uploaded by a program named *avr-dude*, using the command

```
avrdude -b 38400 -P /dev/ttyUSB0 -pm32 -c arduino -U flash:w:add.hex
```

The parameters are for the KuttyPy board.

If you want to generate a .lst file, to view the generated instructions, use the command

```
avr-objdump -S add > add.lst
```

Look for the lines following <**main**>**:** to view the output of your code. Assembler and linker add several other things for the proper functioning of it.

3.3 Input/Output ports of Atmega32

Executing **add.S** mentioned in the previous section will have the result in R16 but it is not visible to us. We need some method to view the result. The Atmega32 micro-controller has *four Input/Output Ports, named A, B, C and D.* These I/O ports are configured and accessed using four sets of Special Function Registers. Each port has 8 bits and the each bit can be configured as input or output, as explained in the next section.

The pinout diagram of Atmega32 is shown in figure 3.2. There are 32 pins organized as four ports named A, B, C and D, each 8 bit wide. Each pin can be configured as an input or output. The data direction and transfer are done by writing to the registers DDRx, PORTx and PINx (where x stands for A, B, C or D).

- DDRx : Direction of every pin of an I/O port is decided by the state of corresponding bit in the Data Direction registers DDRx. To configure a pin as output, make the bit 1, and to make it as input make it zero. For example, DDRA = 1 will configure BIT 0 of Port A (PA0) as output, and all other pins as input.

- PORTx : For pins that are configured as ouput, assigning a value to PORTX will set that data on them. For example PORTA = 1 will make PA0 high, that can be measured on the pin number 40 of the IC.

- PINx : For the pins configured as inputs, PINx will read the status of the external voltage level connected to the pins. For pins that are configured as outputs, PINx will return the data written to PORTx.

If the pins configured as inputs are left unconnected, there could be unwanted level changes due to electrical noise, this can be prevented by enabling the internal pull-up resistor. For pins that are configured as inputs, setting/clearing the bits in PORTx will enable/disable the corresponding internal pull-up resistor.

3.4 How to view the Register Contents

The KuttyPy board has LEDs connected to Port B and D. To view the content of any register, we can transfer it to either port B or D. The example program **'add.S'** shows howto display the results on the LEDs connected to the I/O ports.

```
ddrb = 0x17          ; I/O mapped address of DDRB, use OUT instruction
portb = 0x18
.section .text
.global main
main:
  LDI    R16, 255    ; load 255 to R16
  OUT    ddrb, R16   ; send R16 contents to DDRB, make all pins as outputs.
  LDI    R16, 5      ; load R16 with a number
  LDI    R17, 10     ; load R17 also
  ADD    R16, r17    ; R16 <- R16 + R17
  OUT    portb, R16  ; send R16 to port B, to light the LEDs
```

The *OUT DDRB, 255* instruction sets all the bits of DDRB, making all the pins of Port B as output. OUT PORTB, R16 copies the content of R16 to the register PORTB. The value of the individual bits decides the voltage level of the corresponding pin. For example, if R16 is 15, the four LSBs will become HIGH. If we use the STS instruction instead of OUT, then the memory mapped addresses 0x37 and 0x38 should be used.

3.4.1 Using the Pre-processor Option

The constants like PORTB, DDRB etc. are defined inside a file and we can request the pre-processor to include them. In order to do this

- We need to use the **.S** file extension to tell avr-gcc to call the assembler with the suitable pre-processor options.
- The source should have a line
  - #include <avr/io.h>

This is an option provided by the AVR GCC Assembler. Methods may be different for other assemblers. We will be using this option in the examples given in this document. The program **add-2.S,** that uses the header file, is shown below.

The include file *avr/io.h* contains the memory mapped addresses of the Special Function Registers. You need to use STS and LDS instead of OUT and IN. To avoid any conflict we are defining the I/O mapped addresse names using small letters.

```
#include <avr/io.h>
.section .text    ; denotes code section
.global main
main:
   LDI    R16, 255     ; load R16 with 255
   STS    DDRB, R16    ; set all bits of port B as output
   LDI    R16,  11     ;  load R16 with 2
   LDI    R17,  4      ;  load R17 with 4
   ADD  R16, R17       ;  R16 <- R16 + R17
   STS    PORTB, R16   ; result to port B
.END
```

3.5 Atmega32 Instruction Set

The AVR instructions are broadly classified into several types. We will consider some examples belonging to each type. For a complete description, refer to the Atemega32 databook.

- Data Transfer, using different addressing modes
  - MOV Rd, Rs ; Move between registers, from source (Rs) to destination (Rd)
  - LDI Rd, K ; Load immediate. The byte is part of the program
  - LD Rd, X ; Load indirect, from the address specified by X
  - ST X, Rs ; Store indirect, to the location specified by X (R26 and R27)
  - STS K, Rd ; Store direct to SRAM location K
  - LDS K, Rd ; Loads direct from SRAM location K
  - PUSH Rr ; Stores Rr to the memory specified by the Stack Pointer, decreases SP by 1
  - POP Rd ; Loads Rd from the memory specified by SP, increases SP by 1
  - OUT P, Rr ; Sends Rr to the I/O mapped address P
- Arithmetic and Logic
  - ADD Rd, Rr ; Adds to registers, result goes to Rd
  - SUB Rd, Rr ; Subtracts to registers.
  - AND Rd, Rr ; Logical AND of two registers, result goes to Rd
  - CLR Rd ; Clears the register
- Branching Instructions
  - JMP K ; Puts K in PC, results in a direct jump to location K
  - RJMP K ; Relative jump PC <- PC + K + 1

  - CALL K ; PC goes to K+1, PC is pushed to the Stack
  - RCALL k ; Relative call, PC <- PC + K + 1
  - RET ; PC is retrieved from Stack
  - BRNE k ; Relative jump if Z bit in SREG is set. PC <- PC + K + 1

- But Testing and Setting
  - CP Rd, Rr ; Compare the registers and set the bits of SREG accordingly
  - LSL Rd ; Logical Shift Left

- MCU control
  - NOP ; Does nothing, takes one clock cycle
  - SLEEP ; Enters Sleep mode, to be waken by an interrupt

For a complete list of instructions supported by Atmega32, refer to the data sheet. We will only examine some of them to demonstrate different types of memory addressing and the arithmetic and logical operations.

3.6 Data Transfer, Addressing Modes[2]

The micro-controller spends most of the time transferring data between the Register File, SFRs and the RAM. Let us examine the different modes of addressing the Registers and Memory.

3.6.1 Register Direct (Single Register)

The contents of the register is read, specified operation is performed on it and the result is written back to the same register. For example

```
Lab1:  INC  R2   ; increments Register 2
```

The line above shows the format a line of code in assembly language. The label field is required only if the program needs to jump to that line. Everything after the semicolon is comment only.

3.6.2 Register Direct (Two Registers)

The contents of the source and destination registers are read, specified operation is performed and the result is written back to the destination register. The format is to specify the destination first. For example

```
MOV  R2, R5   ; content of R5 is copied to R2
ADD  R1, R2   ; r1 + r2 stored to r1
```

[2]https://www.arxterra.com/5-avr-branching/

3.6.3 Load Immediate

In this mode, data to be transferred to a register, is part of the program itself. **Registers below R16 cannot be used under this mode.**

```
; data-immed.S  , demonstrate Load Immediate mode
ddrb = 0x17          ; I/O mapped address of DDRB
portb = 0x18         ; and PORTB
.section .text    ; denotes code section
.global main
main:
  LDI R16, 255     ; load R16 with 255
  OUT ddrb, R16    ; Display content of R16
  OUT portb, R16     ; using LEDs on port B
.end
```

The LEDs connected to port B will display the value moved to R16

3.6.4 Data Direct

In this mode, the address of the memory location containing the data is put in a register, and it is used in the Data Transfer instruction. Data could be transferred from memory register (LDS) or from a register to memory(STS). The example **data-direct.S** demonstrates the usage of LDS and STS instructions. The address that can be accessed is limited from 0 to 255, because we use an 8 bit register.

First we use the Load Immediate mode to initialize R16 with some value. Then R16 contents are Stored to the memory location DDRB. After that R17 is loaded from the memory location DDRB. This is then Stored to location PORTB.

```
ddrb = 0x17
portb = 0x18
PORTA = 0x3b      ; memory mapped address of PORTA
.section .text    ; denotes code section
.global main
main:
   LDI R16, 0xff    ; load r16 with 255
   OUT ddrb, R16    ;
   STS PORTA, R16   ; Store R16 to the memory location PORTA
   LDS R17, PORTA    ; read it back to R17
   OUT portb, R17   ; display it on port B LEDs
  .end
```

3.6.5 Defining variables in memory

In the previous example we used the addresses of the Special Function Registers. Now we define a variable in the Static RAM area. **'data-direct-var.S'** is listed below.

```
ddrb = 0x17
portb = 0x18
```

```
.section .data
var1:
.section .text    ; denotes code section
.global main
main:
   LDI R16, 0xff   ; load r16 with 255
   OUT ddrb, R16   ; make all bits of port B as output
   LDI R16, 15     ; load R16 with a number
   STS var1, R16   ; Store R16 to location var1
   LDS R17, var1   ; Load R17 from var1
   OUT portb, R17  ; display R17 contents
.end
```

The label 'var1', defined inside the data section is used inside the code. The actual value can be seen from the .lst file generated by the avr-objdumb program. Generated machine language code for the section 'main' is shown below.

```
0000006c <main>:
6c:  0f ef           ldi r16, 0xFF ; 255
6e:  07 bb        out 0x17, r16 ; 23
70:  0f e0        ldi r16, 0x0F ; 15
72:  00 93 60 00  sts
0x0060, r16  ; 0x800060 <__DATA_REGION_ORIGIN__>   76:
10 91 60 00  lds
r17, 0x0060  ; 0x800060 <__DATA_REGION_ORIGIN__>   7a:
18 bb        out 0x18, r17 ; 24
```

It can be seen that the label 'var1' is given the RAM address of 0x0060. Also note that the main is at address 0x0000006c in the program address space. Examine the .lst file to have a look at the complete code, including the sections added by the assembler.

3.6.6 Data Indirect

In the data-direct mode, the address of the memory location is part of the instruction word. In Data Indirect mode the address of the memory location is taken from the contents of the X, Y or Z registers. This mode has several variations like pre and post incrementing of the register or adding an offset to it. Program **data-indirect.S** is listed below.

```
ddrb = 0x17
portb = 0x18
.section .data    ; data section starts here
var1:
.section .text    ; denotes code section
.global main
main:
   LDI R16, 0xff   ; load r16 with 255
   OUT ddrb, R16   ; make all bits of port B as output
   LDI  R17, 0b10101010       ; set r17 to 10101010b
```

```
    STS  var1, R17              ; store it to RAM at var1, using direct mode
    LDI  R26, lo8(var1)         ; R26 and R27 forms X, the 16 bit address
    LDI  R27, hi8(var1)
    LD   R16, X                 ; Load R16 from location pointed to by X

    OUT portb, R16              ; display R16 contents
.end
```

The operators lo8() and hi8() are provided by the assembler to extract the high and low bytes of the 16bit memory address. Indirect addressing using the registers X,Y and Z offer many variations, like auto-increment of the pointer after the operation.

3.7 Variable Initialization

In the previous examples, we have not initialized the global variable 'var1' inside the program. The example **global-init.S** listed below demonstrates this feature.

```
ddrb = 0x17
portb = 0x18
.section .data
var1:
   .byte 0xee
.section .text    ; denotes code section
   .global __do_copy_data ; initialize global variables
   .global __do_clear_bss ; and setup stack pointer
   .global main
   main:
     LDI R16, 0xff        ; load r16 with 255
     OUT ddrb, R16        ; make all bits of port B as output
     LDS R16, var1        ; load R16 from location 'var1'
     OUT portb, R16       ; display R16 contents
 .end
```

The lines

```
.global __do_copy_data ; initialize global variables
.global __do_clear_bss ; and setup stack pointer
```

are for initializing variables and setting up the stack, essential for programs with initialized data.

3.7.1 Storing to the Stack, PUSH and POP

A region of memory can be allocated as Stack, by pointing the Stack Pointer register (SPL and SPH) to it. Data can be stored and retrieved in the "Last in first out" mode. The last item pushed is retrieved by a pop instruction.

```
ddrd = 0x11
portd = 0x12
ddrb = 0x17
portb = 0x18
.section .text    ; denotes code section
.global __do_clear_bss  ; setup stack pointer
.global main
main:
   LDI R16, 0xff        ; load r16 with 255
   OUT ddrb, R16        ; make all bits of port B as output
   OUT ddrd, R16        ; make all bits of port D as output
   LDI R16, 1
   PUSH R16             ; push R16 content to the stack
   INC R16
   PUSH R16             ; push the incremented value
   POP R17              ; should pop the last pushed value (2)
   POP R18              ; shoul pop the previous value (1)
   OUT portb, R17       ; display on port B
   OUT portd, R18       ; display on port D
.end
```

Port D should display '1' and Port B the incremented value '2'.

3.7.2 I/O Direct

These type of instructions are to transfer data between the Registers (R1 to R31) and the Special Function Registers, that can also be accessed as I/O ports. The following example **io-direct.S** demonstrates this.

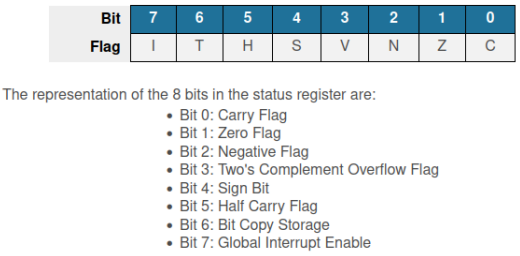
```
.section .text    ; denotes code section
.global main
main:
   LDI R16, 255
   LDI R17, 0b10101010
   OUT 0x17, R16        ; I/O address of DDRB is 0x17
   OUT 0x18, R17        ; PORTB is at 0x18, alternate LEDs should glow
.end
```

Executing this program should switch ON the LED connected to the LSB of Port B. Modify the program to remove the INC instruction, assemble and upload it again, the LED should go off. The generated code is smaller in the case of I/O space addressing using the OUT instruction, compared to the load direct addressing using STS.

3.8 Arithmetic and Logic

The status register bits are explained in figure 3.3. The Arithmetic and Logic instructions affect these bits.

The program **and.s** ,listed below, does a logical AND operation between a register and an integer. Edit the program to AND with zero instead of one, to see the ero Flag of the status register set.



Bit	7	6	5	4	3	2	1	0
Flag	I	T	H	S	V	N	Z	C

The representation of the 8 bits in the status register are:

- Bit 0: Carry Flag
- Bit 1: Zero Flag
- Bit 2: Negative Flag
- Bit 3: Two's Complement Overflow Flag
- Bit 4: Sign Bit
- Bit 5: Half Carry Flag
- Bit 6: Bit Copy Storage
- Bit 7: Global Interrupt Enable

Figure 3.3: Status Register

```
ddrb = 0x17        ; I/O mapped addresses
portb = 0x18
ddrd = 0x11
portd = 0x12
SREG = 0x5F        ; memory mapped address of status register
.section .text
.global main
main:
  LDI    R16, 255    ; load 255 to R16
  OUT    ddrb, R16
  OUT    ddrd, R16
  ANDI   R16, 1      ; result in R16
  OUT    portb, R16  ; send R16 to port B, to light the LEDs
  LDS  R17, SREG
  OUT  portd, R17      ;  status to port D
.END
```

3.9 Program Flow Control

The programs written so far has an execution flow from beginning to end, without any branching or subroutine calls, generally required in all practical programs. The execution flow can be controlled by CALL and JMP

3.9.1 Jump instructions

The program counter can be modified to change the flow of execution of the code. Example 'jump.S' demonstrates a direct jump.

```
#include <avr/io.h>
.section .text    ; denotes code section
.global main
main:
    LDI R16, 255
    STS DDRB, R16

    JMP skip
    LDI R16, 15
skip:
```

```
    STS PORTB, R16
.end
```

Running **jump.S**, will put on all the LEDs. Comment the JMP instruction and execute the code again to figure out the difference it is making. Jumps can be conditional also, like:

```
CPI   R16, 100
BREQ  loop1
```

The branching will happen only if R16 is equal to 100. Write a program to demonstrate this

3.9.2 Calling a Subroutine

Subroutines or functions are an important part of modular programming. It is supported at the hardware level by the CALL and RCALL instructions. For a direct call, the content of the Program Counter is replaced by the operand of the CALL instruction. For a relative call, the operand is added to the current value of the Program Counter. In both cases the current value of the PC is pushed into the memory location pointed by the Stack Pointer register. The RET instruction, inside the called subroutine, pops the stored PC to resume execution from the called point. Program sub-routine.s listed below demonstrates this feature. The program **sub-routine.S** is listed below.

```
#include <avr/io.h>
.section .text    ; denotes code section
.global main
disp:             ; subroutine
  STS PORTB, R17        ; send R17 PORTB
  RET
.global main
main:
  LDI  R16, 255
  STS  DDRB, R16     ; DDRB

  LDI R17, 3
  RCALL disp         ; relative call
  ;CALL  disp    ; direct call
.end
```

The LEDs connected to PB0 and PB1 will glow. Uncomment the line CALL disp and find out the difference in the generated code, from the .lst file. Functionally both are same but relative jump is possible only if the offset is less than 256.

3.9.3 Interrupt, Call from anywhere

So far we have seen that the execution flow is decided by the program instructions. There are situations where the uC should respond to external events, stopping the current

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow
13	$018	SPI, STC	Serial Transfer Complete
14	$01A	USART, RXC	USART, Rx Complete
15	$01C	USART, UDRE	USART Data Register Empty
16	$01E	USART, TXC	USART, Tx Complete
17	$020	ADC	ADC Conversion Complete
18	$022	EE_RDY	EEPROM Ready
19	$024	ANA_COMP	Analog Comparator
20	$026	TWI	Two-wire Serial Interface
21	$028	SPM_RDY	Store Program Memory Ready

Figure 3.4: Interrupt vectors of Atmega32. Addresses according to a 2byte word arrangement.

program temporarily. This is done using Interrupts, that are external signals, either from the I/O pins or from from some of the peripheral devices. On receiving an interrupt signal, the processor stores the current Program Counter to the memory location pointed to by the Stack Pointer and jumps to the corresponding interrupt vector location, as shown in figure . For example, the processor will jump to location 0x0002 (0x0004 if you count them as bytes), if external interrupt pin INT0 is activated, provided the interrupt is enabled by the processor beforehand.

The interrupt vector location is filled with the address of the subroutine handling the interrupt. For the interrupts that are not used by the program, the assembler fills some default values. After executing the Interrupt Service Routine, the program execution resumes at the point where it was interrupted. The program **interrpt.S** listed below shows the usage of interrupts. Connect 8 LEDs to Port B and run the code. Connect PD2 to ground momentarily and watch the LEDs.

3.10 Output of the Assembler

We have learned howto write, assemble and execute simple assembler programs. Let us assemble a program with a single instruction, as shown below.

```
  ; test.s , an single line program
    .section .data ; data section starts here
    .section .text ; denotes code section
    .global main
main:
    clr r1
    .end
```

The generated machine language output can be examined by looking at the .lst output,

shown below, generated by the objdump program. It can be seen that the assembler generates some code that is required for the proper operation of the uC. In the Atmega32 Program memory, the first 80 (50hex) bytes are supposed to be filled with the addresses of the 20 interrupt vectors. It can be seen that, the program jumps to location __ctors_end (54hex). The porcessor status register (0x3F) is cleared and the Stack Pointer is initialized to 0x085F (the last RAM location), before calling our main section. After returning from the main, it jumps to _exit (0x6e), clears the interrupt flag and then enters an infinite loop. That means we need to end the main section with an infinite loop, if our program uses interrupts.

```
/home/ajith/microhope/ASM/test:     file format elf32-avr
Disassembly of section .text:
00000000 <__vectors>:
   0:  0c 94 2a 00  jmp 0x54 ; 0x54 <__ctors_end>
   4:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
   8:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
   c:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  10:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  14:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  18:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  1c:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  20:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  24:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  28:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  2c:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  30:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  34:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  38:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  3c:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  40:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  44:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  48:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  4c:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
  50:  0c 94 34 00  jmp 0x68 ; 0x68 <__bad_interrupt>
00000054 <__ctors_end>:
  54:  11 24        eor r1, r1
  56:  1f be        out 0x3f, r1 ; 63
  58:  cf e5        ldi r28, 0x5F ; 95
  5a:  d8 e0        ldi r29, 0x08 ; 8
  5c:  de bf        out 0x3e, r29 ; 62
  5e:  cd bf        out 0x3d, r28 ; 61
  60:  0e 94 36 00  call 0x6c ; 0x6c <main>
  64:  0c 94 3e 00  jmp 0x7c ; 0x7c <_exit>
00000068 <__bad_interrupt>:
  68:  0c 94 00 00  jmp 0 ; 0x0 <__vectors>
0000006c <main>:
  6c:  88 27        eor r16, r16
0000006e <_exit>:
  6e:  f8 94        cli
```

```
00000070 <__stop_program>:
  70:  ff cf        rjmp .-2; 0x70 <__stop_program>
```

3.11 Using Pre-processor, .s and .S

The examples described so far used the **.s** extension for the filenames. The program **square-wave-tc0.s** listed below generates a 15.93 kHz square wave on PB3.

```
TCCR0 = 0x53
WGM01 =  3
COM00 = 4
OCR0 = 0x5C
DDRB = 0x37
PB3 =  3
  .section .text  ;code section
  .global main
main:
  ldi r16,  (1 << WGM01) | (1 << COM00) |  1  ;CTC mode
  sts TCCR0 , r16
  ldi r16, 100
  sts OCR0, r16
  ldi r16, (1 <<  PB3)
  sts DDRB, r16
  .end
```

The addresses of the Special Function Registers and the various bits inside them are defined inside the program (first 6 lines). Instead of entering them like this, we can use the corresponding include file. We need to use the **.S** file extension to tell avr-gcc to call the assembler with the suitable pre-processor options. The same program re-written with .S extension**, square-wave-tc0.S**, is listed below.

```
#include <avr/io.h>
  .section .text
  .global main
main:
  ldi r16,(1 << WGM01) | (1 << COM00) | 1 ; CTC mode
  sts TCCR0 , r16
  ldi r16, 250
  sts OCR0, r16
  ldi r16, (1 << PB3)
  sts DDRB, r16
  .end
```

The second method is advisable if you plan to develop larger assembler programs for practical applications.

3.12 Example Programs

The programs described below performs better than their C counterparts.

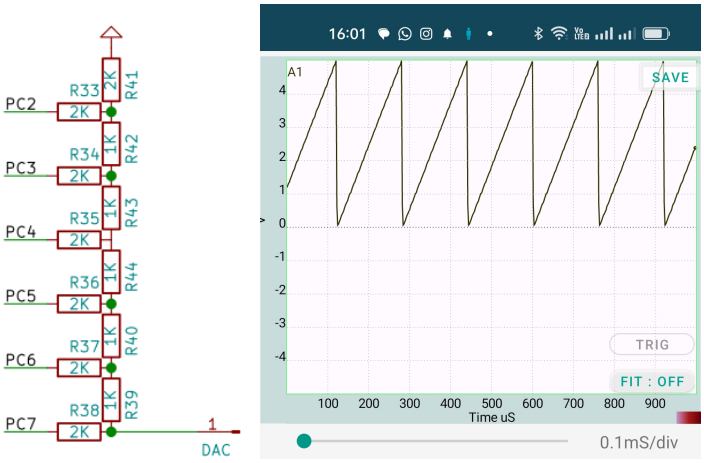


Figure 3.5: R2R DAC on port B (a) schematic (b) output waveform

3.12.1 R2R DAC on Port C

A R2R network, as shown in figure 3.5(a), is connected to port C. The program writes the content of R1 to port C in an infinite loop. R1 ia incremented every time and after reaching 255, it will become 0, resulting in a ramp at the output of the R-2R network, figure 3.5(b). The frequency of the ramp generated is around 6 kHz.

```
; program ramp-on-R2RDAC.S , generates ramp on Port C
#include <avr/io.h>
   .section .text
   .global main
main:
   ldi r16, 255
   sts DDRC, r16 ; make port C as output
loop:
   inc r1
   sts PORTC, r1   ; R1 to PORTC, R-2R DAC
   rjmp loop
   .end
```

3.12.2 Sine wave Generator

The program **sine-wave.S** listed below uses Timer/Counter 0 to trigger an interrupt when the counter reaches the set point register OCR0. Register X is pointed to a sine table stored in the SRAM. On an interrupt the value from sine table, pointed to by X, is written to Port B where the R-2R DAC is connected. Register R22 is used for reseting the pointer after 32 increments. The R-2R DAC on port B generates the instantaneous DC output values, that makes the sine wave.

```
#include <avr/io.h>
   .section .data
   .global stab
stab:   ; sine table
```

```
    .byte 128,150,171,191,209,223,234,240,242,240,234,\
223,209,191,171,150,128,105,84,64,46,32,21,\
     15,13,15,21,32,46,64,84,105,127
    .section .text   ; code section
    .global __do_copy_data
    .global __do_clear_bss
    .global  TIMER0_COMP_vect
TIMER0_COMP_vect:     ; ISR
    ld   r24, X+ ; load from table, increment
    sts  PORTC, r24 ; write it to PORTB
    inc  r22        ; increment r22
    CPSE r20,r22 ; reached the end of table
    reti         ; return if NOT equal
    clr  r22 ; ready for next round
    subi r26,32  ; subtract 32 from XL, point to table start
    reti

    .global main
main:
    ldi  r16, 255
    sts  DDRC, r16
    ldi  r16, (1 << WGM01) |  1  ; TCCR0 to CTC mode
    sts  TCCR0 , r16
    ldi  r16, 50   ; Set point reg to 50
    sts  OCR0, r16
    ldi  r16, (1 << OCIE0)  ; set TC0 compare interrupt enable
    sts  TIMSK, r16
    ldi  r16, (1 << OCF0)   ; interrupt enable bit
    sts  TIFR, r16
    ldi  XL, lo8(stab)      ; point X to the sine table
    ldi  XH, hi8(stab)      ; XL = R26, XH = R27
    clr  r22                ; R22 will keep track of the  location in table
    ldi  r20,32             ; Store size of the table in R20
    sei
loop:
    rjmp loop   ; infinite loop
    .end
```

The output of the code is shown in figure3.6. The resolution of the R-2R DAC is only 6 bits.

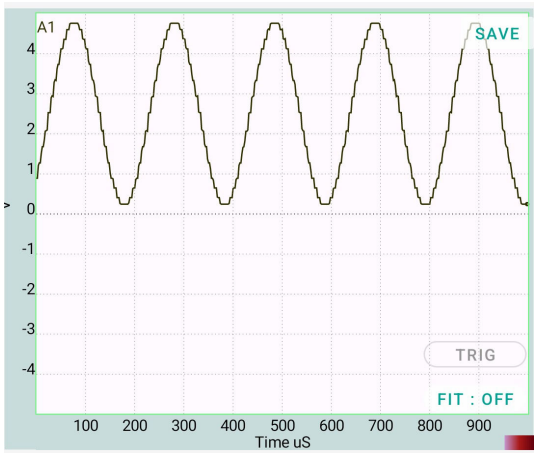


Figure 3.6: Sinewave output

Chapter 4

Programming Atmega32 using C language

For most of the practical applications, programming the micro-controller involves accessing the peripheral devices like I/O ports, ADCs, Timer/Counter, Serial communication port etc. This is done by reading/writing to the Special Function Registers. This chapter start with some simple examples and presents a library of functions.

4.1 Blinking LED

Writing zeros and ones in a closed loop to an output pin will blink the LED connected to it. To make it slower we need to insert a delay. The program **blink.c** is listed below.

```
#include <avr/io.h>
void delay_ms (uint16_t k)  // waits for k milliseconds, for 8MHz clock only
  {
   volatile uint16_t x;
   while(k--) {x=532; while (x--);}
 }
int main (void)
{
   DDRB = 1;     // configure PB0 as output
   for(;;)
   {
   PORTB = 1;
   delay_ms(500);
   PORTB = 0;
   delay_ms(500);
   }
}
```

The functions like delay_ms() will be used by many programs and it is more convenient to put them in a library.

The program **blink-2.c** listed below is a modified version of **blink.c**. This one uses the delay_ms() function from the library. The Kuttypy IDE will include **libkp.a** while linking, you may enable/disable this feature from the IDE.

```
#include <avr/io.h>
extern void delay_ms (uint16_t k);  // delay_ms() is inside the kuttypy library
int main (void)
{
   DDRB = 1;     // configure PB0 as output
   for(;;)
   {
   PORTB = 1;
   delay_ms(500);
   PORTB = 0;
   delay_ms(500);
   }
}
```

4.2 The Kuttypy Library

This section explains several functions, mainly for accessing the peripheral devices like I/O ports, ADCs, Timer/Counters etc. They are distributed as a static library named **libkp.a**, included in the KuttyPy software package. Linking with this library can be enabled/disabled from the KuttyPy IDE.. The source files is available in the files starting with "kp-".

- kp-utils.c
  - delay_100us(x) : waits for 100x microseconds, assumes 8MHz clock
  - delay_ms(x) : waits for x milliseconds, assumes 8MHz clock

Other functions related to the peripheral devices are explained in the corresponding sections.

4.2.1 Creating and testing the Static Library

This section is only for those who are interested in the details of making a library. The source code files for the KuttyPy library are kept inside the subdirectory named 'kplib'. Change to this directory, and issue the following commands

```
avr-gcc -g -O2 -mmcu=atmega32a -c kp-adc.c kp-utils.c  # add all .c files here
avr-ar -r libkp.a kp-adc.o kp-utils.o      # make the library archive, add all
```

copy the library file 'libkp.a' to the avr-gcc library path (/usr/lib/avr/lib on Ubuntu)

```
sudo cp libkp.a /usr/lib/avr/lib/
```

The header file containing the prototype definitions of all the functions are inside 'kp.h', kept inside the directory
'/usr/lib/avr/include/avr' on Ubuntu. The programs using the library should have the following line.

```
#include <avr/kp.h>
```

The following commands can be used for compiling, generating the .hex file and uploading a program. Linking with the libkp.a library is done using the **-lkp** option.

```
avr-gcc -Wall -O2 -mmcu=atmega32 -o blink blink.c -lkp
avr-objcopy -j .text -j .data -O ihex blink blink.hex
avrdude -b 38400 -P /dev/ttyUSB0 -pm32 -c arduino -U flash:w:blink.hex
```

4.3 Macros to TEST, SET and Clear BITs

We often need to test the status of specific BITs in an SFR or SET/CLEAR them. The macros given below may be used for this purpose. The macros are converted into appropriate C code by the pre-processor.

```
#define BITVAL(bit) (1 << (bit))
#define CLRBIT(sfr, bit) (_SFR_BYTE(sfr) &= ~BITVAL(bit))
#define SETBIT(sfr, bit) (_SFR_BYTE(sfr) |= BITVAL(bit))
#define GETBIT(sfr, bit) (_SFR_BYTE(sfr) & BITVAL(bit))
```

BITVAL(bit position)

The value of bit position could be 0 to 7 in the case of 8 bit integers and 0 to 15 for 16 bit integers. This macro returns (1 << bit position). For example BITVAL(3), will give 8, that is binary 1000, obtained by left shifting of 1 thrice.

SETBIT(variable, bit position)

This macro SETS the specified bit in the given variable, without affecting the other bits. For example SETBIT(DDRB, 7), will make the last bit of DDRB high.

CLRBIT(variable, bit position)

This macro clears the specified bit of the given variable. For example CLRBIT(val, 0), clears the least significant bit of 'val', that is an integer type variable.

GETBIT(variable, bit position)

This macro returns the value the specified bit if the specified bit of the variable is 1, else it returns zero. For example: if x = 3, GETBIT(x, 1) will return 2 and GETBIT(x,3) will return zero.

4.3.1 Example Programs

The program **copy.c** ,listed below, reads port A and sets the same on port B LEDs. We have enabled the internal pullup resistor on port A so that and it will go LOW only when it is connected to ground. The Kuttypyplus board has 8 DIP switches connected from port A to ground.

```
#include <avr/io.h>   // Include file for I/O operations
int main (void)
{
```

```
DDRA = 0;              // Port A as Input
PORTA = 255;             // Enable pullup on PA0
DDRB = 255;              // Configure PB0 as output
for(;;)
     PORTB = PINA;    // Read Port A and write it to Port B
}
```

This code cannot be use for manipulating a specific bit without affecting the others. For example, if we need to change PB0 depending on the state of PA0, the macros listed below comes handy. These macros can be used on variables, defined in the program, and also on registers like DDRX, PORTX etc.

Let us rewrite the previous program as **copy-bit.c**, using these macros as:

```
#include <avr/io.h>
  int main (void)
  {
  uint8_t  val;
  DDRA = 0;              // Port A as Input
  PORTA = 1;             // Enable pullup on PORTA, bit 0
  DDRB = 1;              // Pin 0 of Port B as output
  for(;;)
     {
     val = GETBIT(PORTA, 0);
     if (val != 0)
        SETBIT(PORTB, 0);
     else
         CLRBIT(PORTB, 0);
     }
  }
```

The same can be done, without using the bit manipulation macros, as shown in **copy-bit-2.c**

```
#include <avr/io.h>   // Include file for I/O operations
int main (void)
{
DDRA = 0;              // Port A as Input
PORTA = 1;             // Enable pullup on PA0
DDRB = 1;              // Configure PB0 as output
for(;;)
   if(PINA & 1)          // If PA0 is set
       PORTB |= 1;       // Set PB0, by ORing with 00000001b
   else                  // otherwise clear PB0
       PORTB &= ~1;      // by ANDing with 11111110b (~00000001b)
}
```

The code fragment shown above uses the Bitwise AND, OR and NOT operators.

4.4 The alphanumeric LCD Display

https://en.wikipedia.org/wiki/Hitachi_HD44780_LCD_controller
LCD display

Thefollowing functions to access the display, connected to port C, are available in the library.

- void lcd_init(void) : initializes the LCD display
- void lcd_clear(void) : clears the LCD display
- void lcd_put_char(char ch) : Outputs a single character to the LCD display
- void lcd_put_string(char* p) : : Displays a string to the LCD
- void lcd_put_byte(uint8_t val) : Diplays an 8 bit unsigned integer
- void lcd_put_int(uint16_t val) : Diplays a 16 bit unsigned integer
- lcd_put_long(uint32_t val) : Diplays an 32 bit unsigned integer
- lcd_put_float(float val, int ndec) : Diplays a decimal number, maximum 3 digits after decimal point .Defining float type data increases the program size a lot.

The example program **test-lcd.c** listed below demonstrates the usage of the LCD display.

```
#include <avr/kp.h>
int main()
{
lcd_init();
lcd_put_string("A ");
lcd_put_float(45.3, 1);
lcd_put_char(' ');
lcd_put_byte(255);
lcd_put_char(' ');
lcd_put_int(65534);
lcd_put_char(' ');
lcd_put_long(100000);
}
```

4.5 Serial Communication Port

The Atmega32 micro-controller has a Universal Serial Asynchronous Port (UART). On the KuttyPy board, the receiver and transmit pins are connected to the USB to Serial Converter IC. Programs are transferred to the uC using this path, by using the boot loader program. User programs also can use this path to communicate to the PC via the USB port.

The following functions are available for handling the UART.

- void uart_init(int baud)
  - The maximum baudrate supported is 38400. You may also use 19200, 9600 etc., configured to use 1 Stop Bit and even parity.

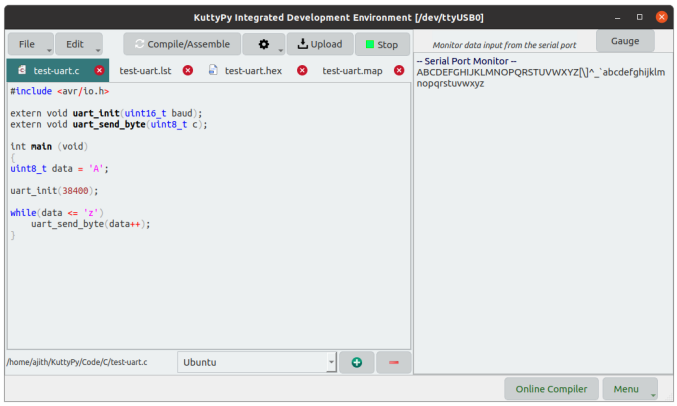


Figure 4.1: KuttyPy IDE showing output from test-uart.c

- uint8_t uart_recv_byte(void) : Waits on the UART receiver for a character and returns it
- void uart_send_byte(uint8_t c) : Sends one character over the UART transmitter.

The KuttyPy IDE displays the data send by the UART in a window. The program **test-uart.c** is shown below and the output is visible in the IDE screenshot shown in figure 4.1.

```
#include <avr/kp.h>
int main (void)
  {
  uint8_t data = 'A';

  uart_init(38400);
  while(data <= 'z') uart_send_byte(data++);
}
```

4.5.1 Communication between PC and uC, via USB port

The USB to Serial chip of the KuttyPy board is connected to the USB port of the PC. To the programs running on the PC, The USB to Serial interface will appear as a virtual COM port. On GNU/Linux systems it can be accessed as /dev/ttyUSB0 and on MSWindows as COMx.

We can use a simple Python program to handle the communicate on the PC side. to the micro-controller. You need to install Python interpreter and the python-serial module on the PC for this to work. The KuttyPy IDE should be closed before running these Python programs, because both of them use the same communication channel.

The program **echo.c** waits for data from the PC, vis the USB to serial converter, increment it by one and sends it back. The received data is also send to port B LEDs.

```
#include <avr/kp.h>
int main(void)
{
uint8_t data;
```

```
DDRB = 255;
uart_init(38400);
for(;;)
     {
      data = uart_recv_byte();
      PORTB = data;
      uart_send_byte(data + 1);
     }
}
```

After uploading this program, open a terminal window, and run the python program **echo.py** listed below, using the commands:[1]

$ python3 echo.py

```
import serial, time
fd = serial.Serial('/dev/ttyUSB0', 38400, stopbits=1, timeout = 1.0)
time.sleep(1)        # wait for the uC to reach our code after the reset.
fd.write(b'\x04') # send one byte
print (fd.read(1))   # receive one byte
time.sleep(4)        # The port B LEDs will show the result before the program
```

It should be noted that the Python program resets the uC while opening and closing the virtual port connection. The one second wait after opening gives enough time for the uC to start our code, echo.c. While exiting the Python program the uC is reset again. We have added another sleep() statement so that the result can be seen on port B. While deploying this board permanently for communication applications it is better to remove the capacitor C4 (refer to the schematic 2.5.). Once you remove C4, program upload must be done using the SPI interface, using USBASP programmer, as explained in section **??**.

We can rewrite echo.c without using the library functions. The program **echo-direct.c** listed below id functionally identical to echo.c

```
#include <avr/io.h>
int main(void)
{
  uint8_t data;
  DDRB = 255;

  UCSRB = (1 << RXEN) | (1 << TXEN);
  UBRRH = 0;              ////38400 baudrate, 8 databit, 1 stopbit, No parity
  UBRRL = 12;             // At 8MHz (12 =>38400)
  UCSRC = (1<<URSEL) | (1<<UCSZ1) | (1<< UCSZ0);
  for(;;)
     {
     while ( !(UCSRA & (1<<RXC)) );  //wait on Rx
     data = UDR;                     // read a byte
     PORTB = data;
```

[1] If you are using some USB to Serial converters like MCP20=2200, the virtual com port will appear as ttyACMx instead of with ttyUSBx.

```
        while ( !(UCSRA & (1<<UDRE)) ); // Rx Empty ?
        UDR = data + 1;
    }
}
```

4.6 The Analog to Digital Converter (ADC)

Most of the I/O PORT pins of Atmega32 have alternate functions. PA0 to PA7 can be used as ADC inputs by enabling the built-in ADC. All the pins configured as inputs in the DDRA will become ADC inputs, but the ones configured as outputs will remain as digital output pins. The ADC converts the analog input voltage in to a 10-bit number. The minimum value represents GND and the maximum value represents the ADC reference voltage. The reference inputs could be AVCC, an internal 2.56V or a voltage connected to the AREF pin. The selection is done in software. The ADC operation is controlled via the registers ADMUX and ADCSRA. The data is read from ADCL (must be read first) and ADCH. If the Left adjust option (ADLAR bit in register ADMUB) is selected, 8 bit data will be available in ADCL.

The KuttyPy ibrary contains the following functions to use the ADC.

- void adc_enable()
- void adc_disable()
- void adc_set_ref(uint8_t val)
  - 0 : Externally applied reference voltage
  - 1 : AVCC as reference
  - 2 : Interval 2.56 volts reference
- uint16_t read_adc(uint8_t ch) // returns 10 bit data
- uint8_t read_adc_8bit(uint8_t ch) // returns 8 bit data

4.6.1 Reading an Analog Voltage

The example program ***adc-read.c*** is listed below. It reads an ADC channel zero and displays the result on port B, after discarding two LSBs. The result is also converted into an ASCII string and send to the Serial port. The KuttyPy IDE will receive and display the data.

```
#include <avr/kp.h>   // Include file for I/O operations
#include <stdlib.h>
int main (void)

    {
    uint16_t data;
    char a[6], *p;
    DDRB = 255;        // Configure port B as output
    adc_enable();
    uart_init(38400);
```

```
while (1)
    {
    data = read_adc(0);
    PORTB = data >> 2;    // convert 10 bit in to 8 bit
    utoa(data, a, 10);    // convert to ASCII string
    p = a;
    while(*p) uart_send_byte(*p++);
    uart_send_byte('\n');
    delay_ms(500);
   }
}
```

4.6.2 Programmig ADC registers

The operation of the ADC is controlled mainly by the registers ADCSRA and ADMUX. Setting ADEN will enable the ADC and setting ADSC will start a conversion. The bit ADIF is set after a conversion and this bit can be cleared by writing a '1' to it. The ADSP bits decide the speed of operation of the ADC, by pre-scaling the clock input. The channel number is selected by the MUX0 to MUX4 bits in the ADMUX rregister. The reference input is selected by the REFS0 and REFS1 bits.

7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

The program adc-read-direct.c, demonstrates the usage of these registers. If the input is left unconnected, the noise level on the input is converted and displayed value could be anywhere between 0 and 255 (in the 8 bit mode).

```
#include <avr/io.h>
main()                    // convert channel 0, set pre-scaler to 7
{
DDRB = 255;
ADCSRA = (1 << ADEN) |  7;              // Enable ADC, set clock pre-scaler
ADMUX =  (1 << REFS0) | (1 << ADLAR);   // AVCC ref, Left adjust, read channel
while (1)
  {
  ADCSRA |=  (1 <<ADSC);                // Start ADC
  while ( !(ADCSRA & (1<<ADIF)) ) ;  // wait for ADC conversion
  ADCSRA |=  (1 <<ADIF);                //reset ADIF bit
  PORTB = ADCH;                         // 8 bit result to port B
  }
}
```

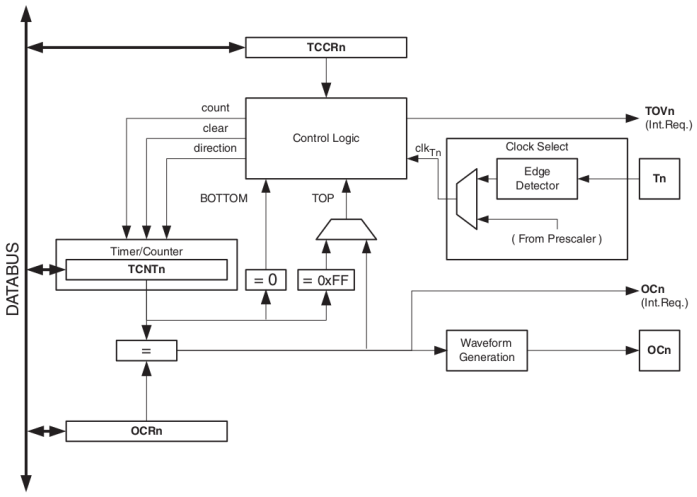


Figure 4.2: 8 bit Timer/Counter0 Schematic

4.7 Timer/Counters

ATmega16 has three counter/timer units. Two of them are of 8 bit size and one is 16 bit. The counter input could be derived from the internal clock or from an external source. The output of the counter is compared with setpoint registers and different types of actions are taken on compare match. The mode of operation of Counter/Timer is programmed by setting the bits in the control registers. These circuits are useful for time interval measurements and generating different kinds of waveforms.

4.7.1 8 bit Timer/Counter0

A block diagram of Timer/Counter0 is shown in figure4.2. The counter TCNT0 gets its input and control signals from the control logic circuit. The counter output is compared with a Output Compare Register OCR0 and a compare match can trigger different types of actions, like generating a waveform on OC0 (pin 4 of Atmega32, same as PB3). The mode of operation is decided by the register TCCR0, shown below:

7	6	5	4	3	2	1	0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

The 8MHz system clock is divided by 8 (csb =2, refer to table below) to get a 1MHz input to the counter. The OCR0 register is set to 99. The mode bits are set such that the when the counter value reaches the OCR0, the output is toggled and counter is cleared. This will result in the waveform generator output toggles after every 100 clock cycles, giving a 5kHz sqaurewave on pin OC0 (PB3). You may view this on an oscilloscope. If you do not have one, connect a loudspeaker with a 100Ω series resistor from PB3 to ground. We have used expEYES for viewing and characterizing the waveforms generated by microHOPE.

Changing ocrval to 199 will give output 2.5kHz on the output. The output frequency is given by the relation

$$f = \frac{f_{clock}}{2.N.(1 + OCR0)}$$

where f_{clock} is the system clock and N is the clock division factor, as shown below.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/(No prescaling)
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.

The library contains the following functions to handle TC0.
Let us start using Timer/Counter0 with the help of the following functions.

- void sqwave_tc0(uint8_t csb, uint8_t ocrval) : generates a square wave on OC0, whose frequency is decided by the clock select bits (csb) and ocrval.
- void pwm_tc0(uint8_t csb, uint8_t ocrval)

This program **sq-tc0.c** listed below generates a square wave on PB0.

```
#include <avr/kp.h>
csb = 2;           // Clock select bits
ocrval = 99;       // Output Compare register vaule
int main()
{
sqwave_tc0(csb, ocrval);
}
```

The clock selection bits are set to 2, means the system clock is divided by 8, resulting in 1MHz. The Output Compare Register is set to 99. After counting 100 clock cycles the counter will be reset and the PB3 pin will toggle. One cycle takes 200 uS, resulting in a frequency of 5000 Hz. The output waveform is fed to SEElab oscilloscope and a screenshot is shown in figure4.3.a.

This function generates a Pulse Width Modulated waveform on OC0, whose frequency is decided by the clock select bits (csb) and the duty cycle by the ocrval. The output OC0 is cleared when the counter reaches the OCR0 value, the counter proceeds upto 255 and then sets OC0. The program **pwm-tc0.c** generates a 3.9 kHz PWM with 25% dutycycle.

```
uint8_t  csb = 2;   // Clock select bits uint8_t
ocrval = 63;       // Output Compare register vaule
int main()
{
  pwm_tc0(csb, ocrval);
}
```

PWM waveforms are often used for generating analog DC voltages, in 0 to 5 volts range, by filtering it using an RC circuit. It is better to set a higher frequency so that the filter RC value could be small. The frequency can be made 31.25kHz by setting csb=1. The DC level is decided by the value of OCR0, ranging from 0 to 255. Once you learn howto manipulate the control registers, the same thing can be done without calling the library function, as shown below.

Figure 4.3: (a)Squarewave on TC0(b)PWM waveform on TC0

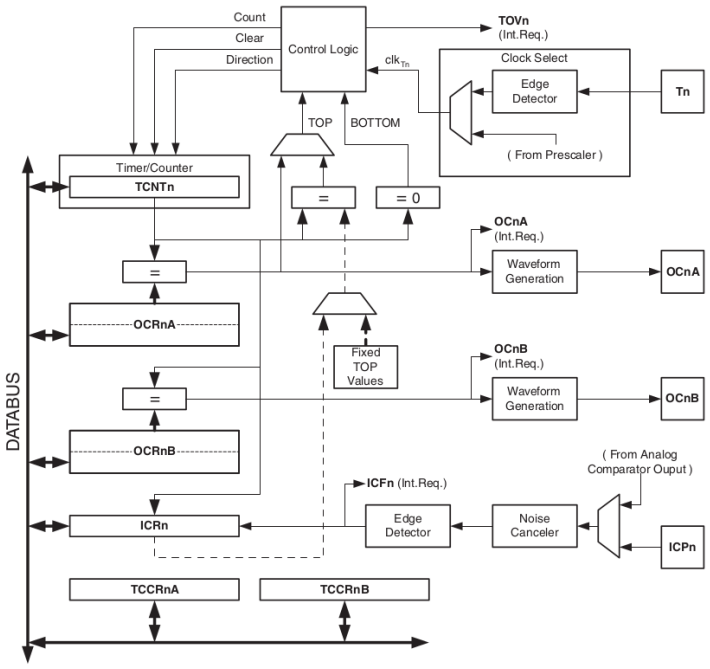
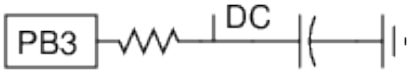


Figure 4.4: 16 bit Timer/Counter1 schematic

```
// example : pwm-tc0-direct.c
#include <avr/io.h>
uint8_t csb = 1;     // Clock select bits uint8_t
uint8_t ocrval = 255/4;      // Output Compare register vaule
int main()
{
// Set TCCR0 in the Fast PWM mode
TCCR0 =(1 << WGM01) | (1 << WGM00) | (1 << COM01) | csb;
OCR0 = ocrval;
TCNT0 = 0;
DDRB |= (1 << PB3); // Set PB3(OC0) as output
}
```

Connect a 1k resistor and 100uF capacitor in series from PB3 to ground,as shown below, and measure the voltage across the capacitor using a voltmeter.



4.7.2 16 bit Timer/Counter1

The Timer/Counter1 has more features like two Output Compare Registers, Input Capture unit etc., as shown in figure4.4. The frequency and duty cycle of the waveforms can be controlled better due to the 16 bit size of the counters. The following functions are available from the KuttyPy library.

- void sqwave_tc1(uint8_t csb, uint16_t ocra) : generates a square wave on OCA1 (PD5), whose frequency is decided by the clock select bits (csb) and ocra.
- void pwm10_tc1(uint8_t csb, uint16_t ocra) :

The example program **sq-tc1.c** generates a 5 kHz square wave.

```
#include <avr/kp.h>
uint8_t  csb = 2;        // 2 is divide by 8 option, 1MHz clock in
uint16_t  ocra = 50000;  // Output Compare register A
int main()

    {
    sqwave_tc1(csb, ocra);
    }
```

pwm10_tc1(csb, OCRA)

This function generates a PWM waveform with 10bit resolution. The value of ocra should be from 0 to 1023 to set the duty cycle.

```
// example : pwm-tc1.c
#include <avr/kp.h>
uint8_t  csb = 1;        // 1 => 8MHz clock in
uint16_t  ocra = 1024/3;  // Duty cycle arounf 33%
int main()
  {
  pwm10_tc1(csb, ocra);
  }
```

4.7.3 8 bit Timer/Counter2

This one is similar to Timer/Counter0.

sqwave_tc2(uint32_t freq)

This function generates a square wave on OC2. The clock selction bits and the OCR2 value are calculated. It is not possible to set all frequency values using this method. The actual frequency set is returned and displayed on the LCD.

```
//Example sq-tc2.c
#include <avr/kp.h>
int main()
{
f = set_sqr_tc2(1500);
}
```

PWM by programming the registers

The example given below demonstrates the usage of TC2 as a PWM waveform generator, by setting the control register bits. The duty cycle is set to 25% by setting the OCR2 to one fourth of the maximum.

```
// example : pwm-tc2.c
#include <avr/io.h>
```

```
uint8_t csb = 2;     // Clock select bits uint8_t
ocrval = 255/4;      // Output Compare register vaule
int main()
{
// Set TCCR2 in the Fast PWM mode
TCCR2 =(1 << WGM21) | (1 << WGM20) | (1 << COM21) | csb;
OCR2 = ocrval;
TCNT0 = 0;
DDRD |= (1 << PD7); // Set PD7(OC2) as output
}
```

Chapter 5

Example programs

5.0.1 Voltmeter

An external voltage connected to the ADC channel 0 (PA0) is measured periodically and displayed on the LCD display. The reference selected is AVCC. The 10 bit data is shifted twice to make it 8 bit and displayed on port B LEDs. The program voltmeter.c is listed below.

```
#include <avr/kp.h>   // Include file for I/O operations
int main (void)
{
uint16_t data;
float v;
DDRB = 255;              // Configure port B as output
adc_enable();
lcd_init();
while (1)
    {
     data = read_adc(0);
     PORTB = data >> 2;     // convert 10 bit in to 8 bit and send to port B
     v = data * 5.0 / 1023;
     lcd_clear();
     lcd_put_float(v, 3);   // 3 decimals
     delay_ms(500);
    }
}
```

5.0.2 Temperature Control

A temperature monitor/controller can be made using the LM35 temperature sensor and a heater controlled by a relay connected to a digital output pin. Connect LM35 output to PA0. At 100^0C , the output of LM35 will be 1 volt. With the internal 2.56 volts as reference, the ADC output will be around 400 (1.0 / 2.56 * 1023).

Drive the relay contact controlling the heater from PB0, via a transistor. Insert the following line in the beginning

```
DDRB = 1
```

and within the loop:

```
if (data > 400)       // switch off heater
    PORTB = 0;
else if (data < 395)   // switch on heater
    PORTB = 1;
```

The heater will be switched OFF when the ADC output is greater than 400. It will be switched ON only when the output goes below 395. The window of 6 is given to avoid the relay chattering.

5.0.3 A simple Oscilloscope

The program **cro.c** can waits for a command byte from the PC. On receiving a '1', it digitizes the input at PA0 500 times, with 100 microseconds in between samples, and sends the data to the PC. The program **cro.py** sends the necessary command, receives the data and displays it as shown in the figure5.1. The C program running on the micro-controller is listed below.

```
#include <avr/io.h>
#define READBLOCK  1    // code for readblock is 1
#define NS 500  // upto1800 for ATmega32
#define TG 100  // 100 usec between samples
uint8_t tmp8, dbuffer[NS];
uint16_t tmp16;
int main (void)
{
// UART at 38400 baud, 8, 1stop, No parity
UCSRB = (1 << RXEN) | (1 << TXEN);   UBRRH = 0;
UBRRL = 12;
UCSRC = (1 <<URSEL) | (1 << UCSZ1) | (1 << UCSZ0);
ADCSRA = (1 << ADEN); // Enable ADC
for(;;)
    {
    while ( !(UCSRA & (1<<RXC)) ); // wait for the PC
    if(UDR == 1)  // '1' is our command
      {
      TCCR1B = (1 << CS11);
      ADMUX = (1 << REFS0) |(1 << ADLAR) | 0;
      ADCSRA |= ADIF;
      for(tmp16 = 0; tmp16 < NS; ++tmp16)
        {
        TCNT1 = 1; // counter for TG
        ADCSRA |= (1 << ADSC) | 1;        // Start ADC
        while ( !(ADCSRA & (1<<ADIF)) );  // Done ?
        dbuffer[tmp16] = ADCH; // Collect Data
        ADCSRA |= ADIF;        // reset ADC DONE flag
```

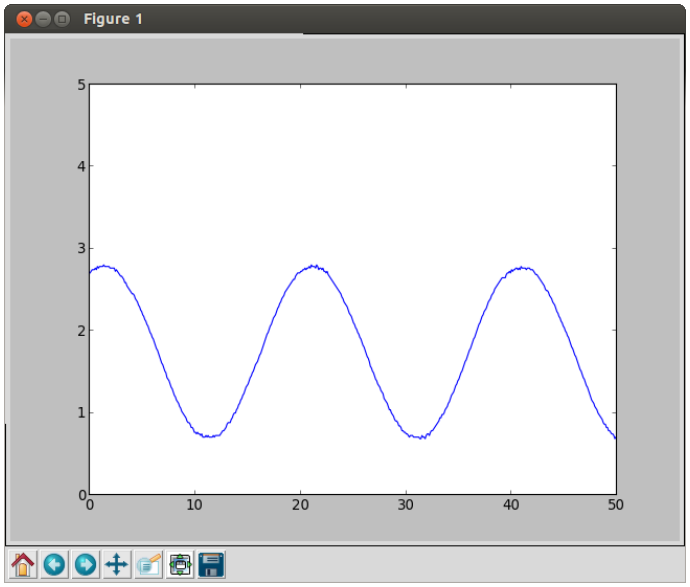


Figure 5.1: Oscilloscope screen shot

```
    while(TCNT1L < TG) ;   // Wait TG usecs
        }
    while( !(UCSRA & (1 <<UDRE) ) );  // Wait Tx empty
    UDR = 'D';                        // Send a 'D' first
    for(tmp16=0; tmp16 < NS; ++tmp16) // Send to the PC
    {
    while( !(UCSRA & (1 <<UDRE) ) );
    UDR = dbuffer[tmp16];
    }
     }
   }
}
```

The Python program cro.py

```
import serial, struct, time
import numpy as np
import matplotlib.pyplot as plt
NP = 500
TG = 100
fd=serial.Serial('/dev/ttyACM0',38400,stopbits=1,timeout = 1.0)
fd.flush()
time.sleep(1)
fig=plt.figure()
plt.axis([0, NP*TG/1000, 0, 5])
plt.ion()
plt.show()
va = ta = range(NP)
line, = plt.plot(ta,va)
while True:
fd.write(b'\x01') # 1 is the readblock command for uC end
```

```
print (fd.read()) # This must be a 'D'
data = fd.read(NP)
raw = struct.unpack('B'* NP, data)  # 8 bit data in byte array
ta = []
va = []
for i in range(NP):
ta.append(0.001 * i * TG)
# convert time from microseconds to milliseconds  va.append(raw[i] * 5.0 / 255
line.set_xdata(ta)
line.set_ydata(va)
plt.draw()
plt.pause(0.05)
```

5.0.4 Frequency Counter

Timer/Counter can be used for timing applications, like measuring the time elapsed between two events or counting the number of pulse inputs during a specified time interval. The program **freq-counter.c** measures the frequency of the pulse connected to PB0 and displays it on the LCD display. A 1000 Hz pulse is generated on PD7.

```
#include <avr/kp.h>   // Include file for libkp
int main()
{
  uint32_t f;
  set_sqr_tc2(1000);    // Set a square wave on TC2 output (PD7)
  lcd_init();
  while(1)
     {
     f = measure_freq();   // Measures on T1 (PB1)
     lcd_clear();
     lcd_put_string("f=");
     lcd_put_long(f);
     delay_ms(200);
  }
}
```

Connect PD7 to PB1 and upload the program **freq-counter.c** to read the frequency on the LCD display. You can also connect PB1 to an external pulse source to measure its frequency. The maximum frequency that can be measured is limited by the size of the counter, that is 63535, means we it can handle upto around 126 kHz.

5.0.4.1 Distance Measurement

This technique is used for measuring distance using an ultrasound echo module HY-SR04, using **ultra-sound-echo.c**. The trigger is connected to PB0 and the echo is connected to PB1.

```
#include <avr/kp.h>
int vsby2 = 17;  // velocity of sound in air = 34 mS/cm
```

```
int main()
{
uint32_t x;
DDRB |=  (1 << PB0);  // set PB0 as output
DDRB &= ~(1 << PB1);  // and PB1 as inpt
lcd_init();
while(1)
   {
   PORTB |=  (1 << PB0);   // set PB0 HIGH
   delay_100us(1);
   PORTB &=  ~(1 << PB0);  // set PB0 LOW
   delay_100us(5);         // Wait for a while to avoid false triggering
   start_timer();
   while( (PINB & 2) != 0 ) ;   // Wait for LOW on PB1
   x = read_timer() + 400;
   lcd_clear();
   lcd_put_long(x*vsby2/1000);  // distance in cm
   delay_ms(500);
   }
}
```