STMIK ATMA LUHUR

KARTU BIMBINGAN SKRIPSI

NIM : 1111 5000 75
NAMA : FIRMANSYAH
DOSEN PEMBIMBING : YLIRINDRA, MT
JUDUL SKRIPSI : PINTU OTOMATIS DENGAN BARCODE SCANNER BERBASIS MIKROKONTROLER ATMEGA 16

No.	TANGGAL	MATERI	PARAF DOSEN
1.	03/03/2015	Konsultasi Judul	
2.	09/03/2015	Bimbingan BAB I	
3.	23/03/2015	Revisi BAB I	
4.	26/03/2015	Bimbingan BAB II	
5.	31/03/2015	Revisi BAB II	
6.	01/04/2015	Bimbingan BAB III	
7.	06/04/2015	Revisi BAB III	
8.	15/04/2015	Bimbingan BAB IV	
9.	20/04/2015	Revisi Analisa BAB IV	
10.	29/04/2015	Revisi Perancangan BAB IV	
11.	27/04/2015	Revisi BAB V	
12	09/05/2015	Bimbingan Program dan Prototype	
13.	01/06/2015	Bimbingan Program dan Prototype	
14.	15/06/2015	Demo Program dan prototype	
15.			

Mahasiswa diatas telah melakukan bimbingan dengan jumlah materi yang telah mencukupi untuk disidangkan.

Pangkalpinang, 22 JUNI 2015

Mahasiswa

(FIRMANSYAH)

Dosen Pembimbing

(Yulindra, MT)

Features

- High-performance, Low-power Atmel® AVR® 8-bit Microcontroller
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single-clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 16 MIPS Throughput at 16 MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory segments
  - 16 Kbytes of In-System Self-programmable Flash program memory
  - 512 Bytes EEPROM
  - 1 Kbyte Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C(1)
  - Optional Boot Code Section with Independent Lock Bits
    In-System Programming by On-chip Boot Program
    True Read-While-Write Operation
  - Programming Lock for Software Security
- JTAG (IEEE std. 1149.1 Compliant) Interface
  - Boundary-scan Capabilities According to the JTAG Standard
  - Extensive On-chip Debug Support
  - Programming of Flash, EEPROM, Fuses, and Lock Bits through the JTAG Interface
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Four PWM Channels
  - 8-channel, 10-bit ADC
    8 Single-ended Channels
    7 Differential Channels in TQFP Package Only
    2 Differential Channels with Programmable Gain at 1x, 10x, or 200x
  - Byte-oriented Two-wire Serial Interface
  - Programmable Serial USART
  - Master/Slave SPI Serial Interface
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
- Special Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated RC Oscillator
  - External and Internal Interrupt Sources
  - Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby and Extended Standby
- I/O and Packages
  - 32 Programmable I/O Lines
  - 40-pin PDIP, 44-lead TQFP, and 44-pad QFN/MLF
- Operating Voltages
  - 2.7V - 5.5V for ATmega16L
  - 4.5V - 5.5V for ATmega16
- Speed Grades
  - 0 - 8 MHz for ATmega16L
  - 0 - 16 MHz for ATmega16
- Power Consumption @ 1 MHz, 3V, and 25°C for ATmega16L
  - Active: 1.1 mA
  - Idle Mode: 0.35 mA
  - Power-down Mode: < 1 µA

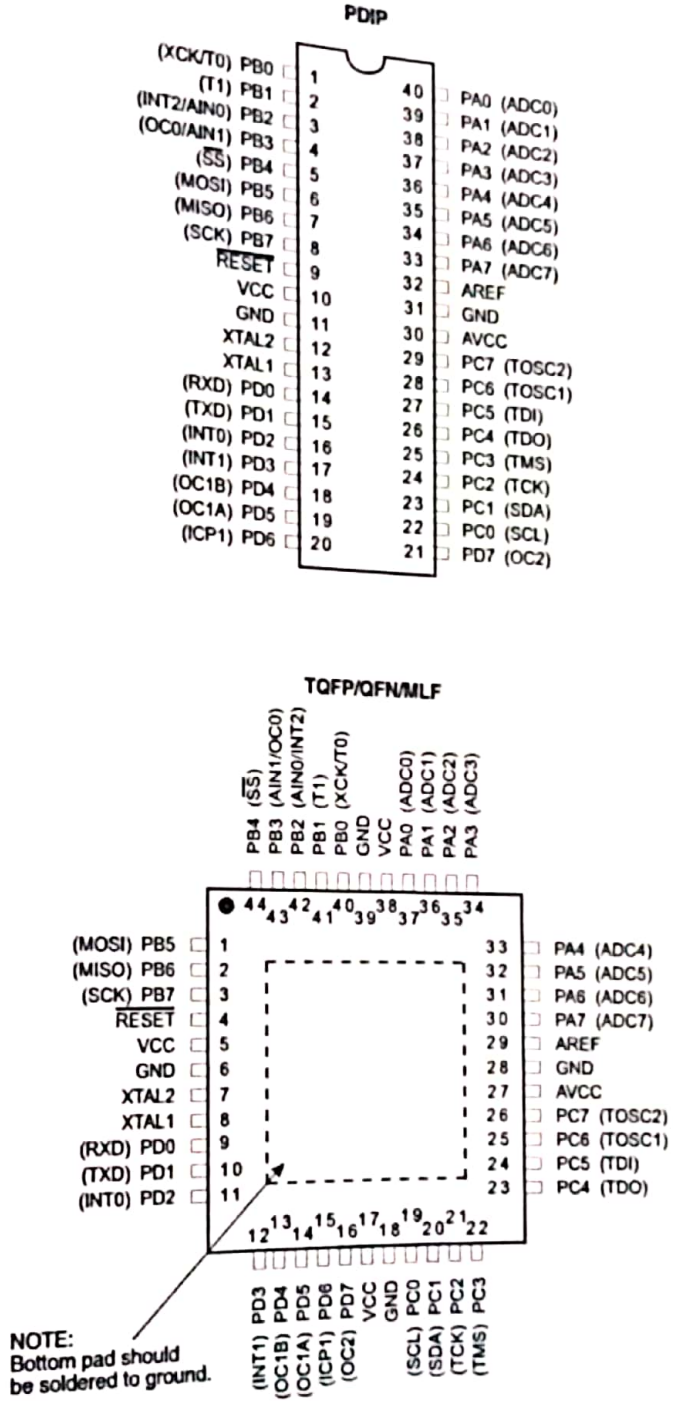
8-bit AVR® Microcontroller with 16K Bytes In-System Programmable Flash

ATmega16
ATmega16L

Rev. 2466T–AVR–07/10

Pin Configurations

Figure 1. Pinout ATmega16



Disclaimer

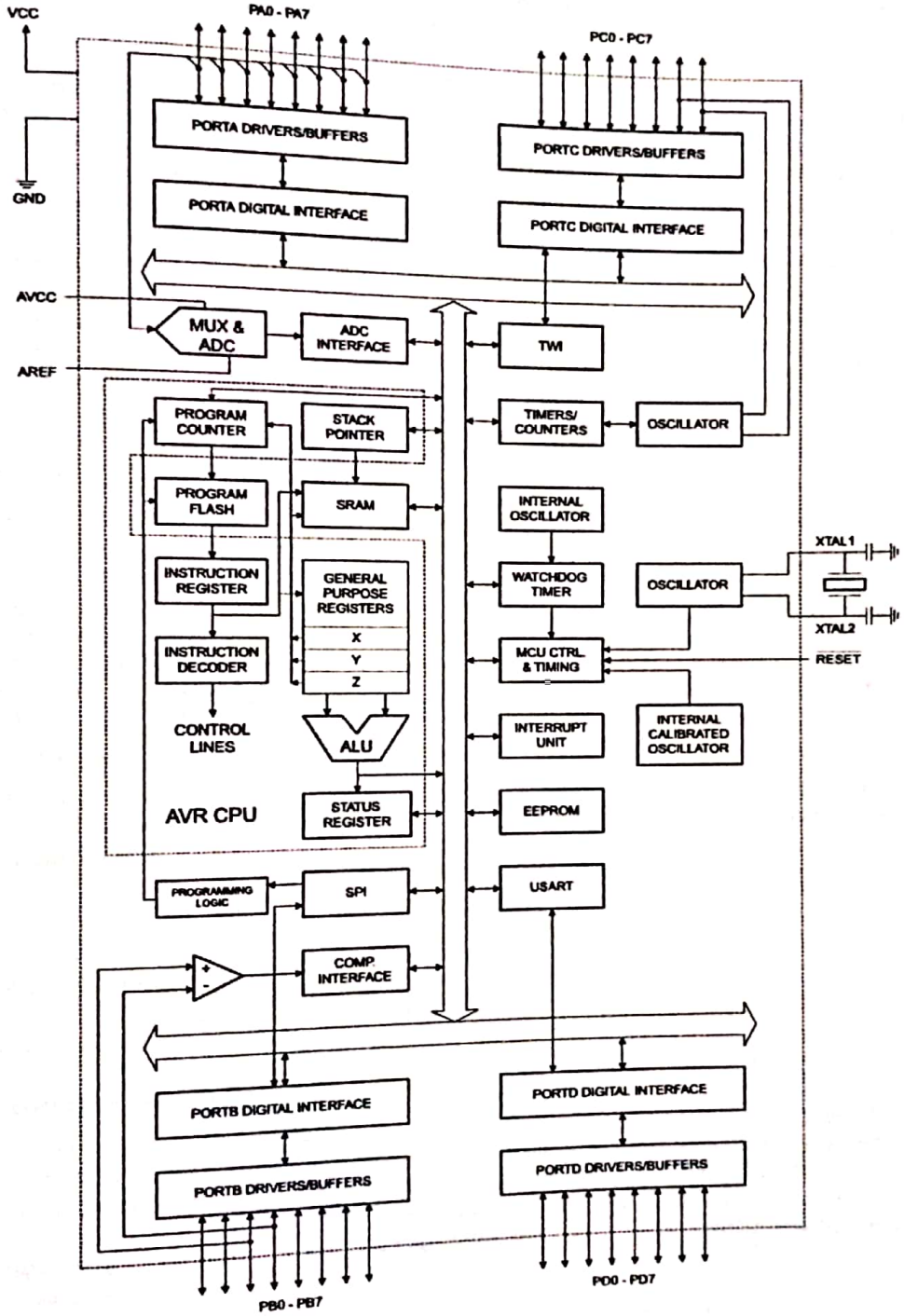
Typical values contained in this datasheet are based on simulations and characterization of other AVR microcontrollers manufactured on the same process technology. Min and Max values will be available after the device is characterized.

Overview

The ATmega16 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega16 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

Block Diagram

Figure 2. Block Diagram



The AVR core combines a rich instruction set with 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

The ATmega16 provides the following features: 16 Kbytes of In-System Programmable Flash Program memory with Read-While-Write capabilities, 512 bytes EEPROM, 1 Kbyte SRAM, 32 general purpose I/O lines, 32 general purpose working registers, a JTAG interface for Boundary-scan, On-chip Debugging support and programming, three flexible Timer/Counters with compare modes, Internal and External Interrupts, a serial programmable USART, a byte oriented Two-wire Serial Interface, an 8-channel, 10-bit ADC with optional differential input stage with programmable gain (TQFP package only), a programmable Watchdog Timer with Internal Oscillator, an SPI serial port, and six software selectable power saving modes. The Idle mode stops the CPU while allowing the USART, Two-wire interface, A/D Converter, SRAM, Timer/Counters, SPI port, and interrupt system to continue functioning. The Power-down mode saves the register contents but freezes the Oscillator, disabling all other chip functions until the next External Interrupt or Hardware Reset. In Power-save mode, the Asynchronous Timer continues to run, allowing the user to maintain a timer base while the rest of the device is sleeping. The ADC Noise Reduction mode stops the CPU and all I/O modules except Asynchronous Timer and ADC, to minimize switching noise during ADC conversions. In Standby mode, the crystal/resonator Oscillator is running while the rest of the device is sleeping. This allows very fast start-up combined with low-power consumption. In Extended Standby mode, both the main Oscillator and the Asynchronous Timer continue to run.

The device is manufactured using Atmel's high density nonvolatile memory technology. The On-chip ISP Flash allows the program memory to be reprogrammed in-system through an SPI serial interface, by a conventional nonvolatile memory programmer, or by an On-chip Boot program running on the AVR core. The boot program can use any interface to download the application program in the Application Flash memory. Software in the Boot Flash section will continue to run while the Application Flash section is updated, providing true Read-While-Write operation. By combining an 8-bit RISC CPU with In-System Self-Programmable Flash on a monolithic chip, the Atmel ATmega16 is a powerful microcontroller that provides a highly-flexible and cost-effective solution to many embedded control applications.

The ATmega16 AVR is supported with a full suite of program and system development tools including: C compilers, macro assemblers, program debugger/simulators, in-circuit emulators, and evaluation kits.

Pin Descriptions

VCC

Digital supply voltage.

GND

Ground.

Port A (PA7..PA0)

Port A serves as the analog inputs to the A/D Converter.

Port A also serves as an 8-bit bi-directional I/O port, if the A/D Converter is not used. Port pins can provide internal pull-up resistors (selected for each bit). The Port A output buffers have symmetrical drive characteristics with both high sink and source capability. When pins PA0 to PA7 are used as inputs and are externally pulled low, they will source current if the internal pull-up resistors are activated. The Port A pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port B also serves the functions of various special features of the ATmega16 as listed on page 58.

Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running. If the JTAG interface is enabled, the pull-up resistors on pins PC5(TDI), PC3(TMS) and PC2(TCK) will be activated even if a reset occurs.

Port C also serves the functions of the JTAG interface and other special features of the ATmega16 as listed on page 61.

Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port D output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running.

Port D also serves the functions of various special features of the ATmega16 as listed on page 63.

Reset Input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running. The minimum pulse length is given in Table 15 on page 38. Shorter pulses are not guaranteed to generate a reset.

Input to the inverting Oscillator amplifier and input to the internal clock operating circuit.

Output from the inverting Oscillator amplifier.

AVCC is the supply voltage pin for Port A and the A/D Converter. It should be externally connected to V_{CC}, even if the ADC is not used. If the ADC is used, it should be connected to V_{CC} through a low-pass filter.

AREF is the analog reference pin for the A/D Converter.

Resources

A comprehensive set of development tools, application notes and datasheets are available for download on http://www.atmel.com/avr.

Data Retention

Reliability Qualification results show that the projected data retention failure rate is much less than 1 PPM over 20 years at 85°C or 100 years at 25°C.

About Code Examples

This documentation contains simple code examples that briefly show how to use various parts of the device. These code examples assume that the part specific header file is included before compilation. Be aware that not all C Compiler vendors include bit definitions in the header files and interrupt handling in C is compiler dependent. Please confirm with the C Compiler documentation for more details.

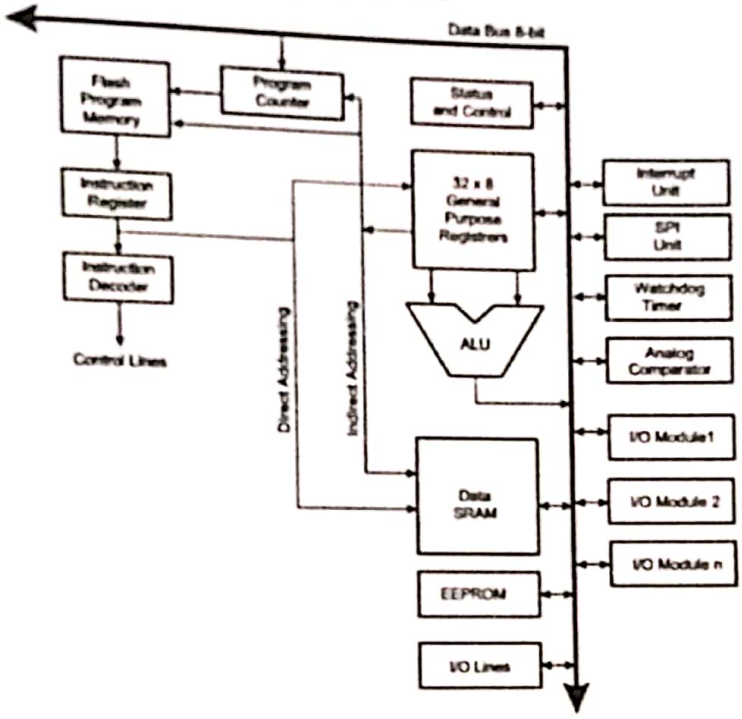
AVR CPU Core

Introduction

This section discusses the AVR core architecture in general. The main function of the CPU core is to ensure correct program execution. The CPU must therefore be able to access memories, perform calculations, control peripherals, and handle interrupts.

Architectural Overview

Figure 3. Block Diagram of the AVR MCU Architecture



In order to maximize performance and parallelism, the AVR uses a Harvard architecture – with separate memories and buses for program and data. Instructions in the program memory are executed with a single level pipelining. While one instruction is being executed, the next instruction is pre-fetched from the program memory. This concept enables instructions to be executed in every clock cycle. The program memory is In-System Reprogrammable Flash memory.

The fast-access Register File contains 32 × 8-bit general purpose working registers with a single clock cycle access time. This allows single-cycle Arithmetic Logic Unit (ALU) operation. In a typical ALU operation, two operands are output from the Register File, the operation is executed, and the result is stored back in the Register File – in one clock cycle.

Six of the 32 registers can be used as three 16-bit indirect address register pointers for Data Space addressing – enabling efficient address calculations. One of the these address pointers can also be used as an address pointer for look up tables in Flash Program memory. These added function registers are the 16-bit X-register, Y-register, and Z-register, described later in this section.

The ALU supports arithmetic and logic operations between registers or between a constant and a register. Single register operations can also be executed in the ALU. After an arithmetic operation, the Status Register is updated to reflect information about the result of the operation.

Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space. Most AVR instructions have a single 16-bit word format. Every program memory address contains a 16-bit or 32-bit instruction.

Program Flash memory space is divided in two sections, the Boot program section and the Application Program section. Both sections have dedicated Lock bits for write and read/write protection. The SPM instruction that writes into the Application Flash memory section must reside in the Boot Program section.

During interrupts and subroutine calls, the return address Program Counter (PC) is stored on the Stack. The Stack is effectively allocated in the general data SRAM, and consequently the Stack size is only limited by the total SRAM size and the usage of the SRAM. All user programs must initialize the SP in the reset routine (before subroutines or interrupts are executed). The Stack Pointer SP is read/write accessible in the I/O space. The data SRAM can easily be accessed through the five different addressing modes supported in the AVR architecture.

The memory spaces in the AVR architecture are all linear and regular memory maps.

A flexible interrupt module has its control registers in the I/O space with an additional global interrupt enable bit in the Status Register. All interrupts have a separate interrupt vector in the interrupt vector table. The interrupts have priority in accordance with their interrupt vector position. The lower the interrupt vector address, the higher the priority.

The I/O memory space contains 64 addresses for CPU peripheral functions as Control Registers, SPI, and other I/O functions. The I/O Memory can be accessed directly, or as the Data Space locations following those of the Register File, $20 - $5F.

ALU – Arithmetic Logic Unit

The high-performance AVR ALU operates in direct connection with all the 32 general purpose working registers. Within a single clock cycle, arithmetic operations between general purpose registers or between a register and an immediate are executed. The ALU operations are divided into three main categories – arithmetic, logical, and bit-functions. Some implementations of the architecture also provide a powerful multiplier supporting both signed/unsigned multiplication and fractional format. See the "Instruction Set" section for a detailed description.

Status Register

The Status Register contains information about the result of the most recently executed arithmetic instruction. This information can be used for altering program flow in order to perform conditional operations. Note that the Status Register is updated after all ALU operations, as specified in the Instruction Set Reference. This will in many cases remove the need for using the dedicated compare instructions, resulting in faster and more compact code.

The Status Register is not automatically stored when entering an interrupt routine and restored when returning from an interrupt. This must be handled by software.

The AVR Status Register – SREG – is defined as:

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	

- **Bit 7 – I: Global Interrupt Enable**

The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable Register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

- **Bit 6 – T: Bit Copy Storage**

The Bit Copy instructions BLD (Bit LoaD) and BST (Bit STore) use the T-bit as source or destination for the operated bit. A bit from a register in the Register File can be copied into T by the BST instruction, and a bit in T can be copied into a bit in a register in the Register File by the BLD instruction.

- **Bit 5 – H: *Half Carry Flag***

The Half Carry Flag H indicates a Half Carry in some arithmetic operations. Half Carry is useful in BCD arithmetic. See the "Instruction Set Description" for detailed information.

- **Bit 4 – S: Sign Bit, $S = N \oplus V$**

The S-bit is always an exclusive or between the Negative Flag N and the Two's Complement Overflow Flag V. See the "Instruction Set Description" for detailed information.

- **Bit 3 – V: Two's Complement Overflow Flag**

The Two's Complement Overflow Flag V supports two's complement arithmetics. See the "Instruction Set Description" for detailed information.

- **Bit 2 – N: Negative Flag**

The Negative Flag N indicates a negative result in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

- **Bit 1 – Z: Zero Flag**

The Zero Flag Z indicates a zero result in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

- **Bit 0 – C: Carry Flag**

The Carry Flag C indicates a carry in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

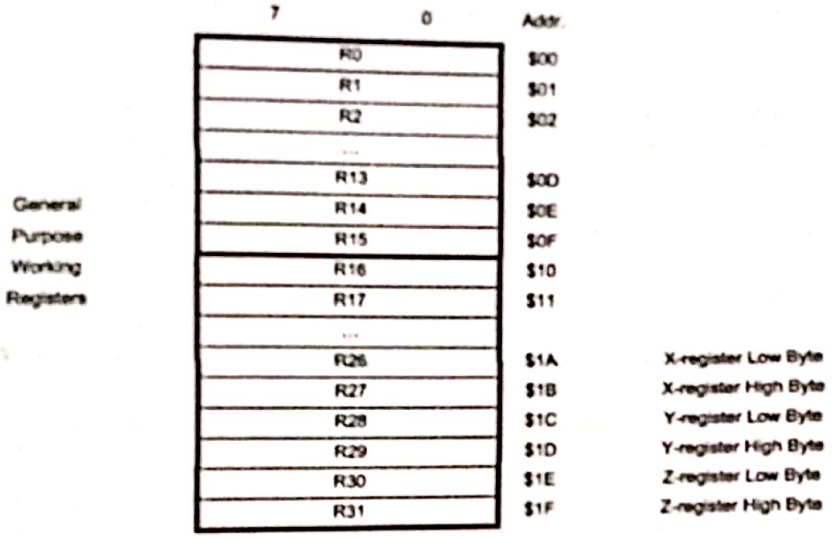
General Purpose Register File

The Register File is optimized for the AVR Enhanced RISC instruction set. In order to achieve the required performance and flexibility, the following input/output schemes are supported by the Register File:

- One 8-bit output operand and one 8-bit result input
- Two 8-bit output operands and one 8-bit result input
- Two 8-bit output operands and one 16-bit result input
- One 16-bit output operand and one 16-bit result input

Figure 4 shows the structure of the 32 general purpose working registers in the CPU.

Figure 4. AVR CPU General Purpose Working Registers



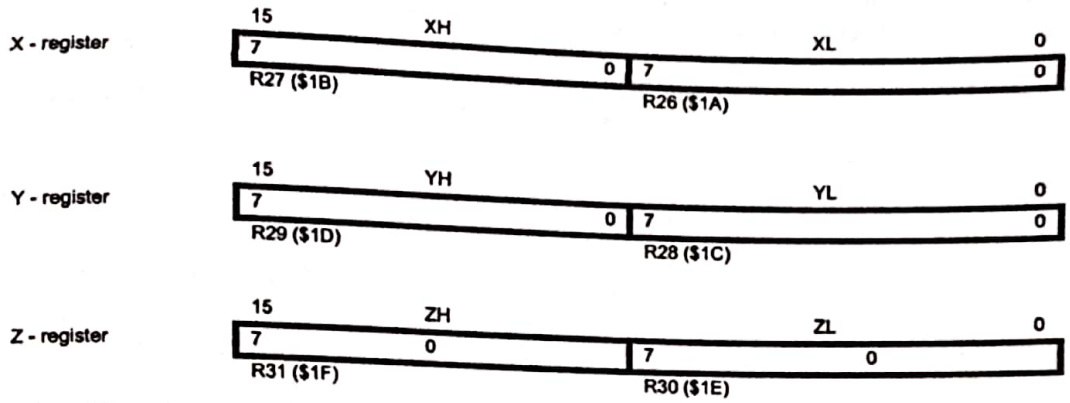
Most of the instructions operating on the Register File have direct access to all registers, and most of them are single cycle instructions.

As shown in Figure 4, each register is also assigned a data memory address, mapping them directly into the first 32 locations of the user Data Space. Although not being physically implemented as SRAM locations, this memory organization provides great flexibility in access of the registers, as the X-, Y-, and Z-pointer Registers can be set to index any register in the file.

The X-register, Y-register and Z-register

The registers R26..R31 have some added functions to their general purpose usage. These registers are 16-bit address pointers for indirect addressing of the Data Space. The three indirect address registers X, Y, and Z are defined as described in Figure 5.

Figure 5. The X-register, Y-register, and Z-register



In the different addressing modes these address registers have functions as fixed displacement, automatic increment, and automatic decrement (see the Instruction Set Reference for details).

Stack Pointer

The Stack is mainly used for storing temporary data, for storing local variables and for storing return addresses after interrupts and subroutine calls. The Stack Pointer Register always points to the top of the Stack. Note that the Stack is implemented as growing from higher memory locations to lower memory locations. This implies that a Stack PUSH command decreases the Stack Pointer. If software reads the Program Counter from the Stack after a call or an interrupt, unused bits (15:13) should be masked out.

The Stack Pointer points to the data SRAM Stack area where the Subroutine and Interrupt Stacks are located. This Stack space in the data SRAM must be defined by the program before any subroutine calls are executed or interrupts are enabled. The Stack Pointer must be set to point above $60. The Stack Pointer is decremented by one when data is pushed onto the Stack with the PUSH instruction, and it is decremented by two when the return address is pushed onto the Stack with subroutine call or interrupt. The Stack Pointer is incremented by one when data is popped from the Stack with the POP instruction, and it is incremented by two when data is popped from the Stack with return from subroutine RET or return from interrupt RETI.

The AVR Stack Pointer is implemented as two 8-bit registers in the I/O space. The number of bits actually used is implementation dependent. Note that the data space in some implementations of the AVR architecture is so small that only SPL is needed. In this case, the SPH Register will not be present.

Bit	15	14	13	12	11	10	9	8	
	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	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	
	0	0	0	0	0	0	0	0	

Instruction Execution Timing

This section describes the general access timing concepts for instruction execution. The AVR CPU is driven by the CPU clock clk_{CPU}, directly generated from the selected clock source for the chip. No internal clock division is used.

Figure 6 shows the parallel instruction fetches and instruction executions enabled by the Harvard architecture and the fast-access Register File concept. This is the basic pipelining concept to obtain up to 1 MIPS per MHz with the corresponding unique results for functions per cost, functions per clocks, and functions per power-unit.

Figure 6. The Parallel Instruction Fetches and Instruction Executions

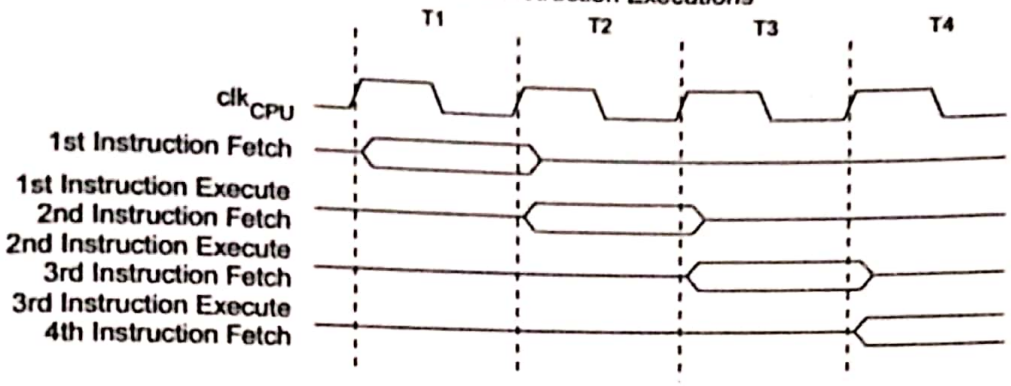
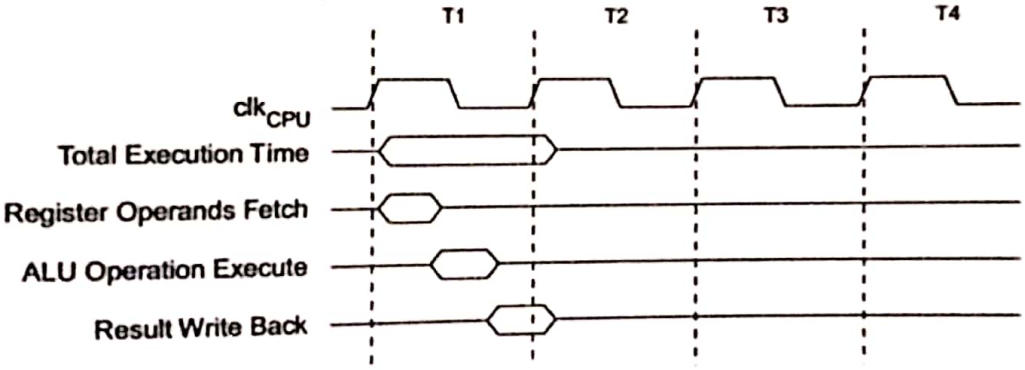


Figure 7 shows the internal timing concept for the Register File. In a single clock cycle an ALU operation using two register operands is executed, and the result is stored back to the destination register.

Figure 7. Single Cycle ALU Operation



Reset and Interrupt Handling

The AVR provides several different interrupt sources. These interrupts and the separate reset vector each have a separate program vector in the program memory space. All interrupts are assigned individual enable bits which must be written logic one together with the Global Interrupt Enable bit in the Status Register in order to enable the interrupt. Depending on the Program Counter value, interrupts may be automatically disabled when Boot Lock bits BLB02 or BLB12 are programmed. This feature improves software security. See the section "Memory Programming" on page 259 for details.

The lowest addresses in the program memory space are by default defined as the Reset and Interrupt Vectors. The complete list of vectors is shown in "Interrupts" on page 45. The list also determines the priority levels of the different interrupts. The lower the address the higher is the priority level. RESET has the highest priority, and next is INT0 – the External Interrupt Request

0. The Interrupt Vectors can be moved to the start of the Boot Flash section by setting the IVSEL bit in the General Interrupt Control Register (GICR). Refer to "Interrupts" on page 45 for more information. The Reset Vector can also be moved to the start of the boot Flash section by programming the BOOTRST Fuse, see "Boot Loader Support – Read-While-Write Self-Programming" on page 246.

When an interrupt occurs, the Global Interrupt Enable I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine. The I-bit is automatically set when a Return from Interrupt instruction – RETI – is executed.

There are basically two types of interrupts. The first type is triggered by an event that sets the Interrupt Flag. For these interrupts, the Program Counter is vectored to the actual Interrupt Vector in order to execute the interrupt handling routine, and hardware clears the corresponding Interrupt Flag. Interrupt Flags can also be cleared by writing a logic one to the flag bit position(s) to be cleared. If an interrupt condition occurs while the corresponding interrupt enable bit is cleared, the Interrupt Flag will be set and remembered until the interrupt is enabled, or the flag is cleared by software. Similarly, if one or more interrupt conditions occur while the Global Interrupt Enable bit is cleared, the corresponding Interrupt Flag(s) will be set and remembered until the global interrupt enable bit is set, and will then be executed by order of priority.

The second type of interrupts will trigger as long as the interrupt condition is present. These interrupts do not necessarily have Interrupt Flags. If the interrupt condition disappears before the interrupt is enabled, the interrupt will not be triggered.

When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served.

Note that the Status Register is not automatically stored when entering an interrupt routine, nor restored when returning from an interrupt routine. This must be handled by software.

When using the CLI instruction to disable interrupts, the interrupts will be immediately disabled. No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with the CLI instruction. The following example shows how this can be used to avoid interrupts during the timed EEPROM write sequence.

Assembly Code Example

```
in  r16, SREG       ; store SREG value
cli     ; disable interrupts during timed sequence
sbi EECR, EEMWE     ; start EEPROM write
sbi EECR, EEWE
out SREG, r16       ; restore SREG value (I-bit)
```

C Code Example

```
char cSREG;
cSREG = SREG;  /* store SREG value */
/* disable interrupts during timed sequence */
_CLI();
EECR |= (1<<EEMWE); /* start EEPROM write */
EECR |= (1<<EEWE);
SREG = cSREG; /* restore SREG value (I-bit) */
```

When using the SEI instruction to enable interrupts, the instruction following SEI will be executed before any pending interrupts, as shown in this example.

Assembly Code Example

```
sei   ; set global interrupt enable
sleep ; enter sleep, waiting for interrupt
; note: will enter sleep before any pending
; interrupt(s)
```

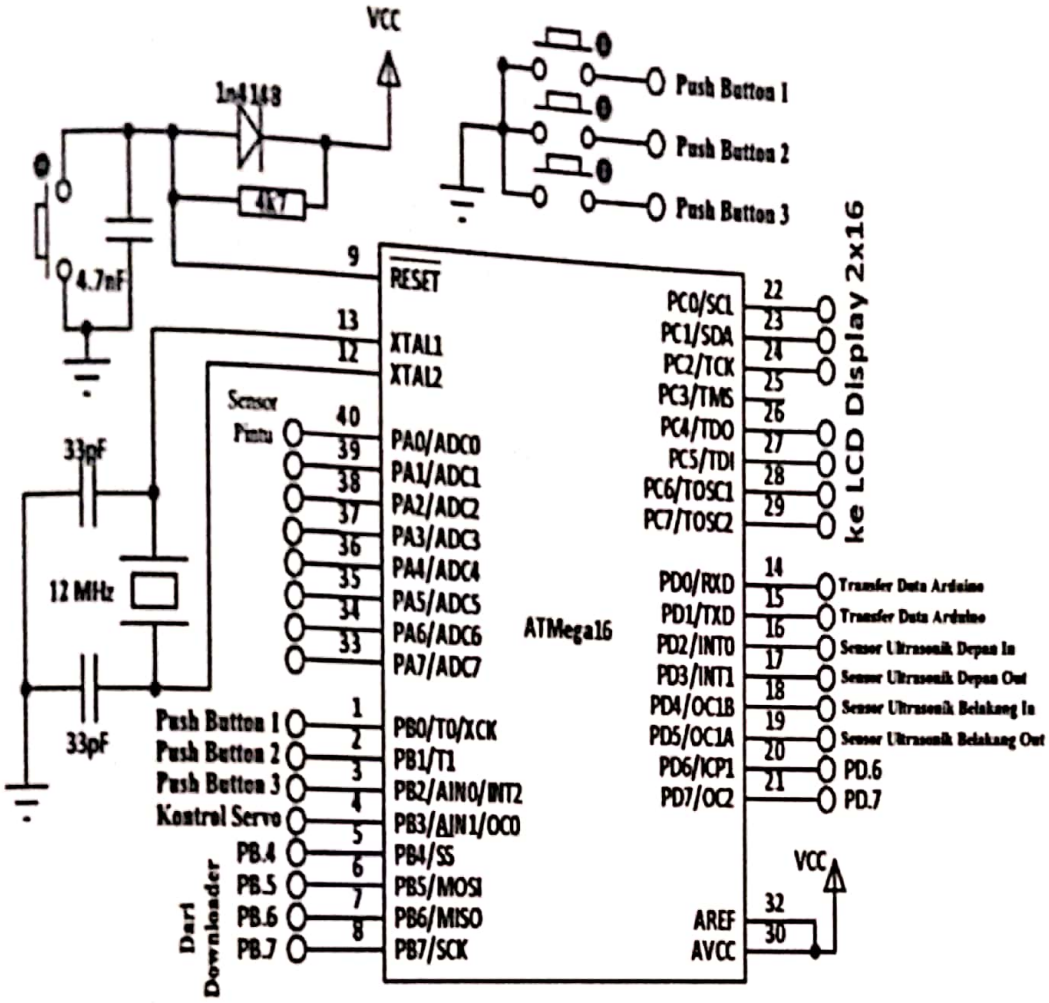
C Code Example

```
_SEI(); /* set global interrupt enable */
_SLEEP(); /* enter sleep, waiting for interrupt */
/* note: will enter sleep before any pending interrupt(s) */
```

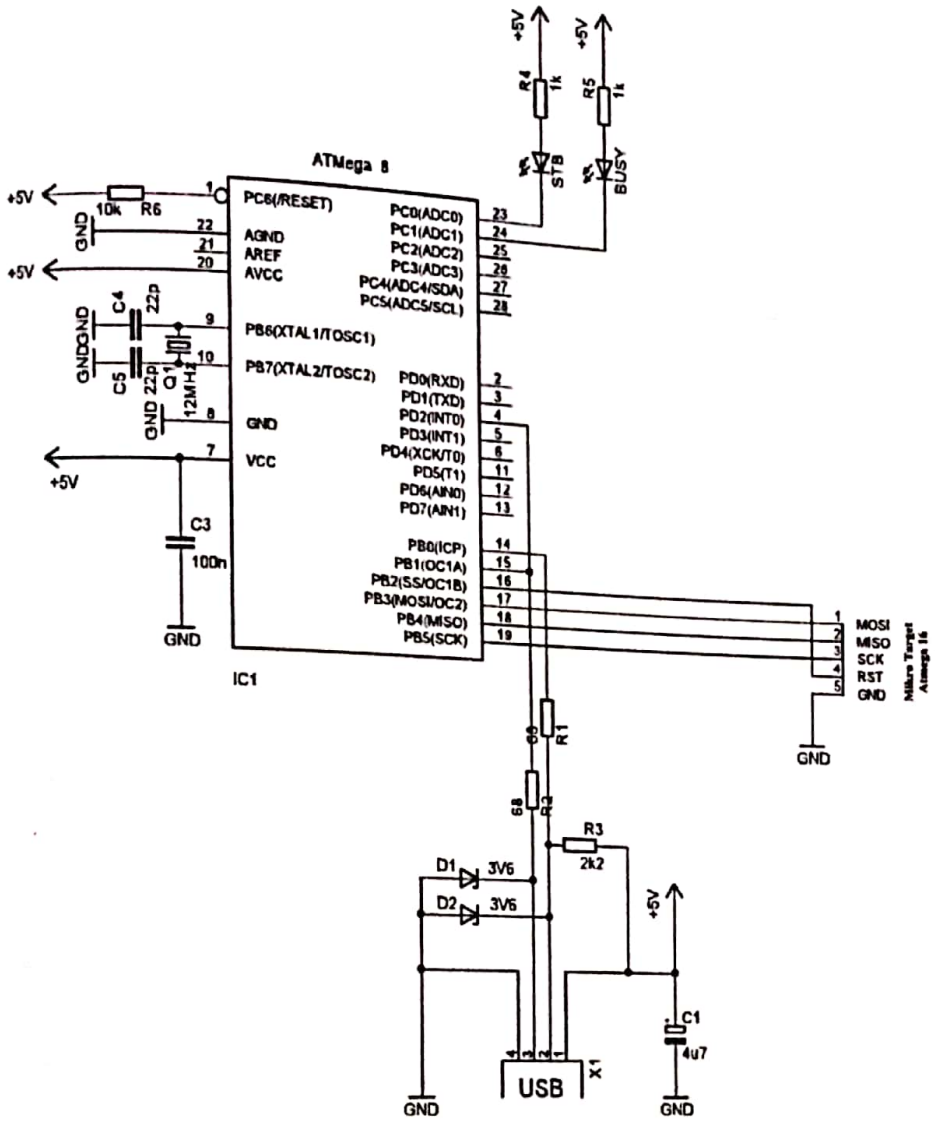
terrupt Response ime

The interrupt execution response for all the enabled AVR interrupts is four clock cycles minimum. After four clock cycles the program vector address for the actual interrupt handling routine is executed. During this four clock cycle period, the Program Counter is pushed onto the Stack. The vector is normally a jump to the interrupt routine, and this jump takes three clock cycles. If an interrupt occurs during execution of a multi-cycle instruction, this instruction is completed before the interrupt is served. If an interrupt occurs when the MCU is in sleep mode, the interrupt execution response time is increased by four clock cycles. This increase comes in addition to the start-up time from the selected sleep mode.

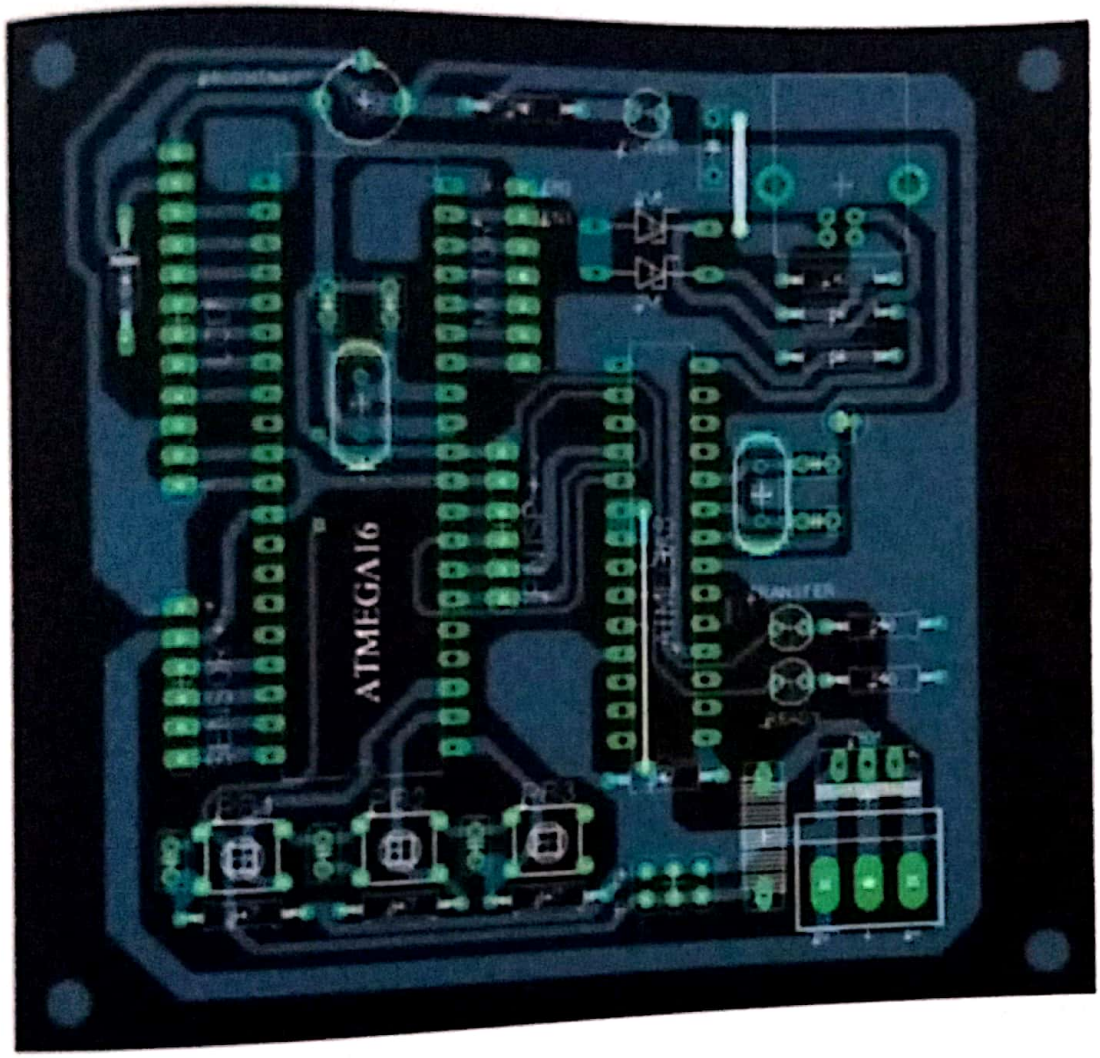
A return from an interrupt handling routine takes four clock cycles. During these four clock cycles, the Program Counter (two bytes) is popped back from the Stack, the Stack Pointer is incremented by two, and the I-bit in SREG is set.



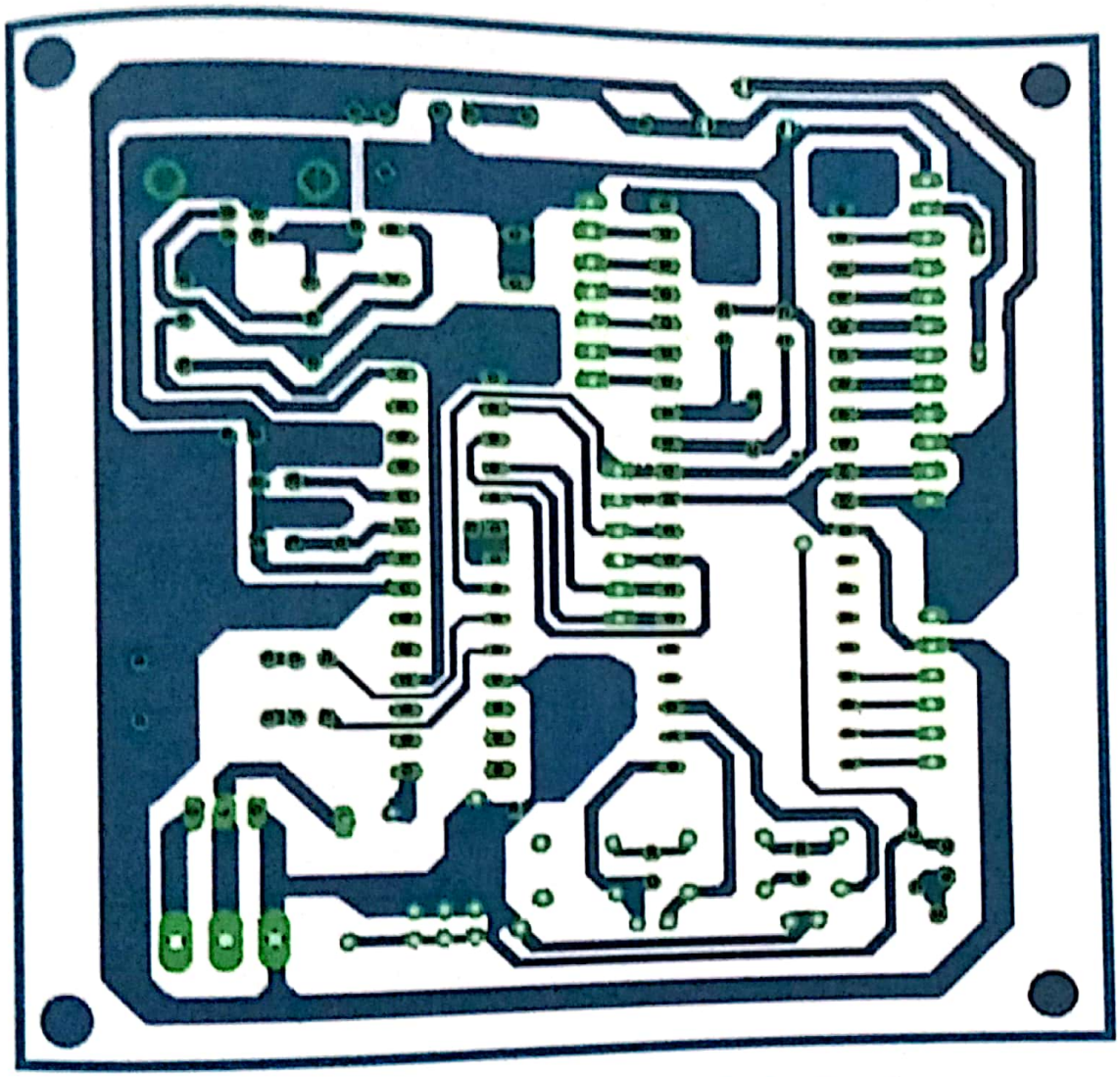
GAMBAR SKEMA RANGKAIAN SISTEM MINIMUM ATMEGA 16



GAMBAR SKEMA RANGKAIAN DOWNLOADER



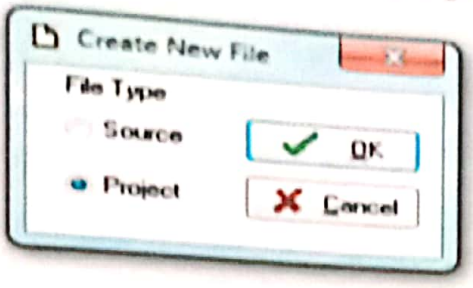
Gambar Layout ATMEGA 16 + Downloader Atas



Gambar Layout ATMEGA 16 + Downloader Bawah

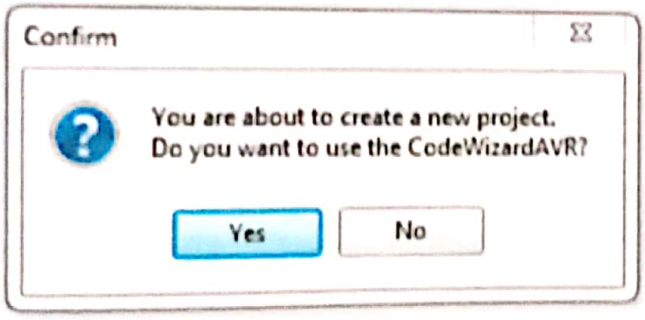
PROSES INISIALISASI PORT PADA APLIKASI CODE VISION AVR V2.05.3

Pada menu awal kita akan kita akan membuat suatu project dengan cara klik menu file dan pilih new maka akan tampil jendela pilihan seprti gambar 4.8



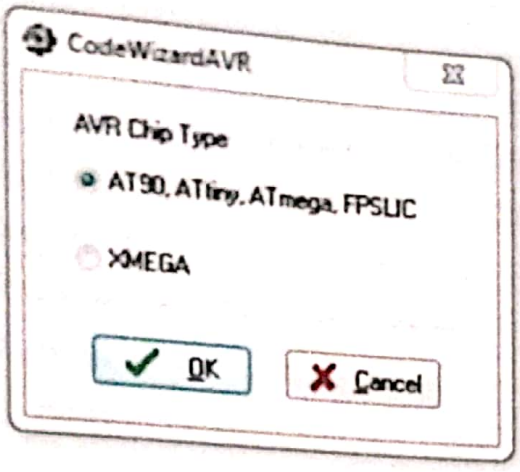
Gambar4.8Jendela Pilihan Tipe File

Untuk membuat program baru pilihlah **Project** kemudian klik **OK**, maka muncul pertanyaan yang menanyakan apakah kita ingin menggunakan CodeWizardAVR seperti pada gambar 4.9 berikut, kemudian pilih **Yes**.



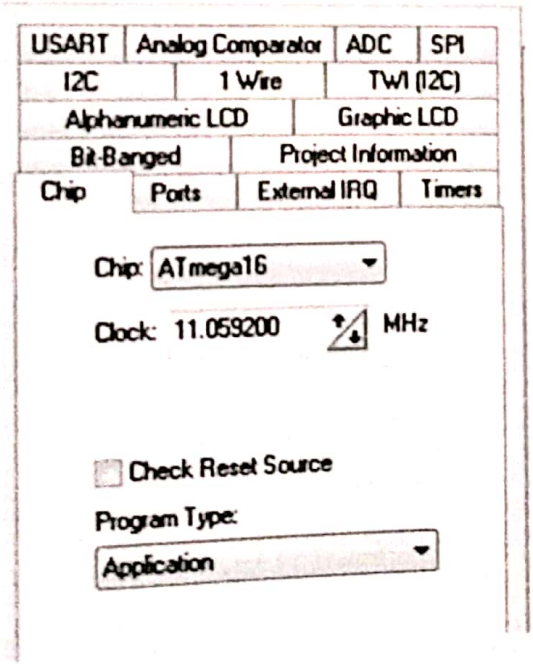
Gambar 4.9 Jendela Confirm CodeWizardAVR

Setelah itu akan muncul pilihan jenis chip yang akan di program menggunakan Code Wizard AVR, karena penulis menggunakan IC jenis AT MEGA 16 maka pilih tipe AT MEGA, seperti gambar 4.10 Pilihan tipe chip berikut:



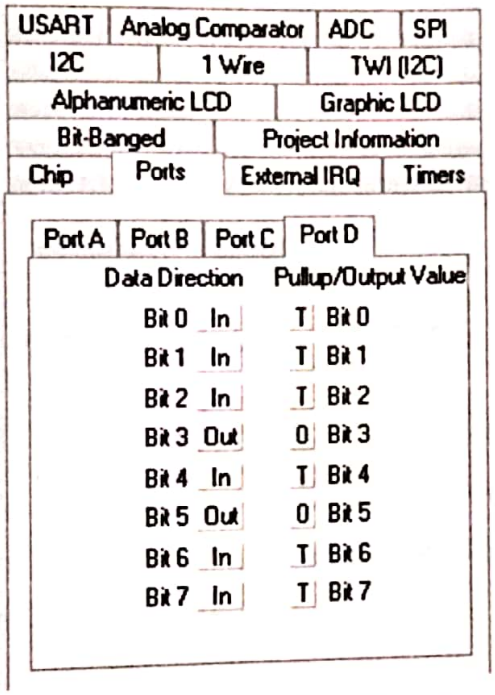
Gambar 4.10 Pilihan tipe Chip

Setelah selesai membuat projek baru maka akan dilanjutkan dengan langkah penginisialisasian port-port yang ada pada IC yang telah kita daftarkan pada table 4.1 tabel inisialisasi port, langkah pertama klik tab Chip dan isi IC sesuai yang kita pakai yaitu IC AT MEGA 16, seperti gambar 4.11 jenis IC berikut:

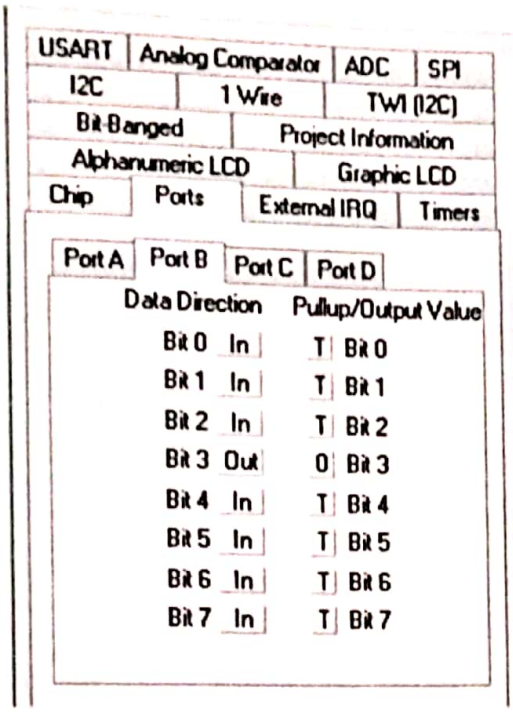


Gambar 4.11 Jenis IC AT MEGA 16

kemudian pilih tab port dimana port A sampai port D harus di inisialisasikan semua pin-pin yang terdapat pada IC AT MEGA 16 yang akan menjadi menjadi input dan output dari perintah yang ada di pin IC, adapun port yang di inisialisasikan sebagai output yaitu hanya Port D Bit 3 sebagai keluaran sensor ultrasonic depan dan Bit 5 sebagai keluaran sensor ultrasonic belakang serta Port B Bit 3 sebagai keluaran motor servo,selain dari itu di inisialisasikan sebagai input, lihat pengaturan port gambar 4.12 dan 4.13 berikut:

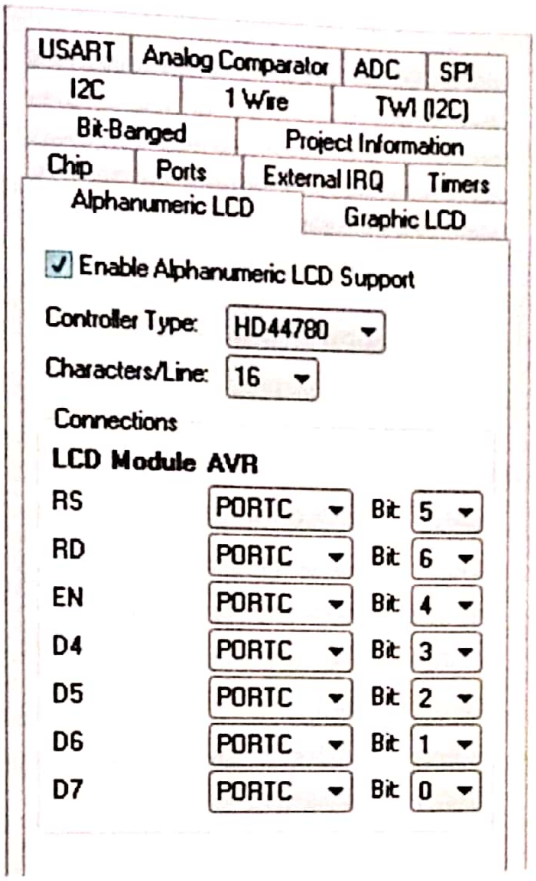


Gambar 4.12 Inisialisasi Port D



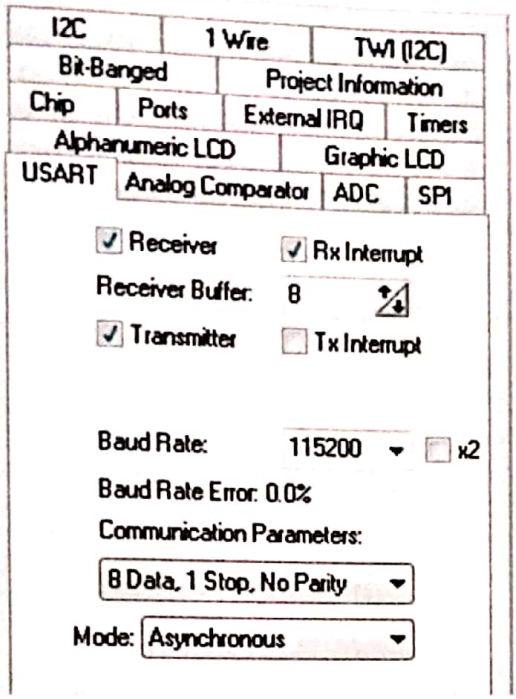
Gambar 4.12 Inisialisasi Port B

Selanjutnya pilih tab Alphenumeric LCD lalu klik enable Alphanumeric LCD support atur port yang akan di pakai pada IC AT MEGA16 yaitu Port C yang sesuai dengan fungsi LCD, adapun pengturannya dapat dilihat pada gambar 4.13 Inisialisasi Alphanumeric LCD berikut:



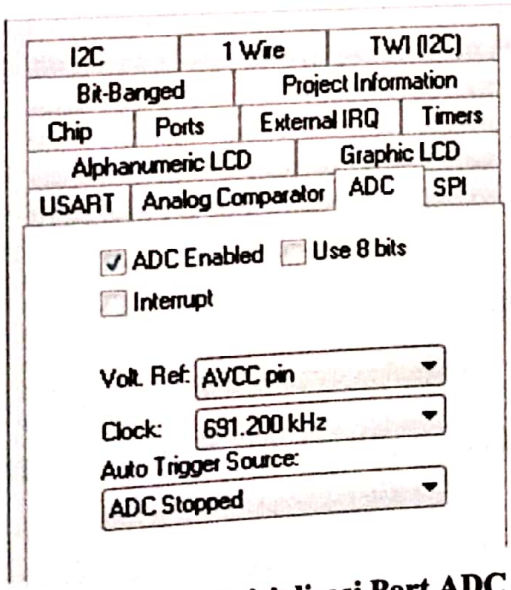
Gambar 4.13 Inisialisasi Alphanumeric LCD

Langkah selanjutnya pilih tab USART yang di alamatkan untuk penerimaan USB dari blok Arduino dengan USB Host Sield ke IC AT MEGA 16. Centang pada pilihan Receiver, Rx Interrupt dan Transmitter, adapun langkah ini dapat dilihat pada gambar 4.14 Inisialisasi Port USART sebagai berikut:



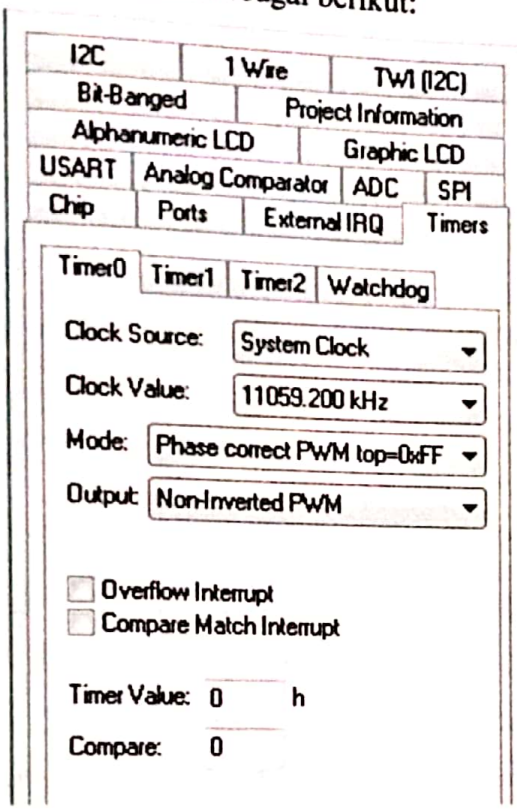
Gambar 4.14 Inisialisasi Port USART

Langkah selanjutnya yaitu pilih tab ADC yaitu salah satu port yang berfungsi sebagai pengkonversian Analog to Digital. Centang ADC enabled dan rubah Volt. Reff menjadi AVCC pin dan dapat dilihat pada gambar 4.14 Inisialisasi Port ADC:



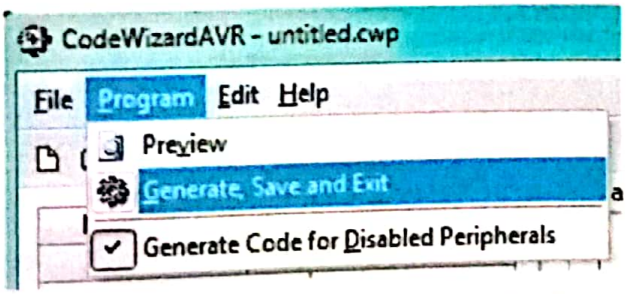
Gambar 4.14 Inisialisasi Port ADC

Adapun langkah terakhir dari penginisialisasian port yaitu pilih tab timers set timer0 pada Clock Value 11059.200 kHz, Mode Phase correct PWM top=0xFF dan pada Output Non-Inverted PWM. Langkah ini dapat dilihat pada gambar 4.15 Inisialisasi Port Timers sebagai berikut:



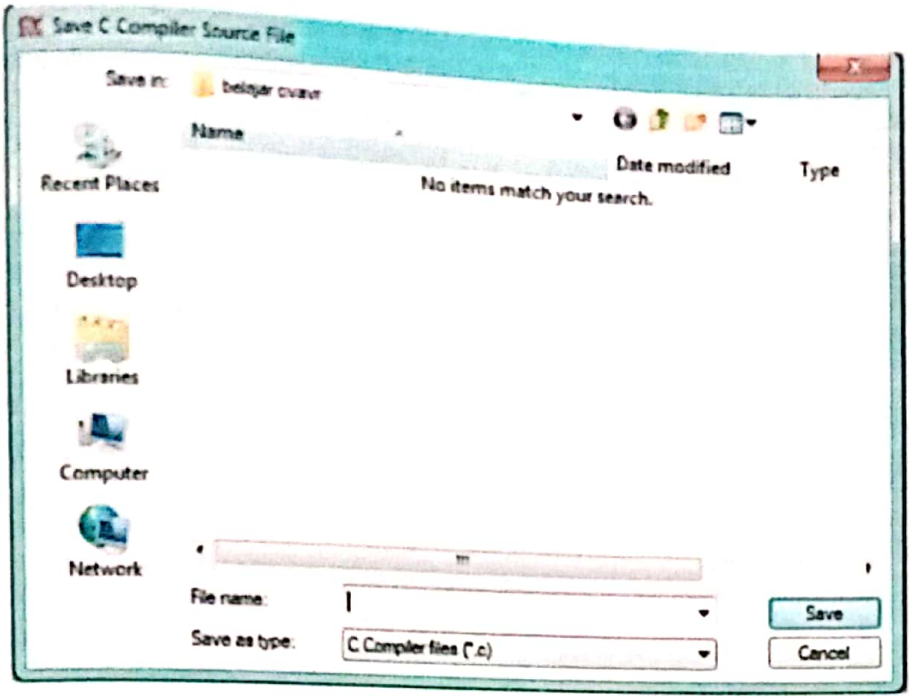
Gambar 4.15 Inisialisasi Port Timers

Langkah selanjutnya setelah selesai yaitu langkah generate, save and exit project yang kita buat, langkah ini dapat dilihat pada gambar 4.16 berikut:



Gambar 4.16 Generate, save and exit

Setelah generate, save and exit maka selanjutnya tahap penyimpanan tiga file project yang dapat dilihat pada gambar 4.17 save file berikut:



Gambar 4.17 Save Project

Inti dari langkah-langkah prosedur inisialisasi port diatas merupakan langkah awal terpenting dalam penyusunan program pintu otomatis ini. Dalam prosedur ini, setiap lajur port yang digunakan dialamatkan sesuai fungsi pengaturannya. Adapun bentuk dari potongan program inisialisasi port dapat dilihat di bawah ini:

```
/*****************************************************
This program was produced by the
CodeWizardAVR V2.05.3 Standard
Automatic Program Generator
© Copyright 1998-2011 Pavel Haiduc, HP InfoTech s.r.l.
http://www.hpinfotech.com

Project  : belajar
Version  :
Date     : 07-May-2015
Author   : firmansyah
Company  : 1111500075
Comments :


Chip type               : ATmega16
Program type            : Application
AVR Core Clock frequency: 11.059200 MHz
Memory model            : Small
```

```
External RAM size       : 0
Data Stack size         : 256
*****************************************************/

#include <mega16.h>

#include <delay.h>

// Alphanumeric LCD functions, fungsi dari alphanumeric lcd
#include <alcd.h>

#ifndef RXB8
#define RXB8 1
#endif

#ifndef TXB8
#define TXB8 0
#endif

#ifndef UPE
#define UPE 2
#endif

#ifndef DOR
#define DOR 3
#endif

#ifndef FE
#define FE 4
#endif

#ifndef UDRE
#define UDRE 5
#endif

#ifndef RXC
#define RXC 7
#endif

#define FRAMING_ERROR (1<<FE)
#define PARITY_ERROR (1<<UPE)
#define DATA_OVERRUN (1<<DOR)
#define DATA_REGISTER_EMPTY (1<<UDRE)
#define RX_COMPLETE (1<<RXC)

// USART Receiver buffer
#define RX_BUFFER_SIZE 8
char rx_buffer[RX_BUFFER_SIZE];

#if RX_BUFFER_SIZE <= 256
```

```
unsigned char rx_wr_index,rx_rd_index,rx_counter;
#else
unsigned int rx_wr_index,rx_rd_index,rx_counter;
#endif

// This flag is set on USART Receiver buffer overflow
bit rx_buffer_overflow;

// USART Receiver interrupt service routine
interrupt [USART_RXC] void usart_rx_isr(void)
{
char status,data;
status=UCSRA;
data=UDR;
if ((status & (FRAMING_ERROR | PARITY_ERROR | DATA_OVERRUN))==0)
  {
  rx_buffer[rx_wr_index++]=data;
#if RX_BUFFER_SIZE == 256
  // special case for receiver buffer size=256
  if (++rx_counter == 0) rx_buffer_overflow=1;
#else
  if (rx_wr_index == RX_BUFFER_SIZE) rx_wr_index=0;
  if (++rx_counter == RX_BUFFER_SIZE)
    {
    rx_counter=0;
    rx_buffer_overflow=1;
    }
#endif
  }
}

#ifndef _DEBUG_TERMINAL_IO_
// Get a character from the USART Receiver buffer
#define _ALTERNATE_GETCHAR_
#pragma used+
char getchar(void)
{
char data;
while (rx_counter==0);
data=rx_buffer[rx_rd_index++];
#if RX_BUFFER_SIZE != 256
if (rx_rd_index == RX_BUFFER_SIZE) rx_rd_index=0;
#endif
#asm("cli")
--rx_counter;
#asm("sei")
return data;
}
#pragma used-
#endif
```

```
// Standard Input/Output functions, deklarasi  fungsi standart input dan output
#include <stdio.h>

#define ADC_VREF_TYPE 0x40

// Read the AD conversion result
unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input | (ADC_VREF_TYPE & 0xff);
// Delay needed for the stabilization of the ADC input voltage
delay_us(10);
// Start the AD conversion
ADCSRA|=0x40;
// Wait for the AD conversion to complete
while ((ADCSRA & 0x10)==0);
ADCSRA|=0x10;
return ADCW;
}

// Declare your global variables here, deklarasikan variable secara global disini
void main(void)
{
// Declare your local variables here, delarasikan variable local disini
//Ketik coding disini
// Input/Output Ports initialization, penginisialisasian port input/output
// Port A initialization, penginisialisasian port A
// Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
// State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
PORTA=0x00;
DDRA=0x00;


// Port B initialization, penginisialisasian port B (pin 3 pada Port B di inisialisasikan
sebagai output)
// Func7=In Func6=In Func5=In Func4=In Func3=Out Func2=In Func1=In Func0=In
// State7=T State6=T State5=T State4=T State3=0 State2=T State1=T State0=T
PORTB=0x00;
DDRB=0x08;


// Port C initialization, penginisialisasian port C
// Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
// State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
PORTC=0x00;
DDRC=0x00;
```

```
// Port D initialization, penginisialisasian port D ( pin 3 dan 5 pada Port D di inisialisasikan sebagai output )
// Func7=In Func6=In Func5=Out Func4=In Func3=Out Func2=In Func1=In Func0=In
// State7=T State6=T State5=0 State4=T State3=0 State2=T State1=T State0=T
PORTD=0x00;
DDRD=0x28;

// Timer/Counter 0 initialization, penginisialisasian port timer counter 0
// Clock source: System Clock
// Clock value: 11059.200 kHz
// Mode: Phase correct PWM top=0xFF
// OC0 output: Non-Inverted PWM
TCCR0=0x61;
TCNT0=0x00;
OCR0=0x00;

// Timer/Counter 1 initialization, penginisialisasian port timer counter 1
// Clock source: System Clock
// Clock value: Timer1 Stopped
// Mode: Normal top=0xFFFF
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
// Timer1 Overflow Interrupt: Off
// Input Capture Interrupt: Off
// Compare A Match Interrupt: Off
// Compare B Match Interrupt: Off
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
ICR1H=0x00;
ICR1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization, penginisialisasian port timer counter 2
// Clock source: System Clock
// Clock value: Timer2 Stopped
// Mode: Normal top=0xFF
// OC2 output: Disconnected
ASSR=0x00;
TCCR2=0x00;
TCNT2=0x00;
OCR2=0x00;

// External Interrupt(s) initialization
```

```
// INT0: Off
// INT1: Off
// INT2: Off
MCUCR=0x00;
MCUCSR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;

// USART initialization, penginisialisasian port USART
// Communication Parameters: 8 Data, 1 Stop, No Parity
// USART Receiver: On
// USART Transmitter: On
// USART Mode: Asynchronous
// USART Baud Rate: 115200
UCSRA=0x00;
UCSRB=0x98;
UCSRC=0x86;
UBRRH=0x00;
UBRRL=0x05;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;
SFIOR=0x00;

// ADC initialization, penginisialisasian port analog to digital convertion
// ADC Clock frequency: 691.200 kHz
// ADC Voltage Reference: AVCC pin
// ADC Auto Trigger Source: ADC Stopped
ADMUX=ADC_VREF_TYPE & 0xff;
ADCSRA=0x84;

// SPI initialization
// SPI disabled
SPCR=0x00;

// TWI initialization
// TWI disabled
TWCR=0x00;

// Alphanumeric LCD initialization, penginisialisasian port lcd
// Connections are specified in the
// Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu:
// RS - PORTC Bit 5
// RD - PORTC Bit 6
// EN - PORTC Bit 4
// D4 - PORTC Bit 3
// D5 - PORTC Bit 2
```

```
// D6 - PORTC Bit 1
// D7 - PORTC Bit 0
// Characters/line: 16
lcd_init(16);

// Global enable interrupts, tempat pengetikan instruksi secara global
#asm("sei")

while (1)
    {
    // Place your code here, ketik codinganmu disini

    }
}
```

CODINGAN PINTU OTOMATIS DENGAN BARCODE SCANNER BERBASIS MIKROKONTROLER ATMEGA 16

```
#include <mega16.h>
#include <delay.h>
#define ADC_VREF_TYPE 0x40

// Read the AD conversion result
unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input | (ADC_VREF_TYPE & 0xff);
// Delay needed for the stabilization of the ADC input voltage
delay_us(10);
// Start the AD conversion
ADCSRA|=0x40;
// Wait for the AD conversion to complete
while ((ADCSRA & 0x10)==0);
ADCSRA|=0x10;
return ADCW;
}


// Alphanumeric LCD functions
#include <alcd.h>

#ifndef RXB8
#define RXB8 1
#endif

#ifndef TXB8
```

```
#define TXB8 0
#endif

#ifndef UPE
#define UPE 2
#endif

#ifndef DOR
#define DOR 3
#endif

#ifndef FE
#define FE 4
#endif

#ifndef UDRE
#define UDRE 5
#endif

#ifndef RXC
#define RXC 7
#endif

#define FRAMING_ERROR (1<<FE)
#define PARITY_ERROR (1<<UPE)
#define DATA_OVERRUN (1<<DOR)
#define DATA_REGISTER_EMPTY (1<<UDRE)
#define RX_COMPLETE (1<<RXC)

// USART Receiver buffer
#define RX_BUFFER_SIZE 8
```

```
char rx_buffer[RX_BUFFER_SIZE];

#if RX_BUFFER_SIZE <= 256
unsigned char rx_wr_index,rx_rd_index,rx_counter;
#else
unsigned int rx_wr_index,rx_rd_index,rx_counter;
#endif

// This flag is set on USART Receiver buffer overflow
bit rx_buffer_overflow;

// USART Receiver interrupt service routine
interrupt [USART_RXC] void usart_rx_isr(void)
{
char status,data;
status=UCSRA;
data=UDR;
if ((status & (FRAMING_ERROR | PARITY_ERROR |
DATA_OVERRUN))==0)
  {
  rx_buffer[rx_wr_index++]=data;
#if RX_BUFFER_SIZE == 256
  // special case for receiver buffer size=256
  if (++rx_counter == 0) rx_buffer_overflow=1;
#else
  if (rx_wr_index == RX_BUFFER_SIZE) rx_wr_index=0;
  if (++rx_counter == RX_BUFFER_SIZE)
    {
    rx_counter=0;
    rx_buffer_overflow=1;
    }
```

```
#endif
  }
}

#ifndef _DEBUG_TERMINAL_IO_
// Get a character from the USART Receiver buffer
#define _ALTERNATE_GETCHAR_
#pragma used+
char getchar(void)
{
  char data;
  if (rx_counter==0);
  if (rx_counter!=0)
  {
    data=rx_buffer[rx_rd_index++];
    #if RX_BUFFER_SIZE != 256
    if (rx_rd_index == RX_BUFFER_SIZE) rx_rd_index=0;
    #endif
    #asm("cli")
    --rx_counter;
    #asm("sei")
    return data;
  }
}
#pragma used-
#endif
// Standard Input/Output functions
#include <stdio.h>

#define PB3 PINB.0
#define PB2 PINB.1
```

```
#define PB1 PINB.2

#define tutup OCR0=219;
#define buka OCR0=224;
#define stop OCR0=0;

#define TRIGGER1 PORTD.2
#define ECHO1 PIND.3
#define TRIGGER2 PORTD.5
#define ECHO2 PIND.4

////////////////////////////////////// INISIALISAI VARIABEL //////////////////////////////////////
char data[16],data1[16],data2[16],a,b;
eeprom char barcode1[10][50];          // lokasi penyimpanan database memory
char barcode2[10];
int x,y,d,e,f,g,step,code,code1[10],us1,us2;
int count1,count2,jarak1,jarak2,clr,link,open;
////////////////////////////////////////////////////////////////////////////////////////////////////

void ultra_sens1()     //mengaktifkan sensor 1, diluar ruangan
{
   count1=0;           //seting awal nilai count
   TRIGGER1=1;         //inisialisasi sensor
   delay_us(10);
   TRIGGER1=0;
   while (ECHO1==0){};
   while (ECHO1==1)
   {
    count1++;
   }
   jarak1=count1*0.034442;     //rumus mencari jarak
```

```
}


void sens1()          // membaca kondisi sensor 1, diluar ruangan
{
  if(jarak1<5)
  {
    us1=1;
    open=1;
  }
  else us1=0;
}


void ultra_sens2()    //mengaktifkan sensor 2, didalam ruangan
{
  count2=0;           //seting awal nilai count
  TRIGGER2=1;         //inisialisasi sensor
  delay_us(10);
  TRIGGER2=0;
  while (ECHO2==0){};
  while (ECHO2==1)
  {
   count2++;
  }
  jarak2=count2*0.034442;     //rumus mencari jarak
}


void sens2()          // membaca kondisi sensor 2, didalam ruangan
{
```

```
  if(jarak2<3)
  {
    us2=1;
    open=2;
  }
  else us2=0;
}

void scan1()          // mendeskripsi data yang barcode yang dikirim dari arduino
{
    a=getchar();
    delay_ms(20);
    if(a=='z')
    {
     barcode2[b]=getchar();
     b++;
    }
    else if(a=='x')
    {
     b=0;
     d=1;
    }

sprintf(data,"%c%c%c%c%c%c%c%c%c%c",barcode2[0],barcode2[1],barcode2[
2],barcode2[3],barcode2[4],barcode2[5],barcode2[6],barcode2[7],barcode2[8],bar
code2[9]);
    lcd_gotoxy(5,1);
    lcd_puts(data);
}
```

```
void scan()                 // meminta data barcode dari arduino
{
  lcd_gotoxy(0,1);
  lcd_putsf("Id :");
  putchar('x');
  code=0;
  scan1();
}


void compare()              // membandingkan hasil scan barcode dengan database
barcode yang tersimpan
{
  for(x=0;x<50;x++)
  {
    for(y=0;y<10;y++)
    {
      if(barcode2[y]==barcode1[y][x])
      {
        code1[y]=1;
      }
    }
  }

code=code1[0]+code1[1]+code1[2]+code1[3]+code1[4]+code1[5]+code1[6]+cod
e1[7]+code1[8]+code1[9];
}


void clear_all()        // menghapus semua data barcode yang tersimpan di memory
{
```

```
  for(x=0;x<50;x++)
  {
    for(y=0;y<10;y++)
    {
      barcode1[y][x]=' ';
    }
  }
  clr=1;
}


void code_clear()          // menghapus history barcode Scaner
{
  for(e=0;e<10;e++)
  {
    barcode2[e]=' ';
    code1[e]=0;
  }
}


void door_control()        // kontrol buka tutup pintu
{
  while(link==1)
  {
    buka;
    if(read_adc(0)>150)
    {
      link=2;
```

```
    }

}
while(link==2)
{
   buka;
   if(read_adc(0)<150)
   {
      link=3;
   }
}
while(link==3)
{
   stop;
   if(open==1)
   {
      ultra_sens2();
      sens2();
      delay_ms(100);
      if(us2==1)
      {
         while(us2==1)
         {
            ultra_sens2();
            sens2();
            delay_ms(100);
         }
         link=4;
         open=0;
         us2=0;
      }
```

```
  }
  else if(open==2)
  {
     ultra_sens1();
     sens1();
     delay_ms(100);
     if(us1==1)
     {
        while(us1==1)
        {
           ultra_sens1();
           sens1();
           delay_ms(100);
        }
        link=4;
        open=0;
        us1=0;
     }
  }
}
while(link==4)
{
   tutup;
   if(read_adc(0)>150)
   {
      link=5;
   }
}
while(link==5)
{
   tutup;
```

```
    if(read_adc(0)<150)
    {
      stop;
      link=6;
    }
  }
  while(link==6)
  {
    stop;
    link=0;
    step=1;
    code_clear();
    lcd_clear();
  }

}

void main(void)
{
PORTA=0x00;
DDRA=0x00;
PORTB=0x00;
DDRB=0x08;
PORTC=0x00;
DDRC=0x00;
PORTD=0x00;
DDRD=0x24;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: 43.200 kHz
```

```
// Mode: Phase correct PWM top=0xFF
// OC0 output: Inverted PWM
TCCR0=0x74;
TCNT0=0x00;
OCR0=0x00;

// USART initialization
// Communication Parameters: 8 Data, 1 Stop, No Parity
// USART Receiver: On
// USART Transmitter: On
// USART Mode: Asynchronous
// USART Baud Rate: 115200
UCSRA=0x00;
UCSRB=0x98;
UCSRC=0x86;
UBRRH=0x00;
UBRRL=0x05;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;
SFIOR=0x00;

// ADC initialization
// ADC Clock frequency: 691.200 kHz
// ADC Voltage Reference: AVCC pin
// ADC Auto Trigger Source: ADC Stopped
ADMUX=ADC_VREF_TYPE & 0xff;
ADCSRA=0x84;
```

```
lcd_init(16);

// Global enable interrupts
#asm("sei")
step=0;
stop;
while (1)
    {

     while(step==0)
     {
       lcd_gotoxy(0,0);
       lcd_putsf("PB1 = Start");
       lcd_gotoxy(0,1);
       lcd_putsf("PB2 = Setup");
       if(PB1==0)              // masuk ke menu start
       {
          while(PB1==0){;}
          code_clear();
          step=1;
          lcd_clear();
       }
       if(PB2==0)              // masuk ke menu setup
       {
          while(PB2==0){lcd_clear();}
          lcd_gotoxy(0,0);
          lcd_putsf("Setup Mode.....");
          delay_ms(500);
          step=2;
          lcd_clear();
       }
```

```
if(PB3==0)              // menu clear all data barcode
{
  while(PB3==0)
  {
    while(PB2==0)
    {
      while(PB1==0)
      {
        clear_all();
        lcd_clear();
      }
    }
    if(clr==1)
    {
      lcd_gotoxy(0,0);
      lcd_putsf("All Data Cleared");
      delay_ms(500);
      lcd_clear();
      clr=0;
    }
  }
}

}
while(step==1)              // menu awal
{
  lcd_gotoxy(0,0);
  lcd_putsf(" Selamat Datang ");
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
if(PB3==0)      // tombol 3 untuk kembali ke menu awal
{               // perhatian, tombol hanya bisa ditekan saat kondisi sensor
                  belum mendeteksi benda kerja
  while(PB3==0){;}
  code_clear();
  step=0;
  lcd_clear();
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
ultra_sens1();        // menunggu sensor deteksi benda
ultra_sens2();        // menunggu sensor deteksi benda
sens1();              // menunggu sensor deteksi benda
sens2();              // menunggu sensor deteksi benda
delay_ms(100);
while(us1==1 || us2==1) // ketika sensor deteksi ada benda di sekitar pintu
{
  lcd_gotoxy(0,0);
  lcd_putsf(" Scan  your  ID ");
  scan();             // minta data barcode dari scanner
  if(d==1)
  {
    compare();      // membandingkan barcode yang di scan dengan
barcode yang tersimpan
      if(code==10)  // apabila barcode sama masuk ke menu buka pintu
      {
        lcd_gotoxy(0,0);
        lcd_putsf("ID Code Accepted");
        d=0;
        delay_ms(800);
        lcd_clear();
```

```
            us1=0;
            us2=0;
            link=1;
            step=5;
          }
          else      // apabila barcode berbeda maka akan menampilkan error dan
                       kembali ke menu awal
          {
            lcd_gotoxy(0,0);
            lcd_putsf("ID Code Error !!");
            d=0;
            delay_ms(800);
            lcd_clear();
            code_clear();
            us1=0;
            us2=0;
            step=1;
          }

        }
      }
    }
    while(step==2)                    // mode setup
    {
      lcd_gotoxy(0,0);
      lcd_putsf("PB1=Input  PB3<<");
      lcd_gotoxy(0,1);
      lcd_putsf("PB2=Hapus");
      if(PB1==0)                      // masuk ke mode input data barcode
      {
        while(PB1==0){lcd_clear();}
```

```
    lcd_gotoxy(0,0);
    lcd_putsf("Input Mode.....");
    delay_ms(300);
    step=3;
    g=0;
    f=0;
    lcd_clear();
  }
  if(PB2==0)                    // masuk ke mode hapus data barcode
  {
     while(PB2==0){lcd_clear();}
     lcd_gotoxy(0,0);
     lcd_putsf("Hapus Mode.....");
     delay_ms(300);
     f=0;
     step=4;
     lcd_clear();
   }
   if(PB3==0)                   // masuk ke mode sebelumnya
   {
      while(PB3==0){;}
      step=0;
      lcd_clear();
    }
}
while(step==3)                  // mode input data barcode
{
   while(g==0)
    {
       lcd_gotoxy(0,0);
       lcd_putsf("P1=OK  P2<< P3>>");
```

```
sprintf(data1,"%d = ",f);
lcd_gotoxy(0,1);
lcd_puts(data1);
for(e=0;e<10;e++)
{
   sprintf(data2,"%c",barcode1[e][f]); // memasukan data barcode ke
                                          variabel "data2"
   lcd_gotoxy(e+4,1);                  // menampilkan data barcode yang
                                          ada di memory
   lcd_puts(data2);
}
if(PB1==0)              // select posisi memory yang akan diisi data
                        barcode dari 0-49
{
   while(PB1==0){;}
   g=1;
   lcd_clear();
   code_clear();
}
if(PB2==0)              // geser kursor ke arah -
{
   while(PB2==0){;}
   f-=1;
   if(f<0)
   {
      f=0;
   }
   lcd_clear();
}
if(PB3==0)              // geser kursor ke arah +
{
```

```
        while(PB3==0){;}
        f+=1;
        if(f>50)
        {
          f=50;
        }
        lcd_clear();
      }
    }
    while(g==1)        // setelah lokasi memory dipilih, maka masuk mode scan
    {
      scan();          // scan mode, meminta data barcode dari scaner
      while(d==1)
      {
        lcd_gotoxy(0,0);
        lcd_putsf("PB1=Save PB2=Not");
        if(PB1==0)     // menu untuk menyimpan data barcode ke memory
        {
          while(PB1==0){lcd_clear();}
          lcd_gotoxy(0,0);
          lcd_putsf("Saving...");
          for(e=0;e<10;e++)   // proses memasukan data barcode ke memory
                              database  dengan lokasi memory (f) 0-49
          {
            barcode1[e][f]=barcode2[e];
            delay_ms(10);
          }
          delay_ms(500);
          step=2;
          lcd_clear();
          delay_ms(100);
```

```
            lcd_gotoxy(0,0);
            lcd_putsf("Save OK!!!");
            delay_ms(200);
            f+=1;
            d=0;
            g=3;
            lcd_clear();

          }
          if(PB2==0)   // menu untuk tidak menyimpan barcode, akan kembali
                          ke menu sebelumnya
          {
            while(PB2==0){;}
            code_clear();
            step=3;
            g=0;
            d=0;
            lcd_clear();
          }
        }
      }

}
while(step==4)          // menu hapus data
{
   lcd_gotoxy(0,0);
   lcd_putsf("P1=Clr P2<< P3>>");
   sprintf(data1,"%d = ",f);
   lcd_gotoxy(0,1);
   lcd_puts(data1);
   for(e=0;e<10;e++)
```

```
{
  sprintf(data2,"%c",barcode1[e][f]);
  lcd_gotoxy(e+4,1);
  lcd_puts(data2);

}
if(PB1==0)              // menu hapus data
{
  while(PB1==0){;}
  for(e=0;e<10;e++)
  {
    barcode1[e][f]=' ';  // menghapus data memory sesuai alamat yang
                            dipilih
  }
  lcd_clear();
  delay_ms(100);
  lcd_gotoxy(0,0);
  lcd_putsf("Data Cleared !!!");
  delay_ms(500);
  lcd_clear();
  step=2;
}
if(PB2==0)                  // memindahkan kursor -
{
  while(PB2==0){;}
  f-=1;
  if(f<0)
  {
    f=0;
  }
  lcd_clear();
```

```
        }
        if(PB3==0)                    // memindahkan kursor +
        {
          while(PB3==0){;}
          f+=1;
          if(f>50)
          {
            f=50;
          }
          lcd_clear();
        }
      }
      while(step==5)          // menu buka pintu
      {
        lcd_gotoxy(0,0);
        lcd_putsf("Door Opened ...");
        door_control();      // kirim data ke aktuator
      }
      lcd_clear();
    }
}
```

KOMPONEN

Nama	Jumlah
ATMEGA 16	1
ATMEGA 8	1
Resistor 560	3
Resistor 4K7	4
Resistor 2K2	1
Resistor 68	2
Dioda Zener 3,6V	2
ElCo 47uF	1
Capasitor 22pF	4
Capasitor 100nF	4
Crystal Oscillator 11,0592 Mhz	1
Crystal Oscillator 12 Mhz	1
IC Regulator 7805	1
Induktor 100 mH	1
LED 3mm Merah	1
LED 3mm Hijau	2
Tact Switch (PB)	3