ACKNOWLEDGMENTS

The Project be inspired for future idea. While I have the honor of having my name attached to this work, there are many others who have helped this idea become a reality.

I would first like to thank my adviser, Assoc. Prof. Dr. Doğan İbrahim, This interesting subject interested by him.Thanks to him.

I would like to thank my father,my mother,my sisters,my brother and I am happy from theirs' be one of piece.

I would like to thank all my friends, specially my best friends Mehmet Sadık Türüt, Fatih Van and Metehan Günde.

Finally, I would like to thank God, for this opportunity give to me.

ABSTRACT

This project give to answer what is the micro controllers? How is working the multitasking?

Microcontrollers had their beginnings in the development of technology of integrated circuits. We use but how does it works?
This development has made it possible to store hundreds of thousands of transistors into one chip. That was a prerequisite for production of microprocessors , and the first computers were made by adding external peripherals such as memory, input-output lines, timers and other.

Microcontroller differs from a microprocessor in many ways. First and the most important is its functionality. In order for a microprocessor to be used, other components such as memory, or components for receiving and sending data must be added to it.

The microprocessors used in the central processing units of computers are the bestknown types of microprocessors. But there are other kinds of microprocessors as well, most notably microcontrollers.

PAGES

CHAPTER ONE
INTRODUCTION TO MICROCONTROLLERS

1.1 Introduction

Circumstances that we find ourselves in today in the field of microcontrollers had their beginnings in the development of technology of integrated circuits. This development has made it possible to store hundreds of thousands of transistors into one chip. That was a prerequisite for production of microprocessors , and the first computers were made by adding external peripherals such as memory, input-output lines, timers and other. Further increasing of the volume of the package resulted in creation of integrated circuits. These integrated circuits contained both processor and peripherals. That is how the first chip containing a microcomputer, or what would later be known as a microcontroller came about.
It was year 1969, and a team of Japanese engineers from the BUSICOM company arrived to United States with a request that a few integrated circuits for calculators be made using their projects. The proposition was set to INTEL, and Marcian Hoff was responsible for the project. Since he was the one who has had experience in working with a computer (PC) PDP8, it occured to him to suggest a fundamentally different solution instead of the suggested construction. This solution presumed that the function of the integrated circuit is determined by a program stored in it. That meant that configuration would be more simple, but that it would require far more memory than the project that was proposed by Japanese engineers would require. After a while, though Japanese engineers tried finding an easier solution, Marcian's idea won, and the first microprocessor was born. In transforming an idea into a ready made product , Frederico Faggin was a major help to INTEL. He transferred to INTEL, and in only 9 months had succeeded in making a product from its first conception. INTEL obtained the rights to sell this integral block in 1971. First, they bought the license from the BUSICOM company who had no idea what treasure they had. During that year, there appeared on the market a microprocessor called 4004. That was the first 4-bit microprocessor with the speed of 6 000 operations per second. Not long after that, American company CTC requested from INTEL and Texas Instruments to make an 8-bit microprocessor for use in terminals. Even though CTC gave up this idea in the end, Intel and Texas Instruments kept working on the microprocessor and in April of 1972, first 8-bit microprocessor appeard on the market under a name 8008. It was able to address 16Kb of memory, and it had 45 instructions and the speed of 300 000 operations per second.

That microprocessor was the predecessor of all today's microprocessors. Intel kept their developments up in April of 1974, and they put on the market the 8-bit processor under a name 8080 which was able to address 64Kb of memory, and which had 75 instructions.

In another American company Motorola, they realized quickly what was happening, so they put out on the market an 8-bit microprocessor 6800. Chief constructor was Chuck Peddle, and along with the processor itself, Motorola was the first company to make other peripherals such as 6820 and 6850.
At that time many companies recognized greater importance of microprocessors and began their own developments.Chuck Peddle leaved Motorola to join MOS Technology and kept working intensively on developing microprocessors.
At the WESCON exhibit in United States in 1975, a critical event took place in the

history of microprocessors. The MOS Technology announced it was marketing microprocessors 6501 and 6502 at $25 each, which buyers could purchase immediately. This was so sensational that many thought it was some kind of a scam, considering that competitors were selling 8080 and 6800 at $179 each. As an answer to its competitor, both Intel and Motorola lowered their prices on the first day of the exhibit down to $69.95 per microprocessor. Motorola quickly brought suit against MOS Technology and Chuck Peddle for copying the protected 6800. MOS Technology stopped making 6501, but kept producing 6502. The 6502 was a 8-bit microprocessor with 56 instructions and a capability of directly addressing 64Kb of memory. Due to low cost , 6502 becomes very popular, so it was installed into computers such as: KIM-1, Apple I, Apple II, Atari, Comodore, Acorn, Oric, Galeb, Orao, Ultra, and many others.

Soon appeared several makers of 6502 (Rockwell, Sznertek, GTE, NCR, Ricoh, and Comodore takes over MOS Technology) which was at the time of its prosperity sold at a rate of 15 million processors a year!

Others were not giving up though. Frederico Faggin leaves Intel, and starts his own Zilog Inc.In 1976 Zilog announced the Z80. During the making of this microprocessor, Faggin made a pivotal decision. Knowing that a great deal of programs have been already developed for 8080, Faggin realized that many would stay faithful to that microprocessor because of great expenditure which redoing of all of the programs would result in. Thus he decided that a new processor had to be compatible with 8080, or that it had to be capable of performing all of the programs which had already been written for 8080. Beside these characteristics, many new ones have been added, so that Z80 was a very powerful microprocessor in its time. It was able to address directly 64 Kb of memory, it had 176 instructions, a large number of registers, a built in option for refreshing the dynamic RAM memory, single-supply, greater speed of work etc. Z80 was a great success and everybody converted from 8080 to Z80. It could be said that Z80 was without a doubt commercially most successful 8-bit microprocessor of that time. Besides Zilog, other new manufacturers like Mostek, NEC, SHARP, and SGS also appeared. Z80 was the heart of many computers like Spectrum, Partner, TRS703, Z-3 .

In 1976, Intel came up with an improved version of 8-bit microprocessor named 8085. However, Z80 was so much better that Intel soon lost the battle. Altough a few more processors appeared on the market (6809, 2650, SC/MP etc.), everything was actually already decided. There weren't any more great improvements to make manufacturers convert to something new, so 6502 and Z80 along with 6800 remained as main representatives of the 8-bit microprocessors of that time.

By 1969, it was generally recognised in electronics industry that it was theoretically possible to use the new metal-on-silicon (MOS) semiconductor manufacturing technology to put all of the function of a calculator on a single chip.

Only in retrospect it is the distance form theory to practice a tine gap. At the time, when you are risking an entire company and its employees, that gap is a frightening chasm. In the world of MOS, the risk was even greater because the technology was so new that it was almost impossible to determine who the industry leaders would be. Certainly, the product choice would have been one of the giant semiconductor corporations, such as Fairchild or Motorola… not a tiny, new start-up in Santa Clara, California named Intel Corp.

Busicom that was a young and aggressive Japanese company decided to take that leap of faith. It wanted to build the first calculator on chips. The decision changed the world. Its timing was perfect. Just as the idea of integrating the components of a calculator on one chip was capturing the fancy of the computation industry, a comparable vision was sweeping the semiconductor business.

1.2 Microcontrollers versus Microprocessors

Microcontroller differs from a microprocessor in many ways. First and the most important is its functionality. In order for a microprocessor to be used, other components such as memory, or components for receiving and sending data must be added to it. In short that means that microprocessor is the very heart of the computer. On the other hand, microcontroller is designed to be all of that in one. No other external components are needed for its application because all necessary peripherals are already built into it. Thus, we save the time and space needed to construct devices.

1.3 The First Microprocessor Family – Intel 4000s

In 1969, some trade magazines and professional conferences had already kicked off a live debate that focused upon the hot new product of the era, the calculator. One side argued that the best way to harness the power of semiconductor technology was to create custom circuits specifically for each calculator model.

A second, lessinfluential, comp held that, no, the best answer was to imitate at the chip level thearchitecture of computers – that is, general purpose chips that would then be programmed for the specific application. In retrospect, it is obvious that the latter position was better, if for nothing else because it opened the prospect of a long-term development strategy that would extend this technology to other applications beyond calculator and watches.

There had even been a few attempts to build such a chip. In Fairchild, there was a brilliant semiconductor scientist named Federico Faggin had invented a new kind of MOS process called silicon gate technology that would supplant bipolar technology as the dominant semiconductor process for advanced circuits. Intel quickly adopted silicon gate MOS and perfected it, a skill that would play a crucial role in the company's success.

But we must remember: at the time there were no other applications for these chips beyond calculators. Busicom thought that building general-purpose chips for a specific application would not be cost-effective, so it put out for contract on its new calculator was for ten custom circuits. Meanwhile, Intel was working on the Busicom chip-set bid, ignored the Busicom specifications and set out to win the contract by creating new general-purpose calculator chip architecture.

In October 1969, Intel defined a new four chip calculator architecture that include a 4-bit logic chip (CPU), read only memory (ROM) to store program instructions, random access memory (RAM) to hold the raw data and the processed results, and a shift

register to provide connects (ports) to a keyboard, printer, switches and light emitting diode (LED) displays. But Intel didn't really know how to translate this architecture into a working chip design.
In fact, probably only one person in the world did know how to do the next step. That was Federico Faggin, but he was at Fairchild. In April 1970, Faggin jointed the younger firm and immediately design the Busicom chip set. Within three months, Faggin had the design for the four chip set in hand. It was to be called the 4000 Family and it consisted of the 4001, a 2,048-bit ROM memory; the 4002, a 320-bit RAM memory; the 4003, a 10-bit input-output shift register; and, most memorably, the 4004, a 4-bit central processor logic chip. In mid-March 1971, Intel shipped the first 4000 Family chip sets to Busicom. The microprocessor revolution had begun.
In April 1972, the 8008 (8-bit microprocessor) introduced and met with enthusiastic response and few sales. After two years, Faggin improved his design and finished a new product – Intel 8080. With the introduction of the 8080, it can truly be said that mankind changed. Unlike many landmark inventions, the extraordinary nature of the 8080 was recognised almost instantly by thousands of engineers throughout the world who would been awaiting its arrival. Within a year, it had been designed into hundreds of different products. Nothing would be the same again.

1.4 First Microcontroller

The microprocessors used in the central processing units of computers are the bestknown types of microprocessors. But there are other kinds of microprocessors as well, most notably microcontrollers, which provide digital intelligence for everything from appliances to engine computers, and which act as the 'engines' for computer peripherals.

Microcontrollers, beyond the features they share with their central processor counterparts, also add another important function: digital signal processing (DSP). DSP can be seen as the way that the microcontroller can deal directly with the messy and unpredictable natural world. Analogy signals arriving form the outside often come in a jumble, distorted and with pieces missing. DSP sorts through this, picking out what matters using a process called analogy/digital conversion.

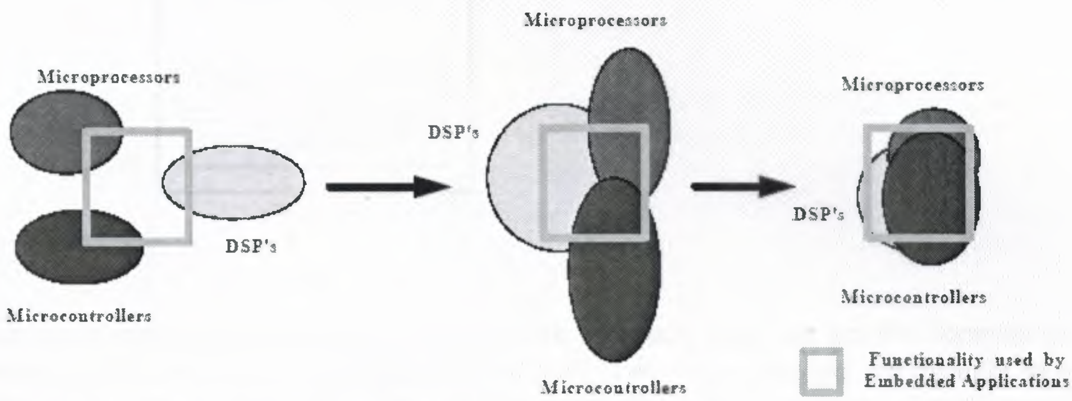
Microcontrollers were designed to fulfil a growing market need for the management of real-time, on-going physical events rather than the number crunching of data processing where microprocessors had found their home. For example, whereas a microprocessor might power the engine computer in an automobile, microcontroller

1.5 The Development of Microcontrollers

Dataquest report that from 1993 to 1995, the global demand for 16-bit microcontroller grew by an impressive compound annual rate of 86 percent.Withthisdramatic expansion comes an increasing worldwide demand for higher performance microcontroller, driven by rapid advances in data processing and telecommunication technology. Traditionally, applications range from hard dish drivers to scanners, office copiers and fax machines to digital cameras, modems and feature phones have employed a microcontroller chip to monitor real-time events and a separated digital signal processor (DSP) chip for numerical processing and digital filtering. This traditional two-chip solution is not only relatively costly to implement, but also consumes valuable board space and can add a needless level of complexity to manufacturing and quality assurance.

1.6 Yesterday to Today

From above general introduction, you can see that each company all has its' own 32-bit microcontroller. Form the structure , almost 32-bit microcontroller use RISC technologies , 32 - bit general - purpose registers , and add DSP function in the microcontroller. The DSP is becoming more ubiquitous since the functionality of embedded systems now encompasses signal processing in one form or the other.

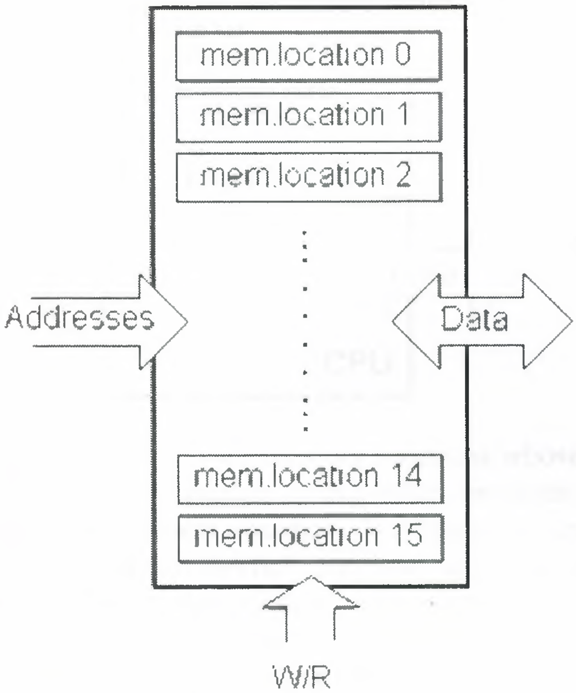


the trend of the convergence of architectures in microcontroller and DSP. For instance, multi-function peripherals, or printer-fax-scanner-copier devices, need DSP capability to perform the V.17 fax algorithm and also the image processing for scanning. From here you can see the trend of the convergence of architectures in the form of microcontroller and DSP.

But there are still some differences between these microcontrollers. In Chapter 3, I will mainly compare the difference of performance,and DSP function between some microcontrollers.

1.7 Memory Unit

Memory is part of the microcontroller whose function is to store data. The easiest way to explain it is to describe it as one big closet with lots of drawers. If we suppose that we marked the drawers in such a way that they can not be confused, any of their contents will then be easily accessible. It is enough to know the designation of the drawer and so its contents will be known to us for sure.

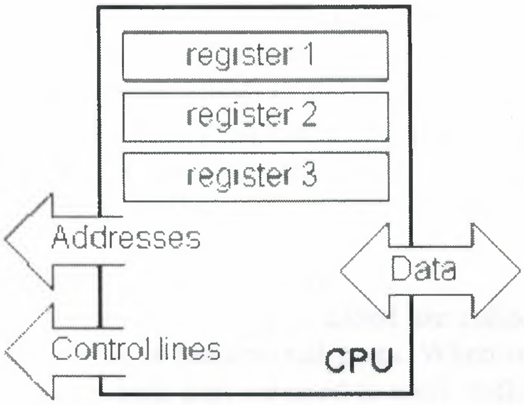


Example of simplified model of a memory unit. For a specific input we get a corresponding output. Line R/W determines wheather we are reading from or writing to memory

Memory components are exactly like that. For a certain input we get the contents of a certain addressed memory location and that's all. Two new concepts are brought to us: addressing and memory location. Memory consists of all memory locations, and addressing is nothing but selecting one of them. This means that we need to select the desired memory location on one hand, and on the other hand we need to wait for the contents of that location. Beside reading from a memory location, memory must also provide for writing onto it. This is done by supplying an additional line called control line. We will designate this line as R/W (read/write). Control line is used in the following way: if r/w=1, reading is done, and if opposite is true then writing is done on the memory location. Memory is the first element, and we need a few operation of our microcontroller .

1.8 Central Processing Unit

Let add 3 more memory locations to a specific block that will have a built in capability to multiply, divide, subtract, and move its contents from one memory location onto another. The part we just added in is called "central processing unit" (CPU). Its memory locations are called registers.

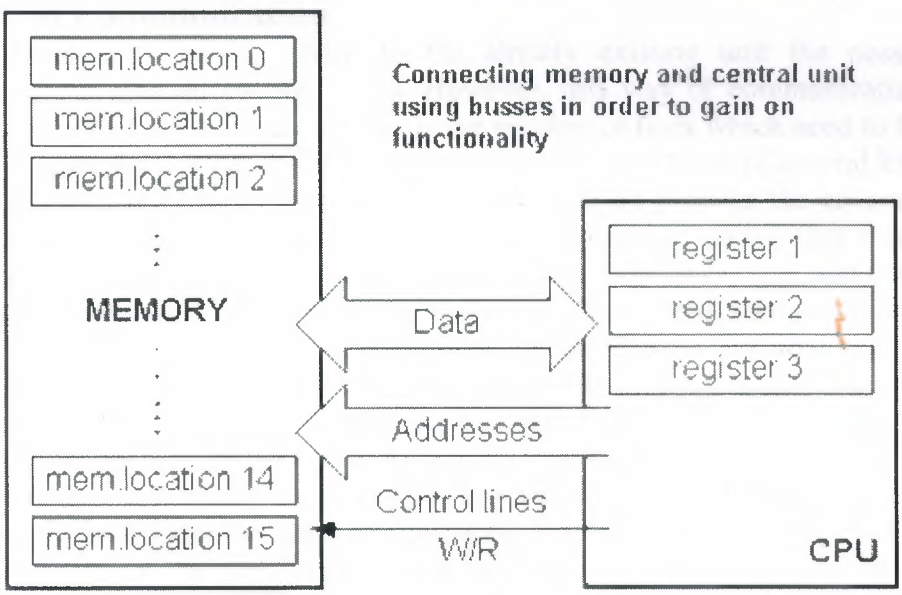


Example of simplified central processing unit with three registers

Registers are therefore memory locations whose role is to help with performing various mathematical operations or any other operations with data wherever data can be found. Look at the current situation. We have two independent entities (memory and CPU) which are interconnected, and thus any exchange of data is hindered, as well as its functionality. If, for example, we wish to add the contents of two memory locations and return the result again back to memory, we would need a connection between memory and CPU. Simply stated, we must have some "way" through data goes from one block to another.

1.9 Bus

That "way" is called "bus". Physically, it represents a group of 8, 16, or more wires There are two types of buses: address and data bus. The first one consists of as many lines as the amount of memory we wish to address, and the other one is as wide as data, in our case 8 bits or the connection line. First one serves to transmit address from CPU memory, and the second to connect all blocks inside the microcontroller.

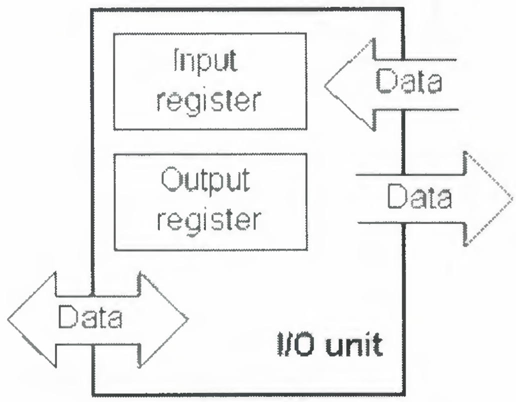


Connecting memory and central unit using busses in order to gain on functionality

As far as functionality, the situation has improved, but a new problem has also appeared: we have a unit that's capable of working by itself, but which does not have any contact with the outside world, or with us! In order to remove this deficiency, let's add a block which contains several memory locations whose one end is connected to the data bus, and the other has connection with the output lines on the microcontroller which can be seen as pins on the electronic component.

1.10 Input-Output unit

Those locations we've just added are called "ports". There are several types of ports : input, output or bidiectional ports. When working with ports, first of all it is necessary to choose which port we need to work with, and then to send data to, or take it from the port.



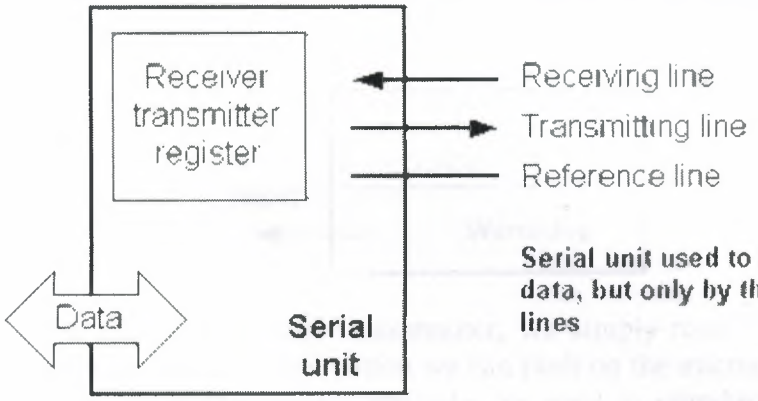
Example of a simplified input-output unit that provides communication with external world

When working with it the port acts like a memory location. Something is simply being written into or read from it, and it could be noticed on the pins of the microcontroller.

1.11 Serial Communication

Beside stated above we've added to the already existing unit the possibility of communication with an outside world. However, this way of communicating has its drawbacks. One of the basic drawbacks is the number of lines which need to be used in order to transfer data. What if it is being transferred to a distance of several kilometers? The number of lines times number of kilometers doesn't promise the economy of the project. It leaves us having to reduce the number of lines in such a way that we don't lessen its functionality. Suppose we are working with three lines only, and that one line is used for sending data, other for receiving, and the third one is used as a reference line for both the input and the output side. In order for this to work, we need to set the rules of exchange of data. These rules are called protocol. Protocol is therefore defined in advance so there wouldn't be any misunderstanding between the sides that are communicating with each other. For example, if one man is speaking in French, and the other in English, it is highly unlikely that they will quickly and effectively understand each other. Let's suppose we have the following protocol. The logical unit "1" is set up on the transmitting line until transfer begins. Once the transfer starts, we lower the transmission line to logical "0" for a period of time (which we will designate as T), so the receiving side will know that it is receiving data, and so it will activate its mechanism for reception. Let's go back now to the transmission side and start putting logic zeros and ones onto the transmitter line in the order from a bit of the lowest value to a bit of the highest value. Let each bit stay on line for a time period which is equal to

T, and in the end, or after the 8th bit, let us bring the logical unit "1" back on the line which will mark the end of the transmission of one data. The protocol we've just described is called in professional literature NRZ (Non-Return to Zero).

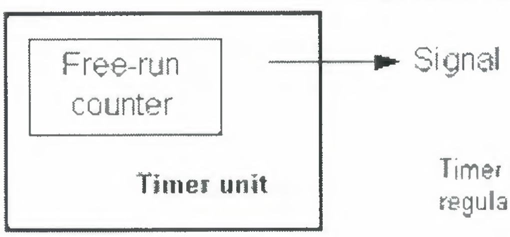


Serial unit used to send data, but only by three lines

As we have separate lines for receiving and sending, it is possible to receive and send data (info.) at the same time. So called full-duplex mode block which enables this way of communication is called a serial communication block. Unlike the parallel transmission, data moves here bit by bit, or in a series of bits what defines the term serial communication comes from. After the reception of data we need to read it from the receiving location and store it in memory as opposed to sending where the process is reversed. Data goes from memory through the bus to the sending location, and then to the receiving unit according to the protocol.

1.12 Timer Unit

Since we have the serial communication explained, we can receive, send and process data.

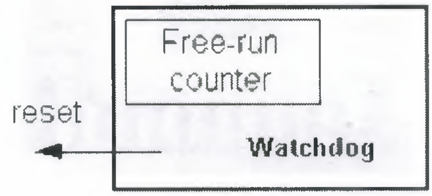


Timer unit generates signals in regular time intervals

However, in order to utilize it in industry we need a few additionally blocks. One of those is the timer block which is significant to us because it can give us information about time, duration, protocol etc. The basic unit of the timer is a free-run counter which is in fact a register whose numeric value increments by one in even intervals, so that by taking its value during periods T1 and T2 and on the basis of their difference we can determine how much time has elapsed. This is a very important part of the microcontroller whose understanding requires most of our time.

1.13 Watchdog

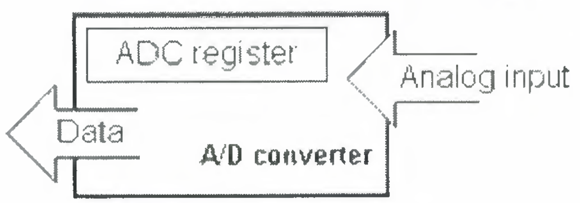
One more thing is requiring our attention is a flawless functioning of the microcontroller during its run-time. Suppose that as a result of some interference (which often does occur in industry) our microcontroller stops executing the program, or worse, it starts working incorrectly.



Of course, when this happens with a computer, we simply reset it and it will keep working. However, there is no reset button we can push on the microcontroller and thus solve our problem. To overcome this obstacle, we need to introduce one more block called watchdog. This block is in fact another free-run counter where our program needs to write a zero in every time it executes correctly. In case that program gets "stuck", zero will not be written in, and counter alone will reset the microcontroller upon achieving its maximum value. This will result in executing the program again, and correctly this time around. That is an important element of every program to be reliable without man's supervision.

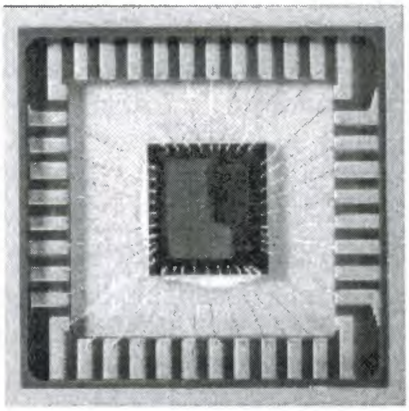
1.14 Analog to Digital Converter

As the peripheral signals usually are substantially different from the ones that microcontroller can understand (zero and one), they have to be converted into a pattern which can be comprehended by a microcontroller. This task is performed by a block for analog to digital conversion or by an ADC. This block is responsible for converting an information about some analog value to a binary number and for follow it through to a CPU block so that CPU block can further process it.



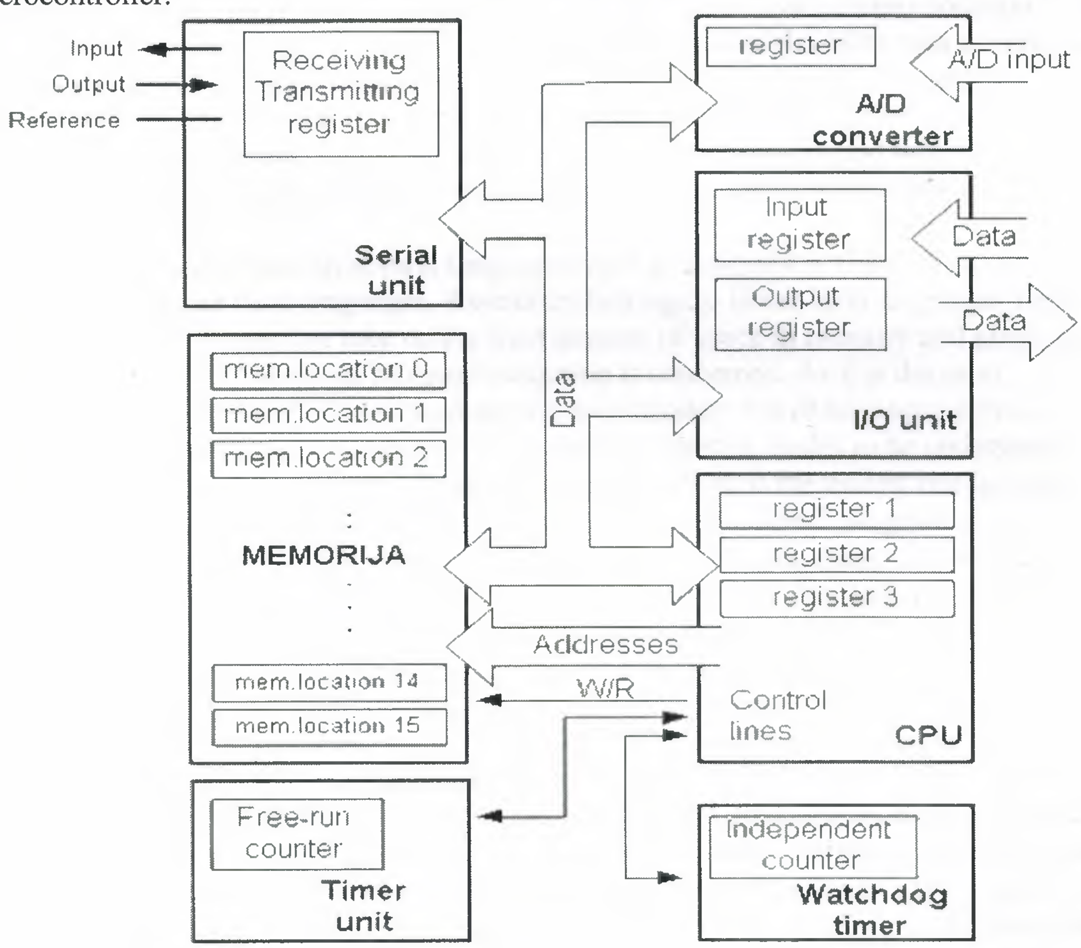
Block for converting an analogue to a digital form

Finnaly, the microcontroller is now completed, and all we need to do now is to assemble it into an electronic component where it will access inner blocks through the outside pins. The picture below shows what a microcontroller looks like inside.



Physical configuration of the interior of a microcontroller

Thin lines which lead from the center towards the sides of the microcontroller represent wires connecting inner blocks with the pins on the housing of the microcontroller so called bonding lines. Chart on the following page represents the center section of a microcontroller.



Microcontroller outline with its basic elements and internal connections

For a real application, a microcontroller alone is not enough. Beside a microcontroller, we need a program that would be executed, and a few more elements which make up a interface logic towards the elements of regulation (which will be discussed in later chapters).

1.15 Program

Program writing is a special field of work with microcontrollers and is called "programming". Try to write a small program in a language that we will make up ourselves first and then would be understood by anyone.

START
REGISTER1=MEMORY LOCATION_A
REGISTER2=MEMORY LOCATION_B
PORTA=REGISTER1 + REGISTER2

END

The program adds the contents of two memory locations, and views their sum on port A. The first line of the program stands for moving the contents of memory location "A" into one of the registers of central processing unit. As we need the other data as well, we will also move it into the other register of the central processing unit. The next instruction instructs the central processing unit to add the contents of those two registers and send a result to port A, so that sum of that addition would be visible to the outside world. For a more complex problem, program that works on its solution will be bigger.

Programming can be done in several languages such as Assembler, C and Basic which are most commonly used languages. Assembler belongs to lower level languages that are programmed slowly, but take up the least amount of space in memory and gives the best results where the speed of program execution is concerned. As it is the most commonly used language in programming microcontrollers it will be discussed in a later chapter. Programs in C language are easier to be written, easier to be understood, but are slower in executing from assembler programs. Basic is the easiest one to learn, and its instructions are nearest a man's way of reasoning, but like C programming language it is also slower than assembler. In any case, before you make up your mind about one of these languages you need to consider carefully the demands for execution speed, for the size of memory and for the amount of time available for its assembly.

After the program is written, we would install the microcontroller into a device and run it. In order to do this we need to add a few more external components necessary for its work. First we must give life to a microcontroller by connecting it to a power supply (power needed for operation of all electronic instruments) and oscillator whose role is similar to the role that heart plays in a human body. Based on its clocks microcontroller executes instructions of a program. As it receives supply microcontroller will perform a small check up on itself, look up the beginning of the program and start executing it. How the device will work depends on many parameters, the most important of which is the skillfulness of the developer of hardware, and on programmer's expertise in getting the maximum out of the device with his program.

CHAPTER TWO
MICROCONTROLLER PIC16F84

2.1 Introduction

PIC16F84 belongs to a class of 8-bit microcontrollers of RISC architecture. Its general structure is shown on the following map representing basic blocks.

Program memory (FLASH)- for storing a written program.
Since memory made in FLASH technology can be programmed and cleared more than once, it makes this microcontroller suitable for device development.

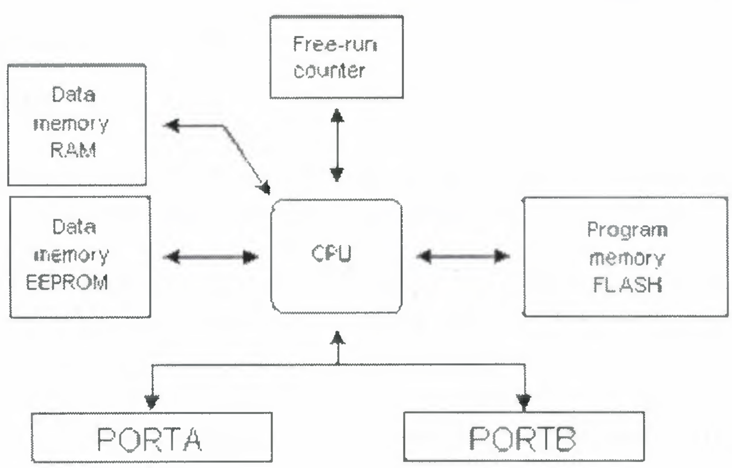
EEPROM - data memory that needs to be saved when there is no supply.
It is usually used for storing important data that must not be lost if power supply suddenly stops. For instance, one such data is an assigned temperature in temperature regulators. If during a loss of power supply this data was lost, we would have to make the adjustment once again upon return of supply. Thus our device looses on self-reliance.

RAM - data memory used by a program during its execution.
In RAM are stored all inter-results or temporary data during run-time.

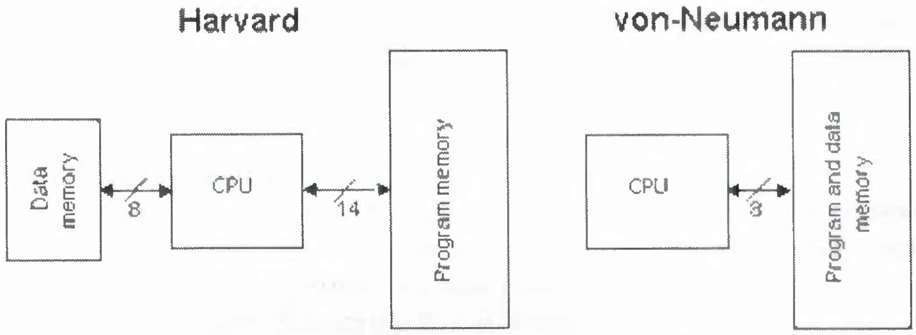
PORTA and PORTB are physical connections between the microcontroller and the outside world. Port A has five, and port B has eight pins.

FREE-RUN TIMER is an 8-bit register inside a microcontroller that works independently of the program. On every fourth clock of the oscillator it increments its value until it reaches the maximum (255), and then it starts counting over again from zero. As we know the exact timing between each two increments of the timer contents, timer can be used for measuring time which is very useful with some devices.

CENTRAL PROCESSING UNIT has a role of connective element between other blocks in the microcontroller. It coordinates the work of other blocks and executes the user program.



PIC16F84 microcontroller outline



Harvard vs. von Neuman Block Architectures

2.2 CISC, RISC

It has already been said that PIC16F84 has a RISC architecture. This term is often found in computer literature, and it needs to be explained here in more detail. Harvard architecture is a newer concept than von-Neumann's. It rose out of the need to speed up the work of a microcontroller. In Harvard architecture, data bus and address bus are separate. Thus a greater flow of data is possible through the central processing unit, and of course, a greater speed of work. Separating a program from data memory makes it further possible for instructions not to have to be 8-bit words. PIC16F84 uses 14 bits for instructions which allows for all instructions to be one word instructions. It is also typical for Harvard architecture to have fewer instructions than von-Neumann's, and to have instructions usually executed in one cycle.

Microcontrollers with Harvard architecture are also called "RISC microcontrollers". RISC stands for Reduced Instruction Set Computer. Microcontrollers with von-Neumann's architecture are called 'CISC microcontrollers'. Title CISC stands for Complex Instruction Set Computer.

Since PIC16F84 is a RISC microcontroller, that means that it has a reduced set of instructions, more precisely 35 instructions . (ex. Intel's and Motorola's microcontrollers have over hundred instructions) All of these instructions are executed in one cycle except for jump and branch instructions. According to what its maker says, PIC16F84 usually reaches results of 2:1 in code compression and 4:1 in speed in relation to other 8-bit microcontrollers in its class.

2.3 Applications

PIC16F84 perfectly fits many uses, from automotive industries and controlling home appliances to industrial instruments, remote sensors, electrical door locks and safety devices. It is also ideal for smart cards as well as for battery supplied devices because of its low consumption.

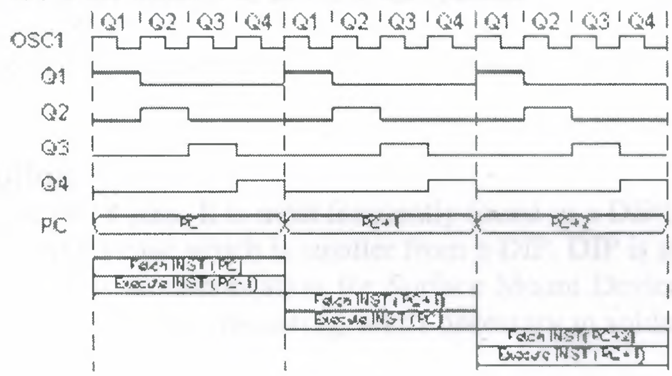
EEPROM memory makes it easier to apply microcontrollers to devices where permanent storage of various parameters is needed (codes for transmitters, motor speed, receiver frequencies, etc.). Low cost, low consumption, easy handling and flexibility make PIC16F84 applicable even in areas where microcontrollers had not previously been considered (example: timer functions, interface replacement in larger systems, coprocessor applications, etc.).

In System Programmability of this chip (along with using only two pins in data transfer) makes possible the flexibility of a product, after assembling and testing have been completed. This capability can be used to create assembly-line production, to store calibration data available only after final testing, or it can be used to improve programs on finished products.

2.4 Clock / Instruction Cycle

Clock is microcontroller's main starter, and is obtained from an external component called an "oscillator". If we want to compare a microcontroller with a time clock, our "clock" would then be a ticking sound we hear from the time clock. In that case, oscillator could be compared to a spring that is wound so time clock can run. Also, force used to wind the time clock can be compared to an electrical supply.

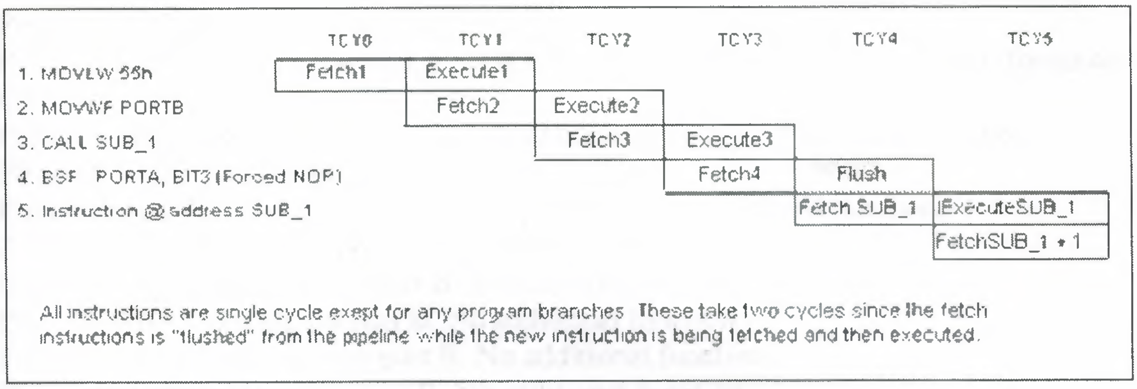
Clock from the oscillator enters a microcontroller via OSC1 pin where internal circuit of a microcontroller divides the clock into four even clocks Q1, Q2, Q3, and Q4 which do not overlap. These four clocks make up one instruction cycle (also called machine cycle) during which one instruction is executed.
Execution of instruction starts by calling an instruction that is next in string. Instruction is called from program memory on every Q1 and is written in instruction register on Q4. Decoding and execution of instruction are done between the next Q1 and Q4 cycles. On the following diagram we can see the relationship between instruction cycle and clock of the oscillator (OSC1) as well as that of internal clocks Q1-Q4. Program counter (PC) holds information about the address of the next instruction.



Clock/Insruction Cycle

2.5 Pipelining

Instruction cycle consists of cycles Q1, Q2, Q3 and Q4. Cycles of calling and executing instructions are connected in such a way that in order to make a call, one instruction cycle is needed, and one more is needed for decoding and execution. However, due to pipelining, each instruction is effectively executed in one cycle. If instruction causes a change on program counter, and PC doesn't point to the following but to some other address (which can be the case with jumps or with calling subprograms), two cycles are needed for executing an instruction. This is so because instruction must be processed again, but this time from the right address. Cycle of calling begins with Q1 clock, by writing into instruction register (IR). Decoding and executing begins with Q2, Q3 and Q4 clocks.

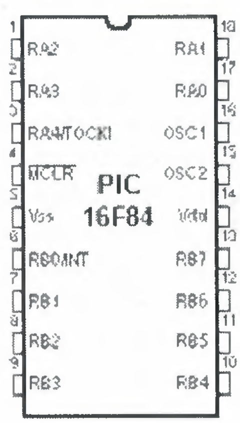


Instruction Pipeline Flow

TCY0 reads in instruction MOVLW 55h (it doesn't matter to us what instruction was executed, because there is no rectangle pictured on the bottom).
TCY1 executes instruction MOVLW 55h and reads in MOVWF PORTB.
TCY2 executes MOVWF PORTB and reads in CALL SUB_1.
TCY3 executes a call of a subprogram CALL SUB_1, and reads in instruction BSF PORTA, BIT3. As this instruction is not the one we need, or is not the first instruction of a subprogram SUB_1 whose execution is next in order, instruction must be read in again. This is a good example of an instruction needing more than one cycle.
TCY4 instruction cycle is totally used up for reading in the first instruction from a subprogram at address SUB_1.
TCY5 executes the first instruction from a subprogram SUB_1 and reads in the next one.

2.6 Pin Description

PIC16F84 has a total of 18 pins. It is most frequently found in a DIP18 type of case but can also be found in SMD case which is smaller from a DIP. DIP is an abbreviation for Dual In Package. SMD is an abbreviation for Surface Mount Devices suggesting that holes for pins to go through when mounting, aren't necessary in soldering this type of a component.



Pins on PIC16F84 microcontroller have the following meaning:

Pin no.1 RA2 Second pin on port A. Has no additional function

Pin no.2 RA3 Third pin on port A. Has no additional function.
Pin no.3 RA4 Fourth pin on port A. TOCK1 which functions as a timer is also found on this pin
Pin no.4 MCLR Reset input and Vpp programming voltage of a microcontroller
Pin no.5 Vss Ground of power supply.
Pin no.6 RB0 Zero pin on port B. Interrupt input is an additional function.
Pin no.7 RB1 First pin on port B. No additional function.
Pin no.8 RB2 Second pin on port B. No additional function.
Pin no.9 RB3 Third pin on port B. No additional function.
Pin no.10 RB4 Fourth pin on port B. No additional function.
Pin no.11 RB5 Fifth pin on port B. No additional function.
Pin no.12 RB6 Sixth pin on port B. 'Clock' line in program mode.
Pin no.13 RB7 Seventh pin on port B. 'Data' line in program mode.
Pin no.14 Vdd Positive power supply pole.
Pin no.15 OSC2 Pin assigned for connecting with an oscillator
Pin no.16 OSC1 Pin assigned for connecting with an oscillator
Pin no.17 RA2 Second pin on port A. No additional function
Pin no.18 RA1 First pin on port A. No additional function.

2.7 Clock Generator – Oscillator

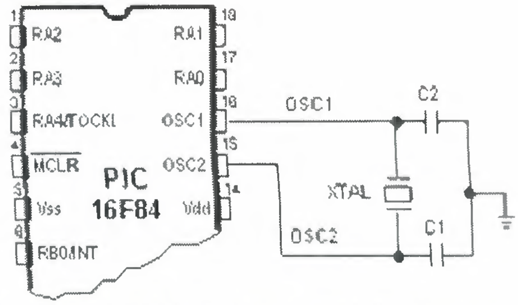
Oscillator circuit is used for providing a microcontroller with a clock. Clock is needed so that microcontroller could execute a program or program instructions.

2.7.1 Types of Oscillators

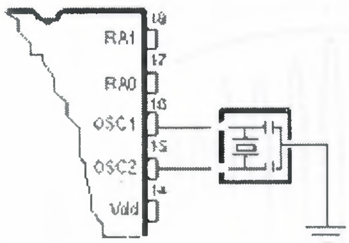
PIC16F84 can work with four different configurations of an oscillator. Since configurations with crystal oscillator and resistor - capacitor (RC) are the ones that are used most frequently, these are the only ones we will mention here. Microcontroller type with a crystal oscillator has in its designation XT , and a microcontroller with resistor-capacitor pair has a designation RC. This is important because you need to mention the type of oscillator when buying a microcontroller.

2.7.2 XT Oscillator

Crystal oscillator is kept in metal housing with two pins where you have written down the frequency at which crystal oscillates. One ceramic capacitor of 30pF whose other end is connected to the ground needs to be connected with each pin.
Oscillator and capacitors can be packed in joint case with three pins. Such element is called ceramic resonator and is represented in charts like the one below. Center pins of the element is the ground, while end pins are connected with OSC1 and OSC2 pins on the microcontroller. When designing a device, the rule is to place an oscillator nearer a microcontroller, so as to avoid any interference on lines on which microcontroller is receiving a clock.



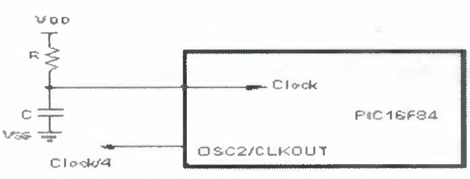
Connecting the quartz oscillator to give clock to a microcontroller



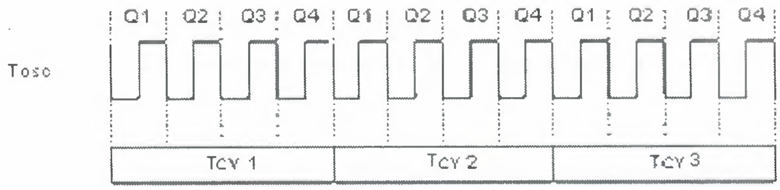
Connecting a resonator onto a microcontroller

2.7.3 RC Oscillator

In applications where great time precision is not necessary, RC oscillator offers additional savings during purchase. Resonant frequency of RC oscillator depends on supply voltage rate, resistance R, capacity C and working temperature. It should be mentioned here that resonant frequency is also influenced by normal variations in process parameters, by tolerance of external R and C components, etc.

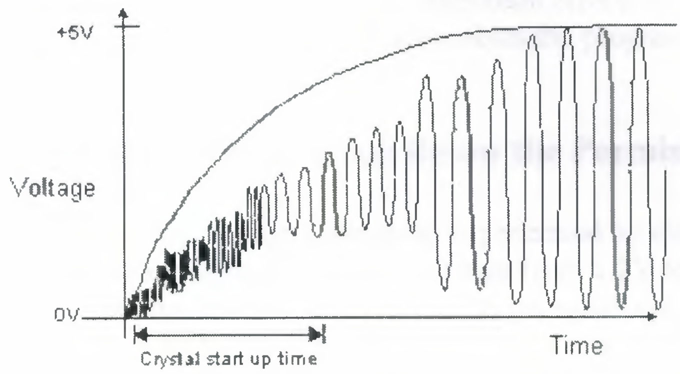


Above diagram shows how RC oscillator is connected with PIC16F84. With value of resistor R being below 2.2k, oscillator can become unstable, or it can even stop the oscillation. With very high value of R (ex.1M) oscillator becomes very sensitive to noise and humidity. It is recommended that value of resistor R should be between 3 and 100k. Even though oscillator will work without an external capacitor (C=0pF), capacitor above 20pF should still be used for noise and stability. No matter which oscillator is being used, in order to get a clock that microcontroller works upon, a clock of the oscillator must be divided by 4. Oscillator clock divided by 4 can also be obtained on OSC2/CLKOUT pin, and can be used for testing or synchronizing other logical circuits.



Relationship between a clock and a number of instruction cycles

Following a supply, oscillator starts oscillating. Oscillation at first has an unstable period and amplitude, but after some period of time it becomes stabilized.



Signal of an oscillator clock after receiving the supply of a microcontroller

To prevent such inaccurate clock from influencing microcontroller's performance, we need to keep the microcontroller in reset state during stabilization of oscillator's clock. Diagram above shows a typical shape of a signal which microcontroller gets from the quartz oscillator.

2.8 Reset

Reset is used for putting the microcontroller into a 'known' condition. That practically means that microcontroller can behave rather inaccurately under certain undesirable conditions. In order to continue its proper functioning it has to be reset, meaning all registers would be placed in a starting position. Reset is not only used when microcontroller doesn't behave the way we want it to, but can also be used when trying out a device as an interrupt in program execution, or to get a microcontroller ready when loading a program.

In order to prevent from bringing a logical zero to MCLR pin accidentally (line above it means that reset is activated by a logical zero), MCLR has to be connected via resistor to the positive supply pole. Resistor should be between 5 and 10K. This kind of resistor whose function is to keep a certain line on a logical one as a preventive, is called a pull up.

Microcontroller PIC16F84 knows several sources of resets:

a) Reset during power on, POR (Power-On Reset)
b) Reset during regular work by bringing logical zero to MCLR microcontroller's pin.
c) Reset during SLEEP regime

d) Reset at watchdog timer (WDT) overflow
e) Reset during at WDT overflow during SLEEP work regime.

The most important reset sources are a) and b). The first one occurs each time a power supply is brought to the microcontroller and serves to bring all registers to a starting position initial state. The second one is a product of purposeful bringing in of a logical zero to MCLR pin during normal operation of the microcontroller. This second one is often used in program development.

During a reset, RAM memory locations are not being reset. They are unknown during a power up and are not changed at any reset. Unlike these, SFR registers are reset to a starting position initial state. One of the most important effects of a reset is setting a program counter (PC) to zero (0000h) , which enables the program to start executing from the first written instruction.

2.8.1 Reset at Supply Voltage Drop Below the Permissible (Brown-out Reset)

Impulse for resetting during voltage voltage-up is generated by microcontroller itself when it detects an increase in supply Vdd (in a range from 1.2V to 1.8V). That impulse lasts 72ms which is enough time for an oscillator to get stabilized. These 72ms are provided by an internal PWRT timer which has its own RC oscillator. Microcontroller is in a reset mode as long as PWRT is active. However, as device is working, problem arises when supply doesn't drop to zero but falls below the limit that guarantees microcontroller's proper functioning. This is a likely case in practice, especially in industrial environment where disturbances and instability of supply are an everyday occurrence. To solve this problem we need to make sure that microcontroller is in a reset state each time supply falls below the approved limit. If, according to electrical specification, internal reset circuit of a microcontroller can not satisfy the needs, special electronic components can be used which are capable of generating the desired reset signal. Beside this function, they can also function in watching over supply voltage. If voltage drops below specified level, a logical zero would appear on MCLR pin which holds the microcontroller in reset state until voltage is not within limits that guarantee accurate performance.

2.9 Central Processing Unit

Central processing unit (CPU) is the brain of a microcontroller. That part is responsible for finding and fetching the right instruction which needs to be executed, for decoding that instruction, and finally for its execution.

Central processing unit connects all parts of the microcontroller into one whole. Surely, its most important function is to decode program instructions. When programmer writes a program, instructions have a clear form like MOVLW 0x20. However, in order for a microcontroller to understand that, this 'letter' form of an instruction must be translated into a series of zeros and ones which is called an 'opcode'. This transition from a letter to binary form is done by translators such as assembler translator (also known as an assembler). Instruction thus fetched from program memory must be decoded by a central processing unit. We can then select from the table of all the instructions a set of actions which execute a assigned task defined by instruction. As instructions may within themselves contain assignments which require different transfers of data from one memory into another, from memory onto ports, or some other calculations, CPU must be connected with all parts of the microcontroller. This is made possible through a data bus and an address bus.

Arithmetic logic unit is responsible for performing operations of adding, subtracting, moving (left or right within a register) and logic operations. Moving data inside a register is also known as 'shifting'. PIC16F84 contains an 8-bit arithmetic logic unit and 8-bit work registers.
In instructions with two operands, ordinarily one operand is in work register (W register), and the other is one of the registers or a constant. By operand we mean the contents on which some operation is being done, and a register is any one of the GPR or SFR registers. GPR is an abbreviation for 'General Purposes Registers', and SFR for 'Special Function Registers'. In instructions with one operand, an operand is either W register or one of the registers. As an addition in doing operations in arithmetic and logic, ALU controls status bits (bits found in STATUS register). Execution of some instructions affects status bits, which depends on the result itself. Depending on which instruction is being executed, ALU can affect values of Carry (C), Digit Carry (DC), and Zero (Z) bits in STATUS register.

2.9.1 STATUS Register

R/W-0	R/W-0	R/W-0	R/W-1	R/W-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C

bit7

Legend:
R = Readable bit **W** = Writable bit
U = Unimplemented bit, read as '0' - n = Value at power-on reset

bit 7 **IRP** (Register Bank Select bit)
Bit whose role is to be an eighth bit for purposes of indirect addressing the internal RAM.
1 = bank 2 and 3
0 = bank 0 and 1 (from 00h to FFh)

bits 6:5 **RP1:RP0** (Register Bank Select bits)
These two bits are upper part of the address for direct addressing. As instructions which address the memory directly have only seven bits, they need one more bit in order to address all 256 bytes which is how many bytes PIC16F84 has. RP1 bit is not used, but is left for some future expansions of this microcontroller.
01 = first bank
00 = zero bank
bit 4 **TO** Time-out ; Watchdog overflow.
Bit is set after turning on the supply and execution of CLRWDT and SLEEP instructions. Bit is reset when watchdog gets to the end signaling that overflow took place.

1 = overflow did not occur
0 = overflow did occur
bit 3 **PD** (Power-down bit)
This bit is set whenever power supply is brought to a microcontroller : as it starts running, after each regular reset and after execution of instruction CLRWDT.Instruction SLEEP resets it when microcontroller falls into low consumption mode. Its repeated setting is possible via reset or by turning the supply off/on . Setting can be triggered also by a signal on RB0/INT pin, change on RB port, upon writing to internal DATA EEPROM, and by a Watchdog.
1 = after supply has been turned on
0 = executing SLEEP instruction

bit 2 **Z** (Zero bit) Indication of a zero result
This bit is set when the result of an executed arithmetic or logic operation is zero.
1 = result equals zero
0 = result does not equal zero

bit 1 **DC** (Digit Carry) DC Transfer
Bit affected by operations of addition, subtraction. Unlike C bit, this bit represents transfer from the fourth resulting place. It is set in case of subtracting smaller from greater number and is reset in the other case.
1 = transfer occurred on the fourth bit according to the order of the result
0 = transfer did not occur
DC bit is affected by ADDWF, ADDLW, SUBLW, SUBWF instructions.

bit 0 **C** (Carry) Transfer
Bit that is affected by operations of addition, subtraction and shifting.
1 = transfer occurred from the highest resulting bit
0 = transfer did not occur
C bit is affected by ADDWF, ADDLW, SUBLW, SUBWF instructions.

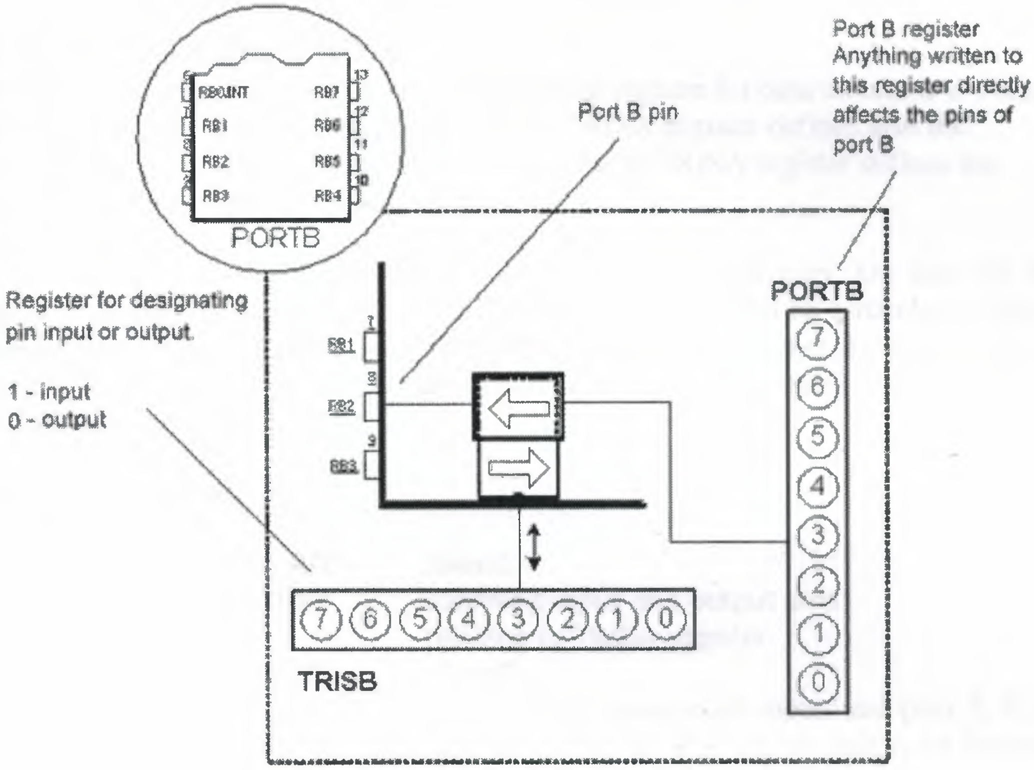
2.10 Ports

Term "port" refers to a group of pins on a microcontroller which can be accessed simultaneously, or on which we can set the desired combination of zeros and ones, or read from them an existing status. Physically, port is a register inside a microcontroller which is connected by wires to the pins of a microcontroller. Ports represent physical connection of Central Processing Unit with an outside world. Microcontroller uses them in order to monitor or control other components or devices. Due to functionality, some pins have twofold roles like PA4/TOCKI for instance, which is in the same time the fourth bit of port A and an external input for free-run counter. Selection of one of these two pin functions is done in one of the configuration registers. An illustration of this is the fifth bit TOCS in OPTION register. By selecting one of the functions the other one is disabled.
All port pins can be designated as input or output, according to the needs of a device that's being developed. In order to define a pin as input or output pin, the right combination of zeros and ones must be written in TRIS register. If the appropriate bit of TRIS register contains logical "1", then that pin is an input pin, and if the opposite is true, it's an output pin. Every port has its proper TRIS register. Thus, port A has TRISA, and port B has TRISB. Pin direction can be changed during the course of work which is

particularly fitting for one-line communication where data flow constantly changes direction. PORTA and PORTB state registers are located in bank 0, while TRISA and TRISB pin direction registers are located in bank 1.

2.10.1 PORTB and TRISB

PORTB has adjoined 8 pins. The appropriate register for data direction is TRISB. Setting a bit in TRISB register defines the corresponding port pin as input, and resetting a bit in TRISB register defines the corresponding port pin as output.



Each PORTB pin has a weak internal pull-up resistor (resistor which defines a line to logic one) which can be activated by resetting the seventh bit RBPU in OPTION register. These 'pull-up' resistors are automatically being turned off when port pin is configured as an output. When a microcontroller is started, pull-ups are disabled.

Four pins PORTB, RB7:RB4 can cause an interrupt which occurs when their status changes from logical one into logical zero and opposite. Only pins configured as input can cause this interrupt to occur (if any RB7:RB4 pin is configured as an output, an interrupt won't be generated at the change of status.) This interrupt option along with internal pull-up resistors makes it easier to solve common problems we find in practice like for instance that of matrix keyboard. If rows on the keyboard are connected to these pins, each push on a key will then cause an interrupt. A microcontroller will determine which key is at hand while processing an interrupt It is not recommended to refer to port B at the same time that interrupt is being processed.

```
bsf     STATUS, RP0     ;Bank1
movlw   0x0F            ;Defining input and output pins
movwf   TRISB           ;Writing to TRISB register
bcf     STATUS, RP0     ;Bank0
bsf     PORTB, 4        ;PORTB <7:4>=0
bsf     PORTB, 5
bsf     PORTB, 6
bsf     PORTB, 7
```

The above example shows how pins 0, 1, 2, and 3 are designated input, and pins 4, 5, 6, and 7 for output, after which PORTB output pins are set to one.

2.10.2 PORTA and TRISA

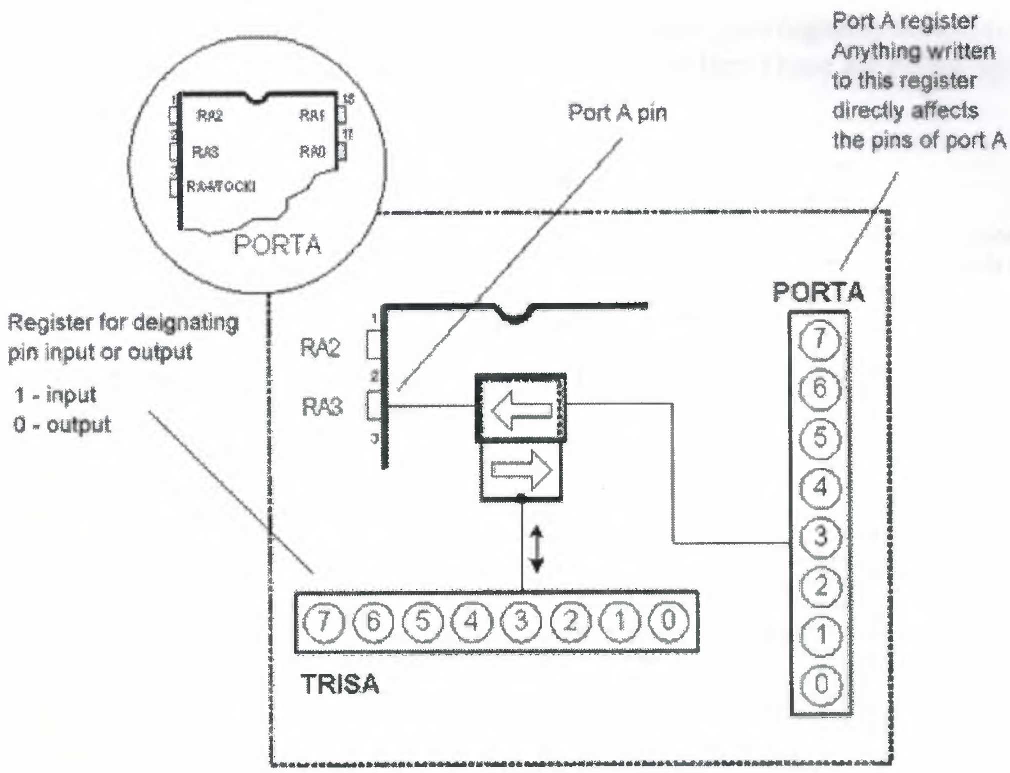
PORTA has 5 adjoining pins. The corresponding register for data direction is TRISA at address 85h. Like with port B, setting a bit in TRISA register defines also the corresponding port pin as input, and clearing a bit in TRISA register defines the corresponding port pin as output.

It is important to note that PORTA pin RA4 can be input only. On that pin is also situated an external input for timer TMR0. Whether RA4 will be a standard input or an input for a counter depends on T0CS bit (*TMR0 Clock Source Select bit*). This pin enables the timer TMR0 to increment either from internal oscillator or via external impulses on RA4/T0CKI pin.

Configuring port A:

```
bsf     STATUS, RP0     ;Bank1
movlw   b'11111100'     ;Defining input and output pins
movwf   TRISA           ;Writing to TRISA register
bcf     STATUS, RP0     ;Bank0
```

Example shows how pins 0, 1, 2, 3, and 4 are designated input, and pins 5, 6, and 7 output. After this, it is possible to read the pins RA2, RA3, RA4, and to set logical zero or one to pins RA0 and RA1.



2.11 Memory Organization

PIC16F84 has two separate memory blocks, one for data and the other for program. EEPROM memory with GPR and SFR registers in RAM memory make up the data block, while FLASH memory makes up the program block.

2.11.1 Program Memory

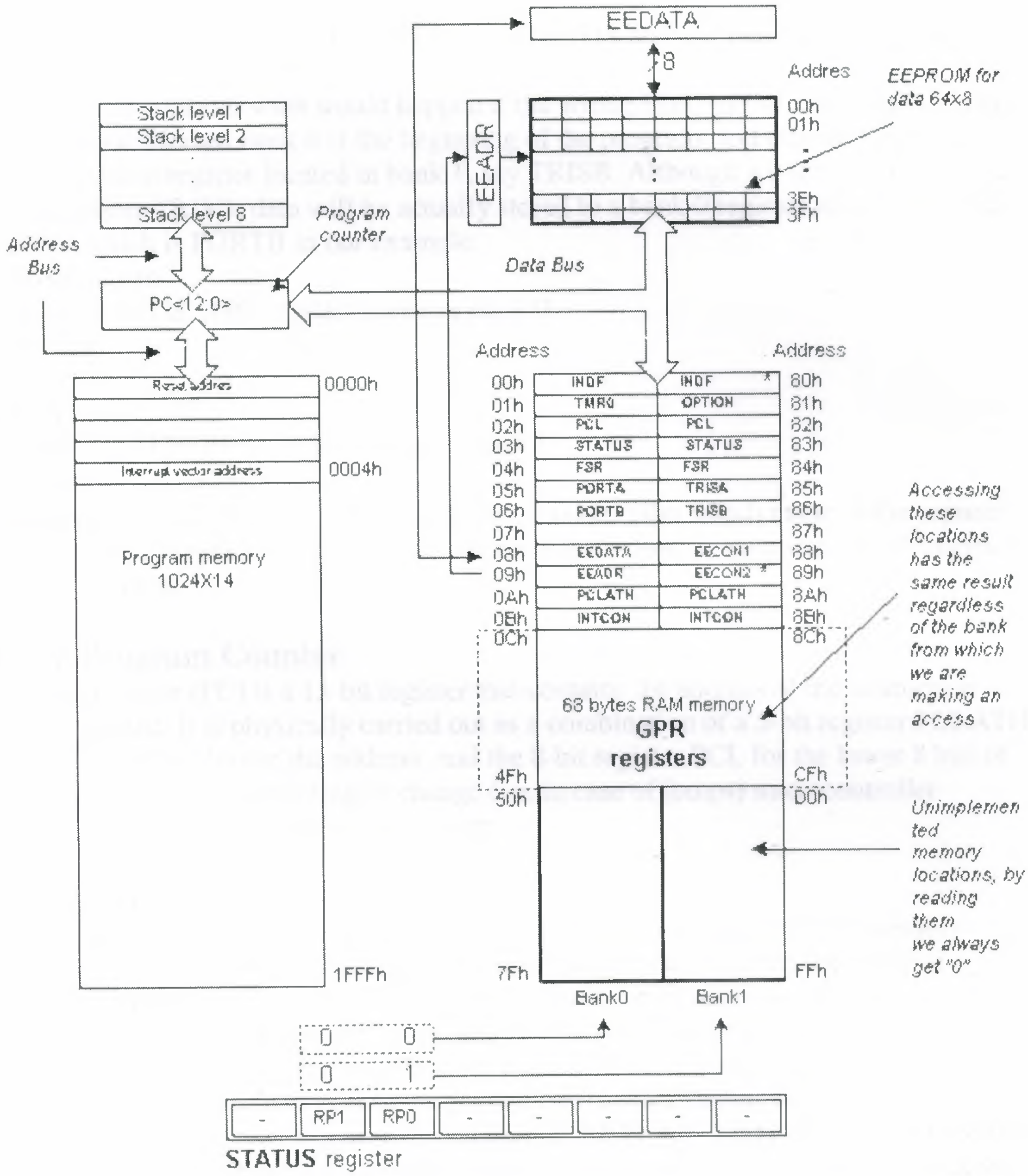
Program memory has been carried out in FLASH technology which makes it possible to program a microcontroller many times before it's installed into a device, and even after its installment if eventual changes in program or process parameters should occur. The size of program memory is 1024 locations with 14 bits width where locations zero and four are reserved for reset and interrupt vector.

2.11.2 Data Memory

Data memory consists of EEPROM and RAM memories. EEPROM memory consists of 64 eight bit locations whose contents is not lost during loosing of power supply. EEPROM is not directly addressable, but is accessed indirectly through EEADR and EEDATA registers. As EEPROM memory usually serves for storing important parameters (for example, of a given temperature in temperature regulators) , there is a strict procedure for writing in EEPROM which must be followed in order to avoid accidental writing. RAM memory for data occupies space on a memory map from location 0x0C to 0x4F which comes to 68 locations. Locations of RAM memory are also called GPR registers which is an abbreviation for General Purpose Registers. GPR registers can be accessed regardless of which bank is selected at the moment.

2.11.3 SFR Registers

Registers which take up first 12 locations in banks 0 and 1 are registers of specialized function assigned with certain blocks of the microcontroller. These are called Special Function Registers.



Memory organization of microcontroller PIC16F84

2.11.4 Memory Banks

Beside this 'length' division to SFR and GPR registers, memory map is also divided in 'width' (see preceding map) to two areas called 'banks'. Selecting one of the banks is done via RP0 bit in STATUS register.

Example:

bcf STATUS, RP0

Instruction BCF clears bit RP0 (RP0=0) in STATUS register and thus sets up bank 0.

bsf STATUS, RP0

Instruction BSF sets the bit RP0 (RP0=1) in STATUS register and thus sets up bank1.

It is useful to consider what would happen if the wrong bank was selected. Let's assume that we have selected bank 0 at the beginning of the program, and that we now want to write to certain register located in bank 1, say TRISB. Although we specified the name of the register TRISB, data will be actually stored to a bank 0 register at the appropriate address, which is PORTB in our example.

```
BANK0 macro
    Bcf STATUS, RP0   ;Select memory bank 0
    endm

BANK1 macro
    Bsf STATUS, RP0   ;Select memory bank 1
    endm
```

Bank selection can be also made via directive *banksel* after which name of the register to be accessed is specified. In this manner, there is no need to memorize which register is in which bank.

2.11.5 Program Counter

Program counter (PC) is a 13-bit register that contains the address of the instruction being executed. It is physically carried out as a combination of a 5-bit register PCLATH for the five higher bits of the address, and the 8-bit register PCL for the lower 8 bits of the address.By its incrementing or change (i.e. in case of jumps) microcontroller executes program instructions step-by-step.

2.11.6 Stack

PIC16F84 has a 13-bit stack with 8 levels, or in other words, a group of 8 memory locations, 13 bits wide, with special purpose. Its basic role is to keep the value of program counter after a jump from the main program to an address of a subprogram . In order for a program to know how to go back to the point where it started from, it has to return the value of a program counter from a stack. When moving from a program to a subprogram, program counter is being pushed onto a stack (example of this is CALL instruction). When executing instructions such as RETURN, RETLW or RETFIE which were executed at the end of a subprogram, program counter was taken from a stack so that program could continue where was stopped before it was interrupted. These operations of placing on and taking off from a program counter stack are called PUSH and POP, and are named according to similar instructions on some bigger microcontrollers.

2.11.7 In System Programming

In order to program a program memory, microcontroller must be set to special working mode by bringing up MCLR pin to 13.5V, and supply voltage Vdd has to be stabilized between 4.5V to 5.5V. Program memory can be programmed serially using two 'data/clock' pins which must previously be separated from device lines, so that errors wouldn't come up during programming.

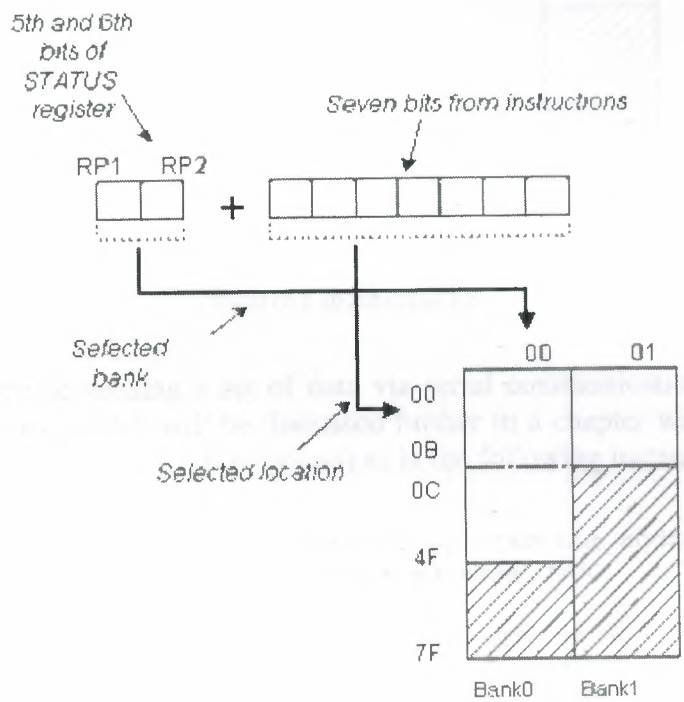
2.11.8 Addressing Modes

RAM memory locations can be accessed directly or indirectly.

2.11.9 Direct Addressing

Direct Addressing is done through a 9-bit address. This address is obtained by connecting 7th bit of direct address of an instruction with two bits (RP1, RP0) from STATUS register as is shown on the following picture. Any access to SFR registers is an example of direct addressing.

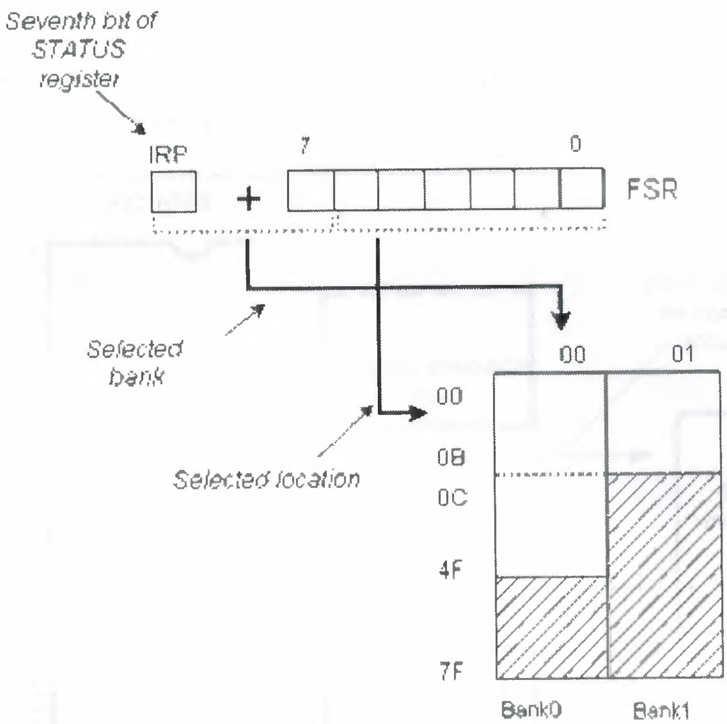
```
Bsf STATUS, RP0 ;Bankl
movlw 0xFF      ;w=0xFF
movwf TRISA     ;address of TRISA register is taken from
                ;instruction movwf
```



2.11.10 Indirect Adressing

Indirect unlike direct addressing does not take an address from an instruction but derives it from IRP bit of STATUS and FSR registers. Addressed location is accessed via INDF register which in fact holds the address indicated by a FSR. In other words, any instruction which uses INDF as its register in reality accesses data indicated by a FSR register. Let's say, for instance, that one general purpose register (GPR) at address 0Fh contains a value of 20. By writing a value of 0Fh in FSR register we will get a register indicator at address 0Fh, and by reading from INDF register, we will get a value of 20, which means that we have read from the first register its value without accessing it directly (but via FSR and INDF). It appears that this type of addressing does not have any advantages over direct addressing, but certain needs do exist during programming which can be solved smoothly only through indirect addressing.

Indirect addressing is very convenient for manipulating data arrays located in GPR registers. In this case, it is necessary to initialize FSR register with a starting address of the array, and the rest of the data can be accessed by incrementing the FSR register.



Indirect addressing

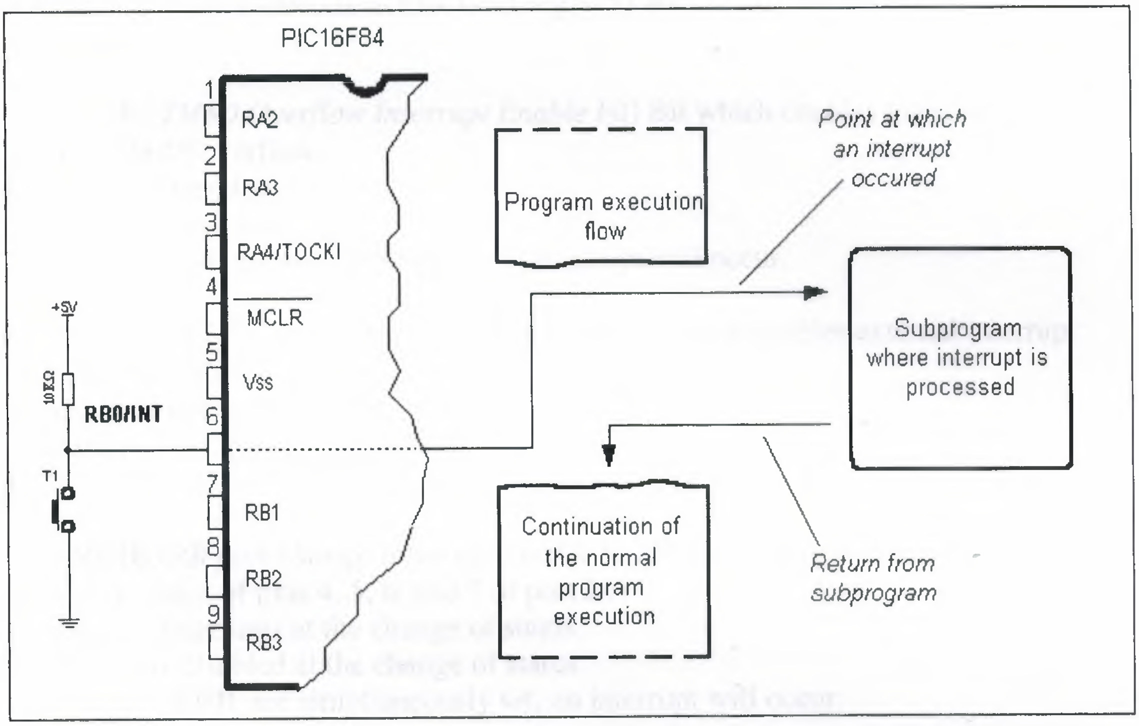
Such examples include sending a set of data via serial communication, working with buffers and indicators (which will be discussed further in a chapter with examples), or erasing a part of RAM memory (16 locations) as in the following instance.

```
       Movlw 0x0C          ;initialization of starting address
       Movwf FSR           ;FSR indicates address 0x0C
LOOP   clrf INDF           ;INDF = 0
       incf FSR            ;address = initial address + 1
       btfss FSR,4         ;are all locations erased
       goto loop           ;no, go through a loop again
CONTINUE
       :                   ; yes, continue with program
```

Reading data from INDF register when the contents of FSR register is equal to zero returns the value of zero, and writing to it results in NOP operation (no operation).

2.12 Interrupts

Interrupts are a mechanism of a microcontroller which enables it to respond to some events at the moment they occur, regardless of what microcontroller is doing at the time. This is a very important part, because it provides connection between a microcontroller and environment which surrounds it. Generally, each interrupt changes the program flow, interrupts it and after executing an interrupt subprogram (interrupt routine) it continues from that same point on.



One of the possible sources of interrupt and how it affects the main program

Control register of an interrupt is called INTCON and can be accessed regardless of the bank selected. Its role is to allow or disallowed interrupts, and in case they are not allowed, it registers single interrupt requests through its own bits.

2.12.1 INTCON Register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
bit7							

Legend:
R = Readable bit **W** = Writable bit
U = Unimplemented bit, read as '0' - n = Value at power-on reset

Bit 7 **GIE** (*Global Interrupt Enable bit*) Bit which enables or disables all interrupts.
1 = all interrupts are enabled
0 = all interrupts are disabled

Bit 6 **EEIE** (*EEPROM Write Complete Interrupt Enable bit*) Bit which enables an interrupt at the end of a writing routine to EEPROM
1 = interrupt enabled
0 = interrupt disabled
If EEIE and EEIF (which is in EECON1 register) are set simultaneously , an interrupt will occur.

bit 5 **T0IE** (*TMR0 Overflow Interrupt Enable bit*) Bit which enables interrupts during counter TMR0 overflow.
1 = interrupt enabled
0 = interrupt disabled
If T0IE and T0IF are set simultaneously, interrupt will occur.

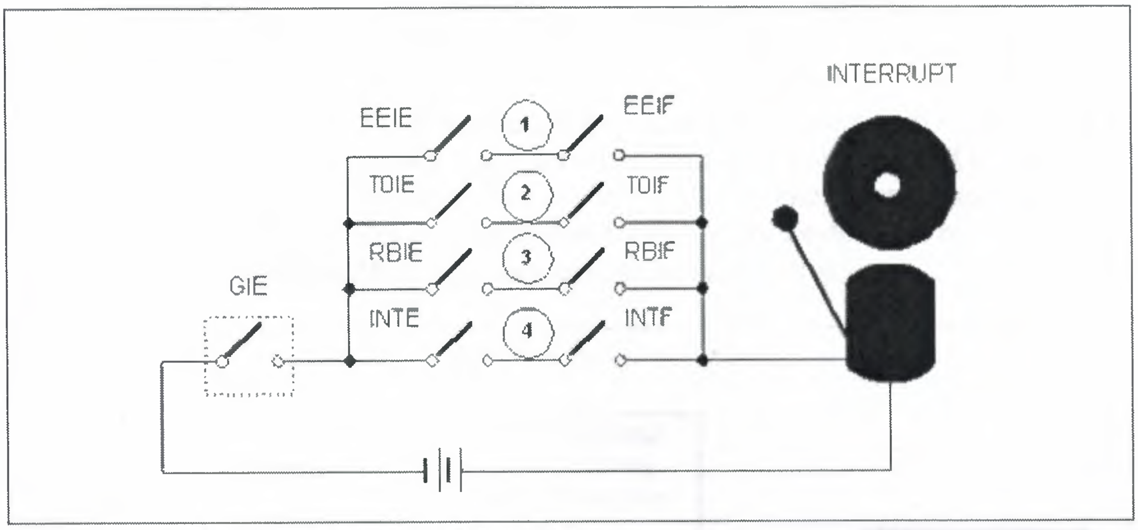
bit 4 **INTE** (*INT External Interrupt Enable bit*) Bit which enables external interrupt from pin RB0/INT.
1 = external interrupt enabled
0 = external interrupt disabled
If INTE and INTF are set simultaneously, an interrupt will occur.

bit 3 **RBIE** (*RB port change Interrupt Enable bit*) Enables interrupts to occur at the change of status of pins 4, 5, 6, and 7 of port B.
1 = enables interrupts at the change of status
0 =interrupts disabled at the change of status
If RBIE and RBIF are simultaneously set, an interrupt will occur.

bit 2 **T0IF** (TMR0 Overflow Interrupt Flag bit) Overflow of counter TMR0.
1 = counter changed its status from FFh to 00h
0 = overflow did not occur
Bit must be cleared in program in order for an interrupt to be detected.

bit 1 **INTF** (INT External Interrupt Flag bit) External interrupt occurred.
1 = interrupt occurred
0 = interrupt did not occur
If a rising or falling edge was detected on pin RB0/INT, (which is defined with bit INTEDG in OPTION register), bit INTF is set.

bit 0 **RBIF** (RB Port Change Interrupt Flag bit) Bit which informs about changes on pins 4, 5, 6 and 7 of port B.
1 = at least one pin has changed its status
0 = no change occurred on any of the pins
Bit has to be cleared in an interrupt subroutine to be able to detect further interrupts.



Simplified outline of PIC16F84 microcontroller interrupt

PIC16F84 has four interrupt sources:

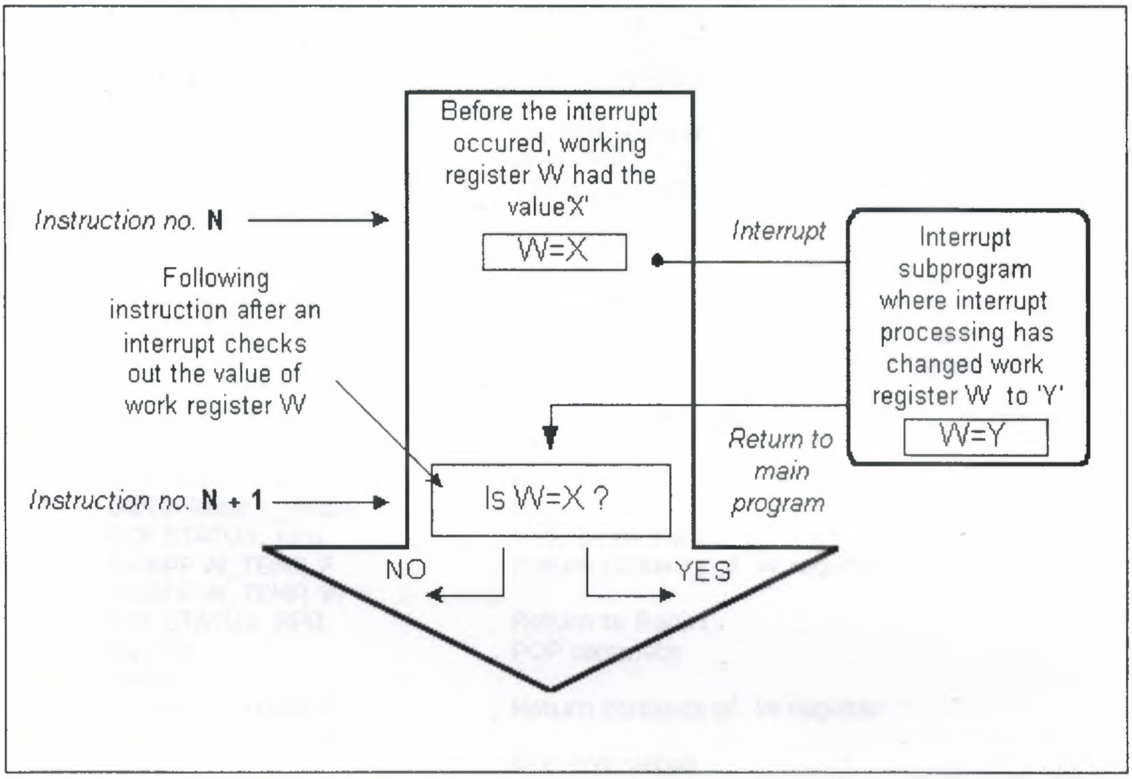
1. Termination of writing data to EEPROM
2. TMR0 interrupt caused by timer overflow
3. Interrupt during alteration on RB4, RB5, RB6 and RB7 pins of port B.
4. External interrupt from RB0/INT pin of microcontroller

Generally speaking, each interrupt source has two bits joined to it. One enables interrupts, and the other detects when interrupts occur. There is one common bit called GIE which can be used to disallow or enable all interrupts simultaneously. This bit is very useful when writing a program because it allows for all interrupts to be disabled for a period of time, so that execution of some important part of a program would not be interrupted. When instruction which resets GIE bit was executed (GIE=0, all interrupts disallowed), any interrupt that remained unsolved should be ignored. Interrupts which remained unsolved and were ignored, are processed when GIE bit (GIE=1, all interrupts allowed) would be cleared. When interrupt was answered, GIE bit was cleared so that any additional interrupts would be disabled, return address was pushed onto stack and address 0004h was written in program counter - only after this does replying to an interrupt begin! After interrupt is processed, bit whose setting caused an interrupt must be cleared, or interrupt routine would automatically be processed over again during a return to the main program.

2.12.2 Keeping the Contents of Important Registers

Only return value of program counter is stored on a stack during an interrupt (by return value of program counter we mean the address of the instruction which was to be executed, but wasn't because interrupt occurred). Keeping only the value of program counter is often not enough. Some registers which are already in use in the main program can also be in use in interrupt routine. If they were not retained, main program would during a return from an interrupt routine get completely different values in those registers, which would cause an error in the program. One example for such a case is contents of the work register W. If we suppose that main program was using work register W for some of its operations, and if it had stored in it some value that's important for the following instruction, then an interrupt which occurs before that instruction would change the value of work register W which would directly be influenced the main program.

Procedure of recording important registers before going to an interrupt routine is called PUSH, while the procedure which brings recorded values back, is called POP. PUSH and POP are instructions with some other microcontrollers (Intel), but are so widely accepted that a whole operation is named after them. PIC16F84 does not have instructions like PUSH and POP, and they have to be programmed.



Common error: saving the value wasn't done before entering the interrupt routine

Due to simplicity and frequent usage, these parts of the program can be made as macros. The concept of a Macro is explained in "Program assembly language". In the following example, contents of W and STATUS registers are stored in W_TEMP and STATUS_TEMP variables prior to interrupt routine. At the beginning of PUSH routine we need to check presently selected bank because W_TEMP and STATUS_TEMP are found in bank 0. For exchange of data between these registers, SWAPF instruction is used instead of MOVF because it does not affect the STATUS register bits.

Example is an assembler program for following steps:

1. Testing the current bank
2. Storing W register regardless of the current bank
3. Storing STATUS register in bank 0.
4. Executing interrupt routine for interrupt processing (ISR)
5. Restores STATUS register
6. Restores W register

If there are some more variables or registers that need to be stored, then they need to be kept after storing STATUS register (step 3), and brought back before STATUS register is restored (step 5).

```
Push
        BTFSS STATUS, RP0          ; Bank0
        GOTO RP0CLEAR              ; Yes
        BCF STATUS, RP0            ; NO, go to Bank0
        MOVWF W_TEMP               ; Save W register
        SWAPF STATUS, W            ; W <- STATUS
        MOVWF STATUS_TEMP          ; STATUS_TEMP <- W
        BSF STATUS_TEMP, 1         ; RP0(STATUS_TEMP)=1
        GOTO ISR_Code              ; Push completed
RP0CLEAR
        MOVWF W_TEMP               ; Save W register
        SWAPF STATUS, W            ; W <- STATUS
        MOVWF STATUS_TEMP          ; STATUS_TEMP <- W
;
ISR_Code
        :
        : (Interrupt subprogram )
        :
;
Pop
        SWAPF STATUS_TEMP, W       ; W <- STATUS_TEMP
        MOVWF STATUS               ; STATUS <-W
        BTFSS STATUS, RP0          ; Bank1?
        GOTO Return_WREG           ; NO,
        BCF STATUS, RP0            ; YES, go to Bank0
        SWAPF W_TEMP, F            ; Return contents of  W register
        SWAPF W_TEMP, W            ;
        BSF STATUS, RP0            ; Return to Bank1
        RETFIE                     ; POP complete
Return_WREG
        SWAPF W_TEMP, F            ; Return contents of W register
        SWAPF W_TEMP, W            ;
        RETFIE                     ; POP completed
```

The same example can be carried out using macros, thus getting a more legible program. Macros that are already defined can be used for writing new macros. Macros BANK1 and BANK0 which are explained in "Memory organization" chapter are used with macros 'push' and 'pop'.

```
push    macro
        movwf   W_Temp              ;W_Temp <- W
        swapf   W_Temp,F            ;Swap them
        BANK1                       ;Macro for switching to Bank1
        swapf   OPTION_REG,W        ;W <- OPTION_REG
        movwf   Option_Temp         ;Option_Temp <- W
        BANK0                       ;macro for switching to Bank0
        swapf   STATUS,W            ;W <- STATUS
        movwf   Stat_Temp           ;Stat_Temp <-W
        endm                        ;End of push macro

pop     macro
        swapf   Stat_Temp,W         ;W <- Stat_Temp
        movwf   STATUS              ;STATUS <- W
        BANK1                       ;Macro for switching to Bank1
        swapf   Option_Temp,W       ;W <- Option_Temp
        movwf   OPTION_REG          ;OPTION_REG <- W
        BANK0                       ;Macro for switching to Bank0
        swapf   W_Temp,W            ;W <- W_Temp
        endm                        ;End of a pop macro
```

2.12.3 External Interrupt on RB0/INT Pin of Microcontroller

External interrupt on RB0/INT pin is triggered by rising signal edge (if bit INTEDG=1 in OPTION<6> register), or falling edge (if INTEDG=0). When correct signal appears on INT pin, INTF bit is set in INTCON register. INTF bit (INTCON<1>) must be cleared in interrupt routine, so that interrupt wouldn't occur again while going back to the main program. This is an important part of the program which programmer must not forget, or program will constantly go into interrupt routine. Interrupt can be turned off by resetting INTE control bit (INTCON<4>). Possible application of this interrupt could be measuring the impulse width or pause length, i.e. input signal frequency. Impulse duration can be measured by first enabling the interrupt on rising edge, and upon its appearing, starting the timer and then enabling the interrupt on falling edge. Timer should be stopped upon the appearing of falling edge - measured time period represents the impulse duration.

2.12.4 Interrupt During a TMR0 Counter Overflow

Overflow of TMR0 counter (from FFh to 00h) will set T0IF (INTCON<2>) bit. This is very important interrupt because many real problems can be solved using this interrupt. One of the examples is time measurement. If we know how much time counter needs in order to complete one cycle from 00h to FFh, then a number of interrupts multiplied by that amount of time will yield the total of elapsed time. In interrupt routine some variable would be incremented in RAM memory, value of that variable multiplied by the amount of time the counter needs to count through a whole cycle, would yield total elapsed time. Interrupt can be turned on/off by setting/resetting T0IE (INTCON<5>) bit.

2.12.5 Interrupt Upon a Change on Pins 4, 5, 6 and 7 of port B

Change of input signal on PORTB <7:4> sets RBIF (INTCON<0>) bit. Four pins RB7, RB6, RB5 and RB4 of port B, can trigger an interrupt which occurs when status on them changes from logic one to logic zero, or vice versa. For pins to be sensitive to this change, they must be defined as input. If any one of them is defined as output, interrupt will not be generated at the change of status. If they are defined as input, their current state is compared to the old value which was stored at the last reading from port B.

2.12.6 Interrupt Upon Finishing Write-Subroutine to EEPROM

This interrupt is of practical nature only. Since writing to one EEPROM location takes about 10ms (which is a long time in the notion of a microcontroller), it doesn't pay off to a microcontroller to wait for writing to end. Thus interrupt mechanism is added which allows the microcontroller to continue executing the main program, while writing in EEPROM is being done in the background. When writing is completed, interrupt informs the microcontroller that writing has ended. EEIF bit, through which this informing is done, is found in EECON1 register. Occurrence of an interrupt can be disabled by resetting the EEIE bit in INTCON register.

2.12.7 Interrupt Initialization

In order to use an interrupt mechanism of a microcontroller, some preparatory tasks need to be performed. These procedures are in short called "initialization". By initialization we define to what interrupts the microcontroller will respond, and which ones it will ignore. If we do not set the bit that allows a certain interrupt, program will not execute an interrupt subprogram. Through this we can obtain control over interrupt occurrence, which is very useful.

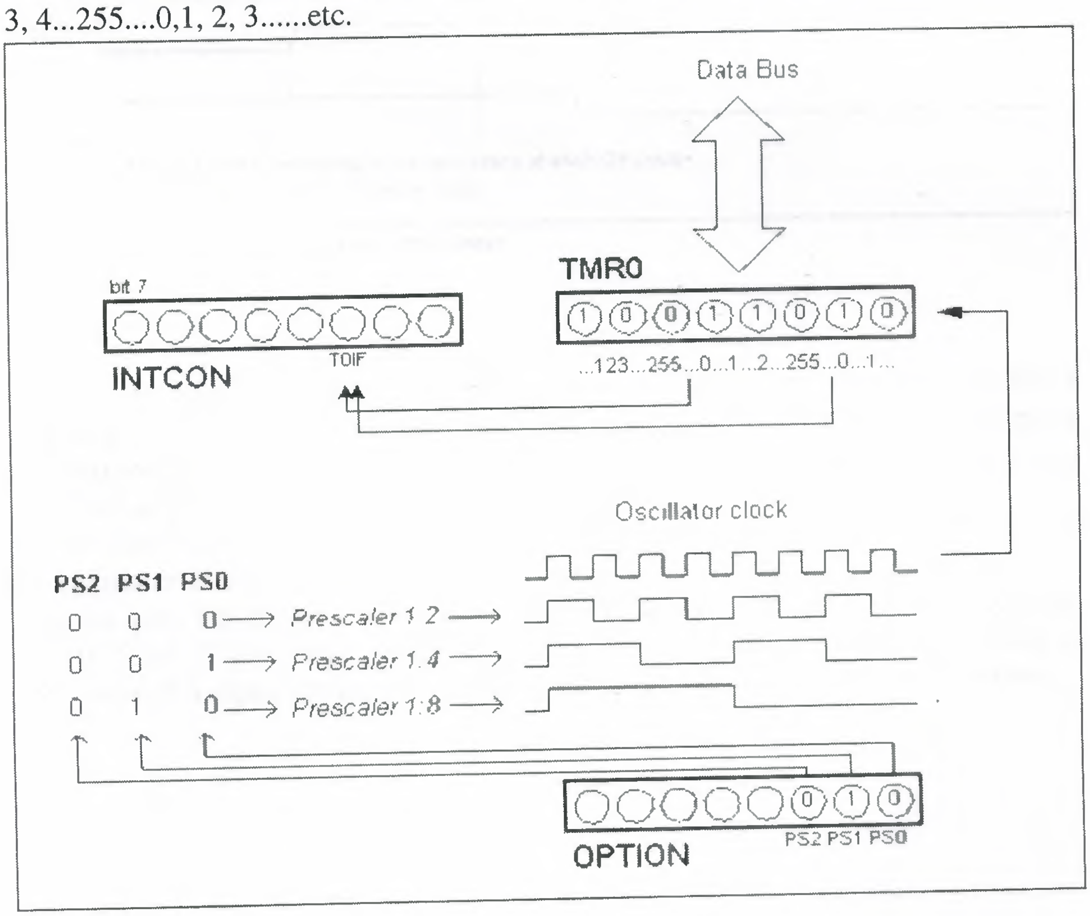
```
clrf INTCON              ; all interrupts disabled
movlw B'00010000'        ; external interrupt only is enabled
bsf INTCON, GIE          ; occurrence of interrupts allowed
```

The above example shows initialization of external interrupt on RB0 pin of a microcontroller. Where we see one being set, that means that interrupt is enabled. Occurrence of other interrupts is not allowed, and interrupts are disabled altogether until GIE bit is set to one.

The following example shows a typical way of handling interrupts. PIC16F84 has got a single location for storing the address of an interrupt subroutine. This means that first we need to detect which interrupt is at hand (if more than one interrupt source is available), and then we can execute that part of a program which refers to that interrupt.

2.13 Free-run Timer TMR0

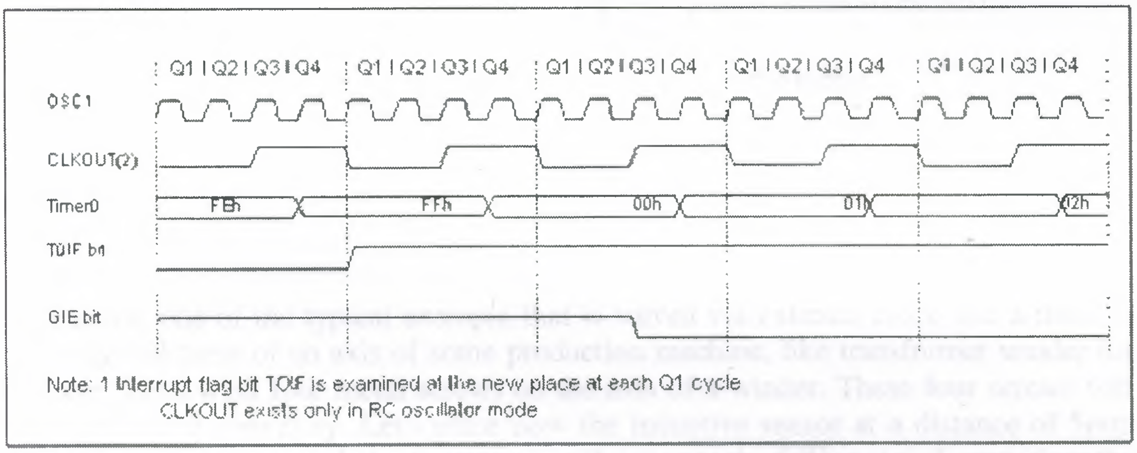
Timers are usually the most complicated parts of a microcontroller, so it is necessary to set aside more time for understanding them thoroughly. Through their application it is possible to establish relations between a real dimension such as "time" and a variable which represents status of a timer within a microcontroller. Physically, timer is a register whose value is continually increasing to 255, and then it starts all over again: 0, 1, 2, 3, 4...255....0,1, 2, 3......etc.



Relation between the timer TMR0 and prescaler

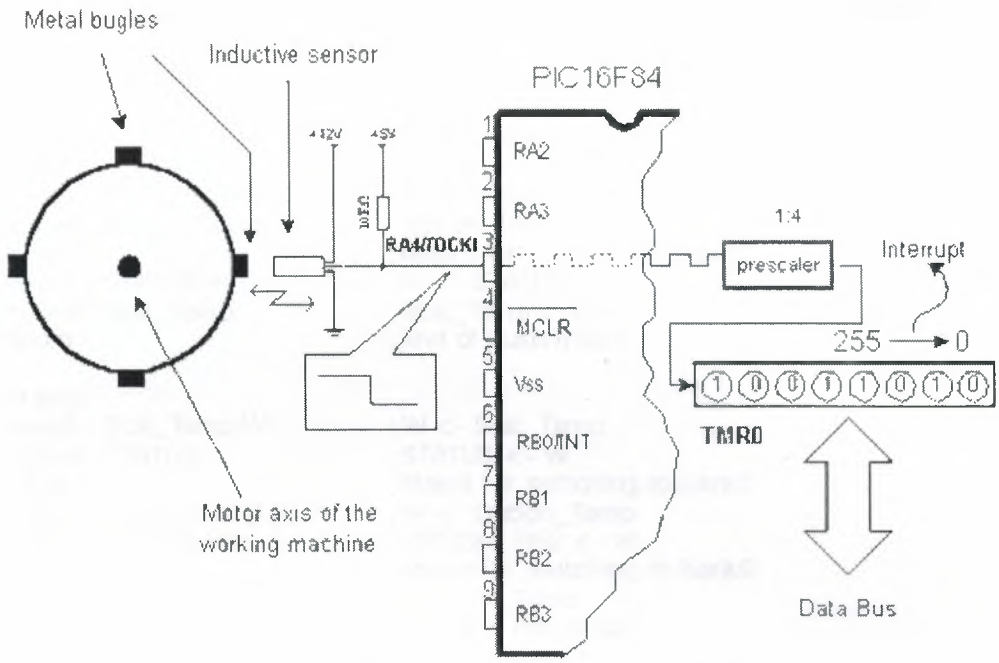
This incrementing is done in the background of everything a microcontroller does. It is up to programmer to think up a way how he will take advantage of this characteristic for his needs. One of the ways is increasing some variable on each timer overflow. If we know how much time a timer needs to make one complete round, then multiplying the value of a variable by that time will yield the total amount of elapsed time.

PIC16F84 has an 8-bit timer. Number of bits determines what value timer counts to before starting to count from zero again. In the case of an 8-bit timer, that number is 256. A simplified scheme of relation between a timer and a prescaler is represented on the previous diagram. Prescaler is a name for the part of a microcontroller which divides oscillator clock before it will reach logic that increases timer status. Number which divides a clock is defined through first three bits in OPTION register. The highest divisor is 256. This actually means that only at every 256th clock, timer value would increase by one. This provides us with the ability to measure longer timer periods.



Time diagram of interrupt occurence with TMR0 timer

After each count up to 255, timer resets its value to zero and starts with a new cycle of counting to 255. During each transition from 255 to zero, T0IF bit in INTCOM register is set. If interrupts are allowed to occur, this can be taken advantage of in generating interrupts and in processing interrupt routine. It is up to programmer to reset T0IF bit in interrupt routine, so that new interrupt, or new overflow could be detected. Beside the internal oscillator clock, timer status can also be increased by the external clock on RA4/TOCKI pin. Choosing one of these two options is done in OPTION register through T0CS bit. If this option of external clock was selected, it would be possible to define the edge of a signal (rising or falling), on which timer would increase its value.



Determining a number of full axis turns of the motor

In practice, one of the typical example that is solved via external clock and a timer is counting full turns of an axis of some production machine, like transformer winder for instance. Let's wind four metal screws on the axis of a winder. These four screws will represent metal convexity. Let's place now the inductive sensor at a distance of 5mm from the head of a screw. Inductive sensor will generate the falling signal every time the head of the screw is parallel with sensor head. Each signal will represent one fourth of a full turn, and the sum of all full turns will be found in TMR0 timer. Program can easily read this data from the timer through a data bus.

The following example illustrates how to initialize timer to signal falling edges from external clock source with a prescaler 1:4. Timer works in "polig" mode.

```
        clrf TMR0           ;TMR0=0
        clrf INTCON         ;Interrupts and T0IF=0 disallowed
        bsf STATUS,RP0      ;Bank1 because of OPTION_REG
        movlw B'00110001'  ;prescaler 1:4, falling edge selected external
                            ;clock source and pull up ;selected resistors
                            ;on port B activated
        movwf OPTION_REG ;OPTION_REG <- W
T0_OVFL
        btfss INTCON, T0IF        ;testing overflow bit
        goto T0_OVFL              ;interrupt has not occured yet, wait
;
; (Part of the program which processes data regarding a number of turns)
;
goto T0_OVFL                      ;waiting for new overflow
```

The same example can be carried out through an interrupt in the following way:

```
push   macro
       movwf  W_Temp                ;W_Temp <- W
       swapf  W_Temp,F              ;Swap them
       BANK1                        ;Macro for switching to Bank1
       swapf  OPTION_REG,W          ;W <- OPTION_REG
       movwf  Option_Temp           ;Option_Temp <- W
       BANK0                        ;macro for switching to Bank0
       swapf  STATUS,W              ;W <- STATUS
       movwf  Stat_Temp             ;Stat_Temp <-W
       endm                         ;End of push macro

pop    macro
       swapf  Stat_Temp,W           ;W <- Stat_Temp
       movwf  STATUS                ;STATUS <- W
       BANK1                        ;Macro for switching to Bank1
       swapf  Option_Temp,W         ;W <- Option_Temp
       movwf  OPTION_REG            ;OPTION_REG <- W
       BANK0                        ;Macro for switching to Bank0
       swapf  W_Temp,W              ;W <- W_Temp
       endm                         ;End of a pop macro
```

Prescaler can be assigned either timer TMR0 or a watchdog. Watchdog is a mechanism which microcontroller uses to defend itself against programs getting stuck. As with any other electrical circuit, so with a microcontroller too can occur failure, or some work impairment. Unfortunately, microcontroller also has program where problems can occur as well. When this happens, microcontroller will stop working and will remain in that state until someone resets it. Because of this, watchdog mechanism has been introduced. After a certain period of time, watchdog resets the microcontroller (microcontroller in fact resets itself). Watchdog works on a simple principle: if timer overflow occurs, microcontroller is reset, and it starts executing a program all over again. In this way, reset will occur in case of both correct and incorrect functioning. Next step is preventing reset in case of correct functioning, which is done by writing zero in WDT register (instruction CLRWDT) every time it nears its overflow. Thus program will prevent a reset as long as it's executing correctly. Once it gets stuck, zero will not be written, overflow of WDT timer and a reset will occur which will bring the microcontroller back to correct functioning again.

Prescaler is accorded to timer TMR0, or to watchdog timer trough PSA bit in OPTION register. By clearing PSA bit, prescaler will be accorded to timer TMR0. When prescaler is accorded to timer TMR0, all instructions of writing to TMR0 register (CLRF TMR0, MOVWF TMR0, BSF TMR0,...) will clear prescaler. When prescaler is assigned to a watchdog timer, only CLRWDT instruction will clear a prescaler and watchdog timer at the same time . Prescaler change is completely under programmer's control, and can be changed while program is running.

2.13.1 OPTION Control Register

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{RBPU}$ (1)	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

Legend:
R = Readable bit **W** = Writable bit
U = Unimplemented bit, read as '0' -n = Value at POR reset

bit 7 **RBPU** *(*PORTB Pull-up Enable bit*)*
This bit turns internal pull-up resistors on port B on or off.
1 = 'pull-up' resistors turned on
0 = 'pull-up' resistors turned off

bit 6 **INTEDG** (Interrupt Edge Select bit)
If occurrence of interrupts was enabled, this bit would determine at what edge interrupt on RB0/INT pin would occur.
1 = rising edge
0 = falling edge

bit 5 **T0CS** (TMR0 Clock Source Select bit)
This pin enables a free-run timer to increment its value either from an internal oscillator, i.e. every 1/4 of oscillator clock, or via external impulses on RA4/T0CKI pin.
1 = external impulses
0 = 1/4 internal clock

bit 4 **T0SE** (TMR0 Source Edge Select bit)

If trigger TMR0 was enabled with impulses from a RA4/T0CKI pin, this bit would determine whether it would be on the rising or falling edge of a signal.
1 = falling edge
0 = rising edge

bit 3 **PSA** (Prescaler Assignment bit)
Bit which assigns prescaler between TMR0 and watchdog timer.
1 = prescaler is assigned to watchdog timer.
0 = prescaler is assigned to free timer TMR0

Bit 0:2 **PS0, PS1, PS2** (Prescaler Rate Select bit)
In case of 4MHz oscillator, one instruction cycle (4 internal clocks) lasts 1µs. Numbers in the following table show the time period in µs between incrementing TMR or WDT.

Bits	TMR0	WDT
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

2.14 EEPROM Data Memory

PIC16F84 has 64 bytes of EEPROM memory locations on addresses from 00h to 63h that can be written to or read from. The most important characteristic of this memory is that it does not lose its contents with the loss of power supply. Data can be retained in EEPROM without power supply for up to 40 years (as manufacturer of PIC16F84 microcontroller states), and up to 1 million cycles of writing can be executed.

In practice, EEPROM memory is used for storing important data or process parameters. One such parameter is a given temperature, assigned when setting up a temperature regulator to some process. If that data wasn't retained, it would be necessary to adjust a given temperature after each loss of supply. Since this is very impractical (and even dangerous), manufacturers of microcontrollers have began installing one smaller type of EEPROM memory.

EEPROM memory is placed in a special memory space and can be accessed through special registers. These registers are:

EEDATA Holds read data or that to be written.

EEADR Contains an address of EEPROM location being accessed.

EECON1 Contains control bits.

EECON2 This register does not exist physically and serves to protect EEPROM from accidental writing.

EECON1 register is a control register with five implemented bits. Bits 5, 6 and 7 are not used, and by reading always are zero. Interpretation of EECON1 register bits follows.

2.14.1 EECON1 Register

U-0	U-0	U-0	R/W-1	R/W-1	R/W-x	R/S-0	R/S-x
—	—	—	EEIF (1)	WRERR	WREN	WR	RD
bit 7							bit 0

Legend:
R = Readable bit **W** = Writable bit
U = Unimplemented bit, read as '0' -n = Value at POR reset

bit 4 **EEIF** (EEPROM Write Operation Interrupt Flag bit) Bit used to inform that writing data to EEPROM has ended.
When writing has terminated, this bit would be set automatically. Programmer must clear EEIF bit in his program in order to detect new termination of writing.

1 = writing terminated
0 = writing not terminated yet, or has not started

bit 3 **WRERR** (Write EEPROM Error Flag) Error during writing to EEPROM
This bit was set only in cases when writing to EEPROM had been interrupted by a reset signal or by running out of time in watchdog timer (if activated).
1 = error occurred
0 = error did not occur

bit 2 **WREN** (EEPROM Write Enable bit) Enables writing to EEPROM
If this bit was not set, microcontroller would not allow writing to EEPROM.
1 = writing allowed
0 = writing disallowed

bit 1 **WR** (Write Control bit)
Setting of this bit initializes writing data from EEDATA register to the address specified trough EEADR register.
1 = initializes writing
0 = does not initialize writing

bit 0 **RD** (Read Control bit)
Setting this bit initializes transfer of data from address defined in EEADR to EEDATA register. Since time is not as essential in reading data as in writing, data from EEDATA can already be used further in the next instruction.
1 = initializes reading
0 = does not initialize reading

2.14.2 Reading from EEPROM Memory

Setting the RD bit initializes transfer of data from address found in EEADR register to EEDATA register. As in reading data we don't need so much time as in writing, data taken over from EEDATA register can already be used further in the next instruction.

Sample of the part of a program which reads data in EEPROM, could look something like the following:

```
bcf    STATUS, RP0        ;bank0, because EEADR is at 09h
movlw  0x00               ;address of location being read
movwf  EEADR              ;address transferred to EEADR
bsf    STATUS, RP0        ;bank1 because EECON1 is at 88h
bsf    EECON1, RD         ;reading from EEPROM
bcf    STATUS, RP0        ;Bank0 because EEDATA is at 08h
movf   EEDATA, W          ;W <-- EEDATA
```

After the last program instruction, contents from an EEPROM address zero can be found in working register w.

2.14.3 Writing to EEPROM Memory

In order to write data to EEPROM location, programmer must first write address to EEADR register and data to EEDATA register. Only then is it useful to set WR bit which sets the whole action in motion. WR bit will be reset, and EEIF bit set following a writing what may be used in processing interrupts. Values 55h and AAh are the first and the second key whose disallow for accidental writing to EEPROM to occur. These two values are written to EECON2 which serves only that purpose, to receive these two values and thus prevent any accidental writing to EEPROM memory. Program lines marked as 1, 2, 3, and 4 must be executed in that order in even time intervals. Therefore, it is very important to turn off interrupts which could change the timing needed for executing instructions. After writing, interrupts can be enabled again .

Example of the part of a program which writes data 0xEE to first location in EEPROM memory could look something like the following:

```
        bcf   STATUS, RP0         ;bank0, because EEADR is at 09h
        movlw 0x00                ;address of location being
                                  ;written to
        movwf EEADR               ;address being transferred to
                                  ;EEADR
        movlw 0xEE                ;write the value 0xEE
        movwf EEDATA              ;data goes to EEDATA register
        bsf STATUS, RP0           ;Bank1 because EEADR is at 09h
        bcf INTCON, GIE           ;all interrupts are disabled
        bsf EECON1, WREN          ;writing enabled
        movlw 55h
1)      movwf EECON2              ;first key 55h --> EECON2
2)      movlw AAh
3)      movwf EECON2              ;second key AAh --> EECON2
4)      bsf EECON1,WR             ;initializes writing
        bsf INTCON, GIE           ;interrupts are enabled
```

CHAPTER THREE
MULTITASKING

3.1 Benefits of Multitasking

You can simplify an otherwise complex software application though the use of a multitasking operating system (OS):

- The multitasking and inter-task communications features of the OS allow the complex application to be partitioned into a set of smaller and more manageable programs (or tasks).
- Complex timing and sequencing details can be removed from the application code and become the responsibility of the OS.
- Testability, work breakdown within teams, code reuse, and so on become more manageable.

3.2 Multitasking Concurrency

A conventional microcontroller can only execute a single task at a time—but by rapidly switching between tasks an OS can make it appear as if each task is executing concurrently.

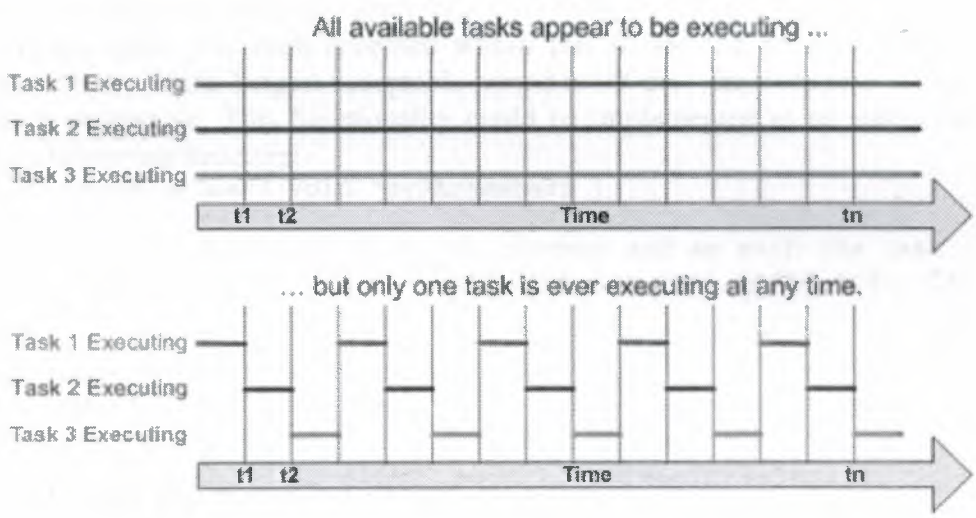


Figure 1: Rapidly switching between tasks can make it appear as if each task is executing concurrently

Figure 1 shows the execution pattern of three tasks with respect to time. The task names are color coded and appear on the left. Time moves from left to right, with the colored lines showing which task is executing at any particular time. The upper diagram demonstrates the perceived concurrent execution pattern, and the lower the actual multitasking execution pattern.

3.3 Task States

In addition to being suspended involuntarily by the RTOS kernel a task can choose to suspend itself. It will do this if it either wants to delay (sleep) for a fixed period, or wait (block) for a resource to become available or an event to occur.

A blocked or sleeping task is not able to execute, and will not be allocated any processing time.

3.4Scheduling

The scheduling policy is the algorithm used by the OS to decide which task should be executing at any moment in time. The scheduling policy is designed to meet the objectives of the OS—which for an RTOS is to provide a timely response to real world events.

The application designer must assign a priority to each task. The higher the criticality of the task (or the shorter its maximum acceptable response time) the higher its relative priority should be. The scheduling policy of the RTOS is then simply to make sure the highest priority task that is ready to execute (not blocked or sleeping) is the task given processing time.

Example Real Time Execution Profile

This section provides a simplistic example that demonstrates the principles of real time scheduling.

A hypothetical embedded system incorporates a keypad and LCD. A user must receive the visual feedback of each key press within a reasonable period—if the user cannot see that the key press has been accepted within this period the product will at best be awkward to use. If the longest acceptable period is 100ms—any response between 0 and 100ms is acceptable. This functionality could be implemented as an autonomous task with the following structure:

```
void vKeyHandlerTask( void *pvParameters )
{
    /* Key handling is a continuous process and as such the task
    is implemented using an infinite loop (as most tasks are). */
    for( ;; )
    {
        [Suspend waiting for a key press]
        [Process the key press]
    }
}
```

Listing 1: Task that records key strokes

Now assume the software is also performing a control function that relies on a digitally filtered input. The input must be sampled, filtered, and the control cycle executed every 2ms. For correct operation of the filter the temporal regularity of the sample must be accurate to 0.5ms. This functionality could be implemented as an autonomous task with the following structure:

```
void vControlTask( void *pvParameters )
{
    for( ;; )
    {
        [Suspend waiting for 2ms since the start of the previous
        cycle]
```

```
        [Sample the input]
        [Filter the sampled input]
        [Perform control algorithm]
        [Output result]
    }
}
```
Listing 2: Sampling the digitally filtered input

The software engineer must assign the control task the highest priority as:
The deadline for the control task is stricter than that of the key handling task.
The consequence of a missed deadline is greater for the control task than for the key handler task.

Figure 2 demonstrates how these tasks would be scheduled by a real time operating system. The RTOS has itself created a task—the idle task—which will execute only when there are no other tasks able to do so. The idle task is always in a state where it is able to execute.

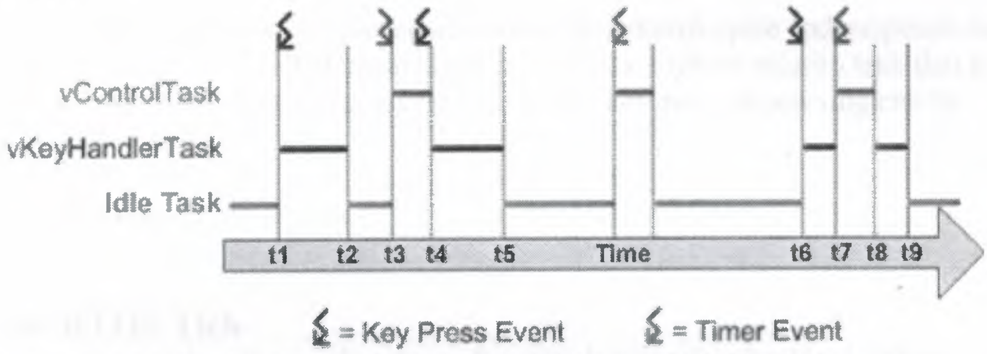


Figure 2: Execution profile of the example tasks

Referring to **Figure 2**:

- At the start neither of our two tasks are able to run—vControlTask is waiting for the correct time to start a new control cycle and vKeyHandlerTask is waiting for a key to be pressed. Processing time is given to the idle task.

- At time t1, a key press occurs. vKeyHandlerTask is now able to execute—it has a higher priority than the idle task so is given processing time.

- At time t2 vKeyHandlerTask has completed processing the key and updating the LCD. It cannot continue until another key has been pressed so suspends itself and the idle task is again resumed.

- At time t3 a timer event indicates that it is time to perform the next control cycle. vControlTask can now execute and as the highest priority task is scheduled processing time immediately.

- Between time t3 and t4, while vControlTask is still executing, a key press occurs. vKeyHandlerTask is now able to execute, but as it has a lower priority than vControlTask it is not scheduled any processing time.

- At t4 vControlTask completes processing the control cycle and cannot restart until the next timer event—it suspends itself. vKeyHandlerTask is now the task with the highest priority that is able to run so is scheduled processing time in order to process the previous key press.

- At t5 the key press has been processed, and vKeyHandlerTask suspends itself to wait for the next key event. Again neither of our tasks are able to execute and the idle task is scheduled processing time.

- Between t5 and t6 a timer event is processed, but no further key presses occur.

- The next key press occurs at time t6, but before vKeyHandlerTask has completed processing the key a timer event occurs. Now both tasks are able to execute. As vControlTask has the higher priority vKeyHandlerTask is suspended before it has completed processing the key, and vControlTask is scheduled processing time.

- At t8 vControlTask completes processing the control cycle and suspends itself to wait for the next. vKeyHandlerTask is again the highest priority task that is able to run so is scheduled processing time so the key press processing can be completed.

Implementation Building Blocks

3.5 The RTOS Tick

When sleeping a task will specify a time after which it requires 'waking'. When blocking a task can specify a maximum time it wishes to wait.

The FreeRTOS.org kernel measures time using a tick count variable. A timer interrupt (the RTOS tick interrupt) increments the tick count with strict temporal accuracy—allowing time to be measured to a resolution of the chosen timer interrupt frequency. Each time the tick count is incremented the RTOS kernel must check to see if it is now time to unblock or wake a task.
It is possible that a task woken or unblocked during the tick ISR will have a priority higher than that of the interrupted task. If this is the case the tick ISR should return to the newly woken/unblocked task—effectively interrupting one task but returning to another (**Figure 3**).

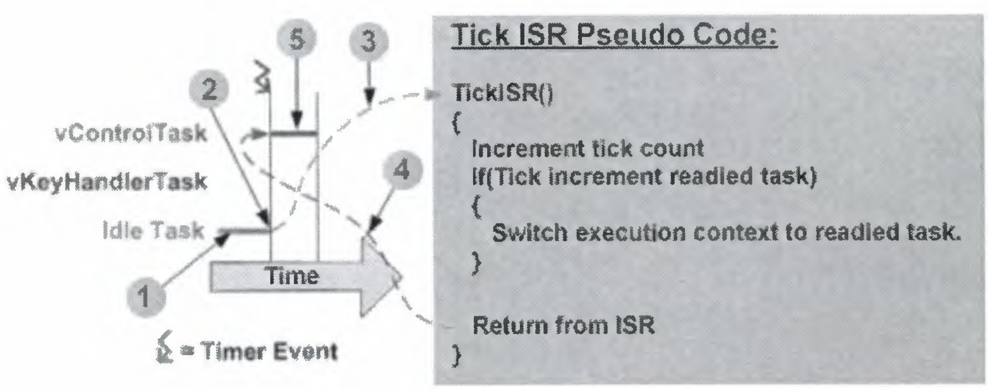


Figure 3: A context switch occurring in an interrupt service routine

Referring to the circled numbers in **Figure 3**:

- At (1) the highest priority task (vControlTask) is blocked waiting for a timer to expire. The next highest priority task (vKeyHandlerTask) is also blocked waiting for a key press event. This leaves the Idle Task as the highest priority task that is able to run.

- At (2) the RTOS tick interrupt occurs. The microcontroller stops executing the Idle Task and starts executing the tick ISR (3).

- The tick ISR increments the tick count which (for the sake of this example) makes vControlTask ready to run. vControlTask has a higher priority than the idle task so a context switch is required. A task switch from the Idle Task to vControlTask occurs within the ISR.

- As the execution context is now that of vControlTask, exiting the ISR (4) returns control to vControlTask, which starts executing (5). The Idle Task remains suspended until it is again the highest priority task that is able to execute.

3.6 "Execution Context"—a Definition

As a task executes it utilizes microcontroller registers and accesses RAM and ROM just as any other program. These resources together (the registers, stack, and so on) comprise the task execution context.

A task is a sequential piece of code that does not know when it is going to get suspended (stopped from executing) or resumed (given more processing time) by the RTOS and does not even know when this has happened. Consider the example of a task being suspended immediately before executing an instruction that sums the values contained within two registers.

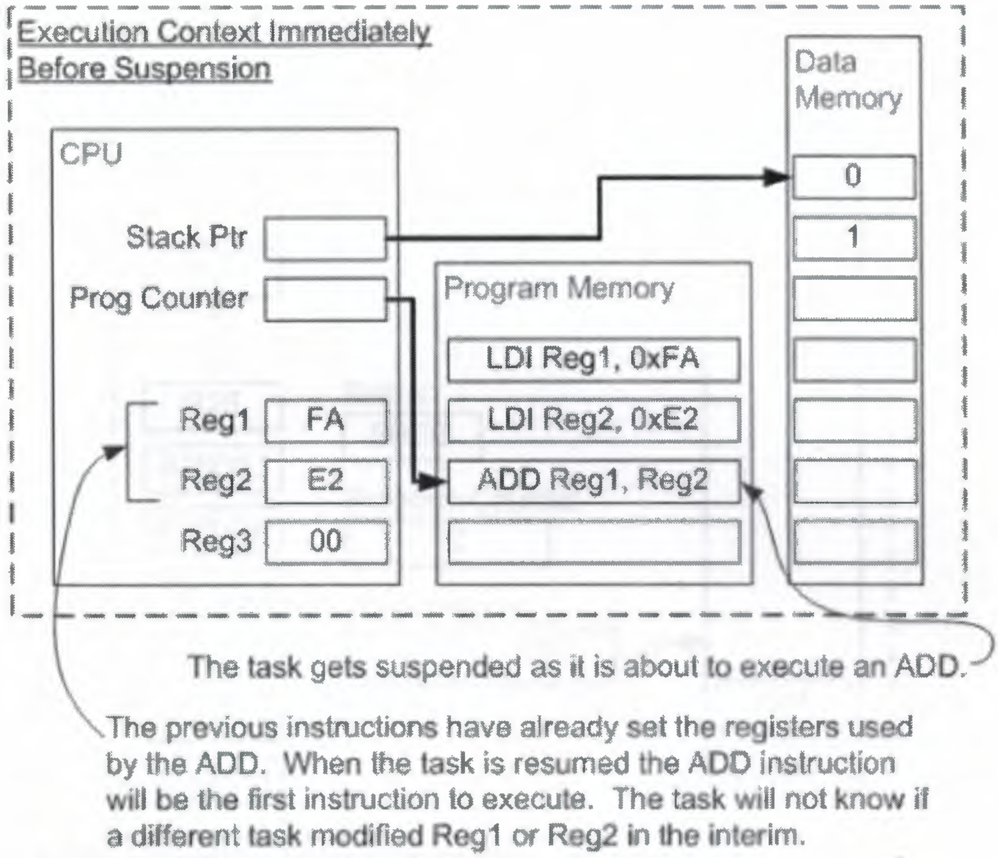


Figure 4: A sample task context immediately prior to the task being suspended
While the task is suspended other tasks will execute and may modify the register values. Upon resumption the task will not know that the registers have been altered—if it used the modified values the summation would result in an incorrect value.
To prevent this type of error it is essential that upon resumption a task has a context identical to that immediately prior to its suspension. The RTOS kernel is responsible for ensuring this is the case—and does so by saving the context of a task as it is suspended. When the task is resumed its saved context is restored by the RTOS kernel prior to its execution. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

The AVR Context

On the AVR microcontroller, the context consists of:

- **32 General Purpose Registers**
  The GCC compiler assumes register R1 is set to zero.

- **Status Register**
  The value of the status register affects instruction execution, and must be preserved across context switches.

- **Program Counter**
  Upon resumption, a task must continue execution from the instruction that was about to be executed immediately prior to its suspension.

- **The Two Stack Pointer Registers.**

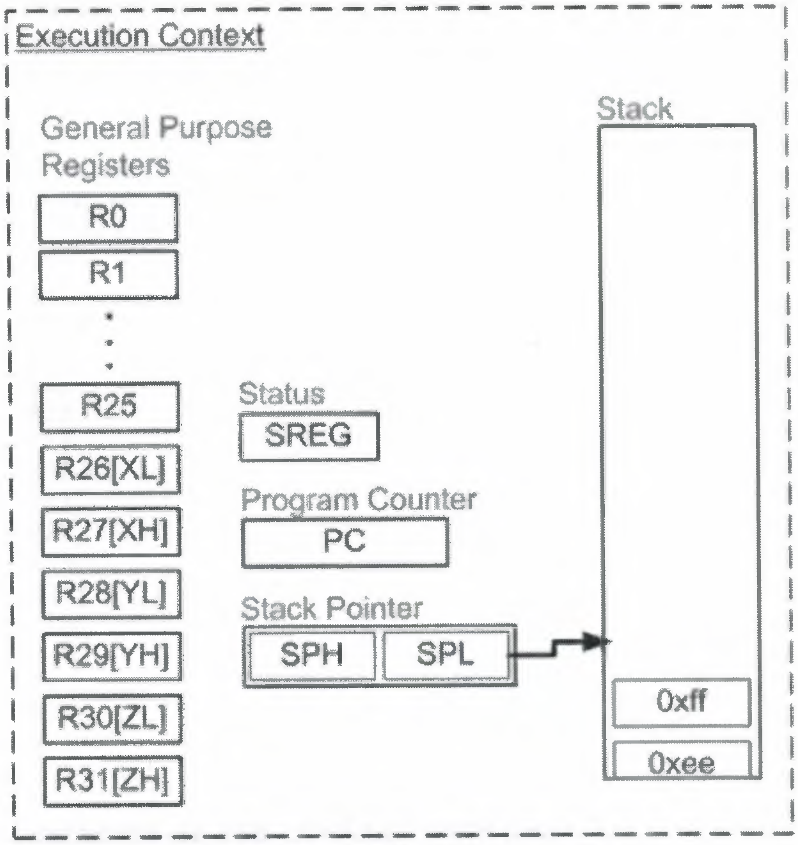


Figure 5: The execution context on the AVR microcontroller

Writing the Tick ISR—The GCC 'signal' Attribute

The MegaAVR port of FreeRTOS.org generates the periodic tick interrupt from a compare match event on the MegaAVR timer 1 peripheral. GCC allows the tick ISR function to be written in C by using the following syntax.

```
void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal ) );
void SIG_OUTPUT_COMPARE1A( void )
{
   /* ISR C code for RTOS tick. */
   vPortYieldFromTick();
}
```

Listing 3: C code for compare match ISR

The '__attribute__ ((signal))' directive on the function prototype informs the compiler that the function is an ISR and results in two important changes in the compiler output:

1. The 'signal' attribute ensures that every AVR register that gets modified during the ISR is restored to its original value when the ISR exits. This is required as the compiler cannot make any assumptions as to when the interrupt will execute, and therefore cannot optimize which registers require saving and which don't.

2. The 'signal' attribute also forces a 'return from interrupt' instruction (RETI) to be used in place of the 'return' instruction (RET) that would otherwise be used. The AVR disables interrupts upon entering an ISR and the RETI instruction is required to re-enable them on exiting.

Compiling the ISR results in the following output:

```
;void SIG_OUTPUT_COMPARE1A( void )
;{
    ; ---------------------------------------
    ; CODE GENERATED BY THE COMPILER TO SAVE
    ; THE REGISTERS THAT GET ALTERED BY THE
    ; APPLICATION CODE DURING THE ISR.
    PUSH    R1
    PUSH    R0
    IN      R0,0x3F
    PUSH    R0
    CLR     R1
    PUSH    R18
    PUSH    R19
    PUSH    R20
    PUSH    R21
    PUSH    R22
    PUSH    R23
    PUSH    R24
    PUSH    R25
    PUSH    R26
    PUSH    R27
    PUSH    R30
    PUSH    R31

    ; ---------------------------------------
    ; CODE GENERATED BY THE COMPILER FROM THE
    ; APPLICATION C CODE.
    ;vTaskIncrementTick();
    CALL        0x0000029B      ;Call subroutine

    ; ---------------------------------------
    ; CODE GENERATED BY THE COMPILER TO
    ; RESTORE THE REGISTERS PREVIOUSLY
    ; SAVED.
    POP     R31
    POP     R30
    POP     R27
    POP     R26
    POP     R25
    POP     R24
    POP     R23
    POP     R22
    POP     R21
    POP     R20
    POP     R19
    POP     R18
    POP     R0
    OUT     0x3F,R0
    POP     R0
    POP     R1
    RETI
;}
```

Listing 4: Compiler output for Listing 3

Organizing the Context—The GCC 'naked' Attribute

The previous section showed how you can use the 'signal' attribute to write an ISR in C and how this results in part of the execution context being automatically saved (only the microcontroller registers modified by the ISR get saved). Performing a context switch however requires the entire context to be saved.

The application code could explicitly save all the registers on entering the ISR, but doing so would result in some registers being saved twice—once by the compiler generated code and then again by the application code. This is undesirable and can be avoided by using the 'naked' attribute in addition to the 'signal' attribute.

```
void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal, naked ) );
void SIG_OUTPUT_COMPARE1A( void )
{
    /* ISR C code for RTOS tick. */
    vPortYieldFromTick();
}
```

Listing 5: Addition of the 'naked' attribute to the compare match ISR

The 'naked' attribute prevents the compiler generating any function entry or exit code. Now, compiling the ISR results in the much simpler output:

```
;void SIG_OUTPUT_COMPARE1A( void )
;{
    ; ---------------------------------------
    ; NO COMPILER GENERATED CODE HERE TO SAVE
    ; THE REGISTERS THAT GET ALTERED BY THE
    ; ISR.
    ; ---------------------------------------
    ; CODE GENERATED BY THE COMPILER FROM THE
    ; APPLICATION C CODE.
    ;vTaskIncrementTick();
    CALL     0x0000029B     ;Call subroutine

    ; ---------------------------------------
    ; NO COMPILER GENERATED CODE HERE TO RESTORE
    ; THE REGISTERS OR RETURN FROM THE ISR.
    ; ---------------------------------------
;}
```

Listing 6: Compiler output from Listing 5

When the 'naked' attribute is used the compiler does not generate any function entry or exit code so this must now be added explicitly. The macros portSAVE_CONTEXT() and portRESTORE_CONTEXT() respectively save and restore the entire execution context.

```
void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal, naked ) );
void SIG_OUTPUT_COMPARE1A( void )
{
    /* Macro that explicitly saves the execution
    context. */
    portSAVE_CONTEXT();

    /* ISR C code for RTOS tick. */
    vPortYieldFromTick();

    /* Macro that explicitly restores the
    execution context. */
    portRESTORE_CONTEXT();

    /* The return from interrupt call must also
    be explicitly added. */
    asm volatile ( "reti" );
}
```

Listing 7: The compare match ISR modified to explicitly save/restore the execution context

The 'naked' attribute gives the application code complete control over when and how the AVR context is saved. If the application code saves the entire context on entering the ISR there is no need to save it again before performing a context switch so none of the microcontroller registers get saved twice.

Saving the Context

Each task has its own stack memory area so the context can be saved by simply pushing registers onto the task stack. Saving the AVR context is one place where assembly code is unavoidable.

`portSAVE_CONTEXT()` is implemented as a macro, the source for which is given below:

```
#define portSAVE_CONTEXT()                  \
asm volatile (                              \
  "push r0                      \n\t" \ (1)
  "in   r0, __SREG__            \n\t" \ (2)
  "cli                          \n\t" \ (3)
  "push r0                      \n\t" \ (4)
  "push r1                      \n\t" \ (5)
  "clr  r1                      \n\t" \ (6)
  "push r2                      \n\t" \ (7)
  "push r3                      \n\t" \
  "push r4                      \n\t" \
  "push r5                      \n\t" \
    :
    :
    :
  "push r30                     \n\t" \
  "push r31                     \n\t" \
  "lds  r26, pxCurrentTCB       \n\t" \ (8)
  "lds  r27, pxCurrentTCB + 1   \n\t" \ (9)
  "in   r0, __SP_L__            \n\t" \ (10)
  "st   x+, r0                  \n\t" \ (11)
  "in   r0, __SP_H__            \n\t" \ (12)
  "st   x+, r0                  \n\t" \ (13)
);
```

Listing 8: `portSAVE_CONTEXT()`

Referring to the numbers in Listing 8:

- Register R0 is saved first (1) as it is used when the status register is saved, and must be saved with its original value.
- The status register is moved into R0 (2) so it can be saved onto the stack (4).
- Interrupts are disabled (3). If `portSAVE_CONTEXT()` was only called from within an ISR there would be no need to explicitly disable interrupts as the AVR will have already done so. As the `portSAVE_CONTEXT()` macro is also used outside

of interrupt service routines (when a task suspends itself) interrupts must be explicitly cleared as early as possible.

- The code generated by the compiler from the ISR C code assumes R1 is set to zero. The original value of R1 is saved (5) before R1 is cleared (6).
- Between (7) and (8) all remaining microcontroller registers are saved in numerical order.
- The stack of the task being suspended now contains a copy of the tasks execution context. The kernel stores the tasks stack pointer so the context can be retrieved and restored when the task is resumed. The x register is loaded with the address to which the stack pointer is to be saved (8 and 9).
- The stack pointer is saved, first the low byte (10 and 11), then the high nibble (12 and 13).

Restoring the Context

`portRESTORE_CONTEXT()` is the reverse of `portSAVE_CONTEXT()`.

The context of the task being resumed was previously stored in the tasks stack. The kernel retrieves the stack pointer for the task then POP's the context back into the correct microcontroller registers.

```
#define portRESTORE_CONTEXT()         \
asm volatile (                        \
   "lds r26, pxCurrentTCB      \n\t" \  (1)
   "lds r27, pxCurrentTCB + 1 \n\t" \  (2)
   "ld  r28, x+                \n\t" \
   "out __SP_L__, r28          \n\t" \  (3)
   "ld  r29, x+                \n\t" \
   "out __SP_H__, r29          \n\t" \  (4)
   "pop r31                    \n\t" \
   "pop r30                    \n\t" \
     :
     :
     :
   "pop r1                     \n\t" \
   "pop r0                     \n\t" \  (5)
   "out __SREG__, r0           \n\t" \  (6)
   "pop r0                     \n\t" \  (7)
);
```

Listing 9: `portRESTORE_CONTEXT()`

Referring to the numbers in Listing 9:

- `pxCurrentTCB` holds the address from where the tasks stack pointer can be retrieved. This is loaded into the X register (1 and 2).

- The stack pointer for the task being resumed is loaded into the AVR stack pointer, first the low byte (3), then the high nibble (4).

- The microcontroller registers are then popped from the stack in reverse numerical order, down to R1.

- The status register stored on the stack between registers R1 and R0, so is restored (6) before R0 (7).

implementation of the RTOS tick is therefore (see the comments within the code for further details):

```
void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal, naked ) );
void vPortYieldFromTick( void ) __attribute__ ( ( naked ) );
/*-----------------------------------------------*/

/* Interrupt service routine for the RTOS tick. */
void SIG_OUTPUT_COMPARE1A( void )
{
  /* Call the tick function. */
  vPortYieldFromTick();

  /* Return from the interrupt. If a context
  switch has occurred this will return to a
  different task. */
  asm volatile ( "reti" );
}

/*-----------------------------------------------*/
void vPortYieldFromTick( void )
{
  /* This is a naked function so the context
  is saved. */
  portSAVE_CONTEXT();

  /* Increment the tick count and check to see
  if the new tick value has caused a delay
  period to expire. This function call can
  cause a task to become ready to run. */
  vTaskIncrementTick();

  /* See if a context switch is required.
  Switch to the context of a task made ready
  to run by vTaskIncrementTick() if it has a
  priority higher than the interrupted task. */
  vTaskSwitchContext();

  /* Restore the context. If a context switch
  has occurred this will restore the context of
  the task being resumed. */
```

```
  portRESTORE_CONTEXT();

  /* Return from this naked function. */
  asm volatile ( "ret" );
}
```

CONCLUSION

Driven both by the economics of super-integration and the performance requirements of next generation electronic products, system designers are seeking new approaches to the development of "systems-on-a-chip." The common element in many of these efforts is the rapid convergence of microcontroller particularly in the field of embedded system design.

This project is research for those interested in the microcontrollers . I have information about microcontrollers and microprocessors.There are many course are not covered, there are simply too many. This knowledges gives very specially view angle to me. I think to use this knowledges real life.

REFERENCES

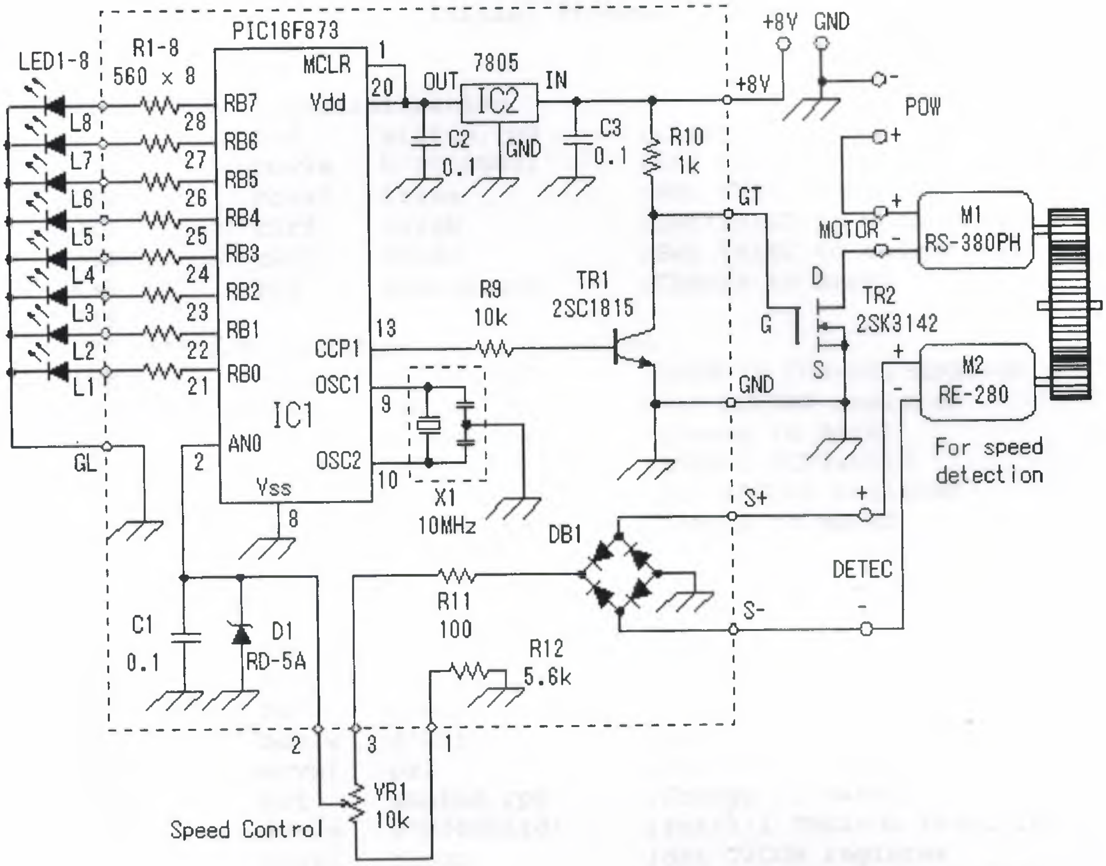
1. http:\ \ www.techonline.com

2. http: \ \ www.microelectronica.co.uk

3. Other internet sites.

APPENDIX A

DC Motor Speed Controller



Source Code file for DC Motor Speed Controller

```
;***************************************************************
;
;                  DC motor speed controller
;
;                                      Device : PIC16F873
;                                      Author : Seiichi Inoue
;***************************************************************

        list            p=pic16f873
        include         p16f873.inc
        __config _hs_osc & _wdt_off & _pwrte_on & _lvp_off
        errorlevel      -302    ;Suppress bank warning

;****************  Label Definition  ********************
speed   equu    d'8'    ;Reference speed (5x8/256=0.156V)
change  equ     d'1'     ;Change value (2mV/ms)

led     equ     h'20'    ;LED control data save area

;****************  Program Start  ***********************
```

```
        org     0               ;Reset Vector
        goto    init
        org     4               ;Interrupt Vector
        goto    int

;****************  Initial Process  ********************
init

;*** Port initialization
        bsf     status,rp0      ;Change to Bank1
        movlw   b'00000001'     ;AN0 to input mode
        movwf   trisa           ;Set TRISA register
        clrf    trisb           ;Set TRISB to uotput mode
        clrf    trisc           ;Set TRISC to output mode
        bcf     status,rp0      ;Change to Bank0

;*** A/D converter initialization
        movlw   b'10000001'     ;ADCS=10 CHS=AN0 ADON=ON
        movwf   adcon0          ;Set ADCON0 register
        bsf     status,rp0      ;Change to Bank1
        movlw   b'00001110'     ;ADFM=0 PCFG=1110
        movwf   adcon1          ;Set ADCON1 register
        bcf     status,rp0      ;Change to Bank0

;*** PWM initialization
        clrf    tmr2            ;Clear TMR2 register
        movlw   b'11111111'     ;Max duty (low speed)
        movwf   ccpr1l          ;Set CCPR1L register
        bsf     status,rp0      ;Change to Bank1
        movlw   d'255'          ;Period=1638.4usec(610Hz)
        movwf   pr2             ;Set PR2 register
        bcf     status,rp0      ;Change to Bank0
        movlw   b'00000110'     ;Pst=1:1 TMR2=ON Pre=1:16
        movwf   t2con           ;Set T2CON register
        movlw   b'00001100'     ;CCP1XY=0 CCP1M=1100(PWM)
        movwf   ccp1con         ;Set CCP1CON register

;*** Compare mode initialization
        clrf    tmr1h           ;Clear TMR1H register
        clrf    tmr1l           ;Clear TMR1L register
        movlw   h'61'           ;H'61A8'=25000
        movwf   ccpr2h          ;Set CCPR2H register
        movlw   h'a8'           ;25000*0.4usec = 10msec
        movwf   ccpr2l          ;Set CCPR2L register
        movlw   b'00000001'     ;Pre=1:1 TMR1=Int TMR1=ON
        movwf   t1con           ;Set T1CON register
        movlw   b'00001011'     ;CCP2M=1011(Compare)
        movwf   ccp2con         ;Set CCP2CON register

;*** Interruption control
        bsf     status,rp0      ;Change to Bank1
        movlw   b'00000001'     ;CCP2IE=Enable
        movwf   pie2            ;Set PIE2 register
        bcf     status,rp0      ;Change to Bank0
        movlw   b'11000000'     ;GIE=ON PEIE=ON
        movwf   intcon          ;Set INTCON register

wait
        goto    $               ;Interruption wait

```

```
;***************  Interruption Process  ****************
int
        clrf    pir2            ;Clear interruption flag
ad_check
        btfsc   adcon0,go       ;A/D convert end ?
        goto    ad_check        ;No. Again
        movfw   adresh          ;Read ADRESH register
        sublw   speed           ;Ref speed - Detect speed
        btfsc   status,c        ;Reference < Detect ?
        goto    check1          ;No. Jump to > or = check

;--- control to low speed ---
        movfw   ccpr1l          ;Read CCPR1L register
        addlw   change          ;Change value + CCPR1L
        btfss   status,c        ;Overflow ?
        movwf   ccpr1l          ;No. Write CCPR1L
        goto    led_cont        ;Jump to LED control

check1
        btfsc   status,z        ;Reference = Detect ?
        goto    led_cont        ;Yes. Jump to LED control

;--- control to fast speed ---
        movlw   change          ;Set change value
        subwf   ccpr1l,f        ;CCPR1L - Change value
        btfsc   status,c        ;Underflow ?
        goto    led_cont        ;Jump to LED control
        clrf    ccpr1l          ;Set fastest speed

;****************  LED control Process  *****************
led_cont
        comf    ccpr1l,w        ;Complement CCPR1L bit
        movwf   led             ;Save LED data
        movlw   b'00010000'     ;Set compare data
        subwf   led,w           ;LED - data
        btfsc   status,c        ;Under ?
        goto    led1            ;No.
        movlw   b'00000000'     ;Set LED control data
        goto    int_end         ;Jump to interrupt end
led1    movlw   b'00100000'     ;Set compare data
        subwf   led,w           ;LED - data
        btfsc   status,c        ;Under ?
        goto    led2            ;No.
        movlw   b'00000001'     ;Set LED control data
        goto    int_end         ;Jump to interrupt end
led2    movlw   b'01000000'     ;Set compare data
        subwf   led,w           ;LED - data
        btfsc   status,c        ;Under ?
        goto    led3            ;No.
        movlw   b'00000011'     ;Set LED control data
        goto    int_end         ;Jump to interrupt end
led3    movlw   b'01100000'     ;Set compare data
        subwf   led,w           ;LED - data
        btfsc   status,c        ;Under ?
        goto    led4            ;No.
        movlw   b'00000111'     ;Set LED control data
        goto    int_end         ;Jump to interrupt end
led4    movlw   b'10000000'     ;Set compare data
        subwf   led,w           ;LED - data
        btfsc   status,c        ;Under ?
```

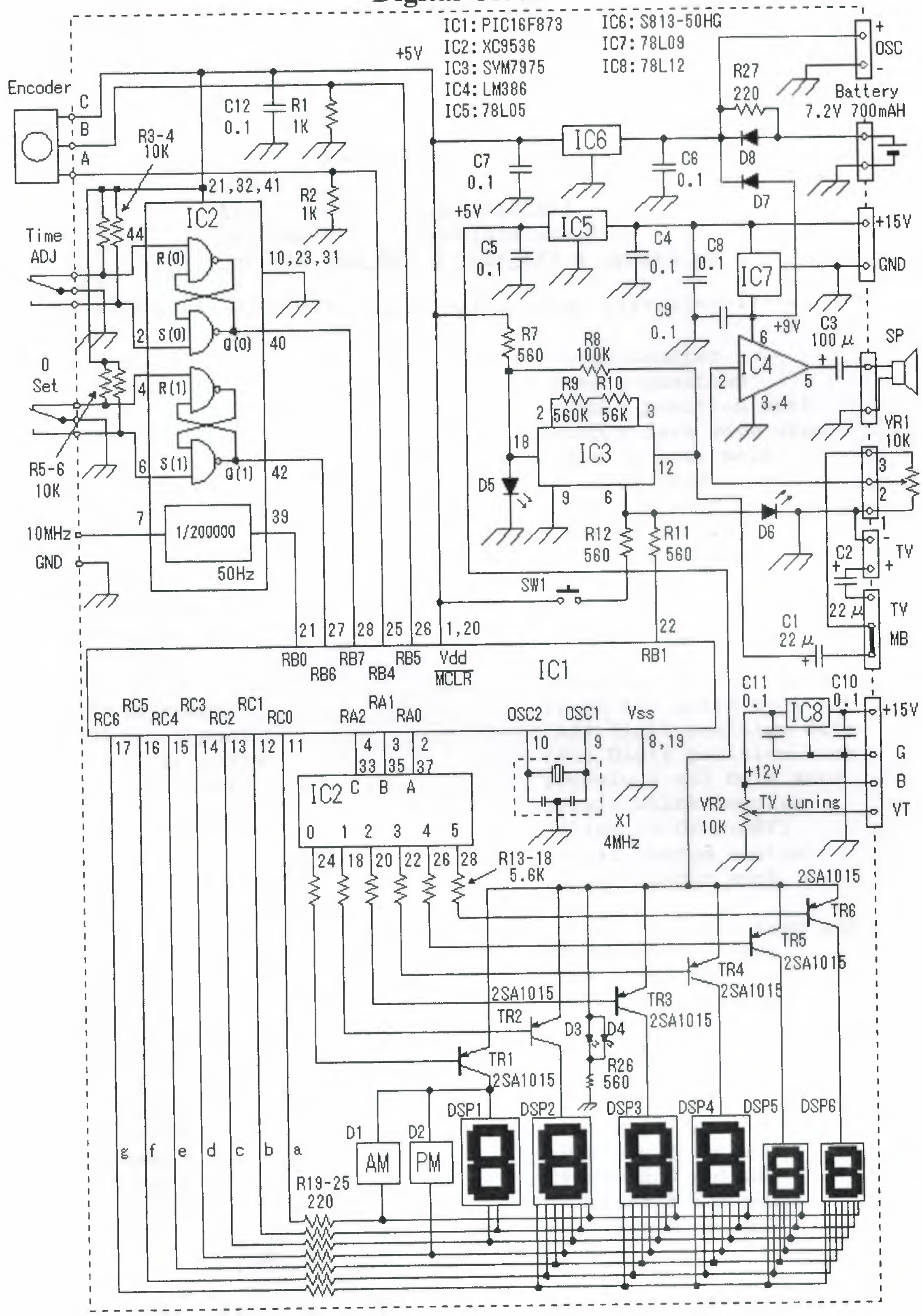
```
        goto    led5            ;No.
        movlw   b'00001111'     ;Set LED control data
        goto    int_end         ;Jump to interrupt end
led5    movlw   b'10100000'     ;Set compare data
        subwf   led,w           ;LED - data
        btfsc   status,c        ;Under ?
        goto    led6            ;No.
        movlw   b'00011111'     ;Set LED control data
        goto    int_end         ;Jump to interrupt end
led6    movlw   b'11000000'     ;Set compare data
        subwf   led,w           ;LED - data
        btfsc   status,c        ;Under ?
        goto    led7            ;No.
        movlw   b'00111111'     ;Set LED control data
        goto    int_end         ;Jump to interrupt end
led7    movlw   b'11100000'     ;Set compare data
        subwf   led,w           ;LED - data
        btfsc   status,c        ;Under ?
        goto    led8            ;No.
        movlw   b'01111111'     ;Set LED control data
        goto    int_end         ;Jump to interrupt end
led8    movlw   b'11111111'     ;Set LED control data

;************  END of Interruption Process *************
int_end
        movwf   portb           ;Set PROTB
        retfie

;*******************************************************
;               END of DC motor speed controller
;*******************************************************

        end
```

HEX CODE

```
:020000000528D1
:080008002D2883160130085004C
:10001000860187018312813O9F0083160E309F0076
```

APPENDIX B

Digital Clock



Source Code file of Digital Clock

```
;**********************************************************
;
;                         Digital Clock
;
;                                    Device : PIC16F873
;                                    Author : Seiichi Inoue
;**********************************************************

        list            p=pic16f873
        include         p16f873.inc
        __config _hs_osc & _wdt_off & _pwrte_on & _lvp_off

;****************  Label Definition  ******************
        cblock  h'20'
count                                   ;Clock counter
disp_p                                  ;Disply position
disp_pw                                 ;Disply position work
disp_data                               ;Disply data save area
disp_h10w                               ;Tens of hour work
disp_h10                                ;Tens of hour
disp_h1                                 ;Units of hour
disp_m10                                ;Tens of minute
disp_m1                                 ;Units of minute
disp_s10                                ;Tens of second
disp_s1                                 ;Units of second
mode                                    ;Mode (0:Adjust 1:Clock)
rb6ll                                   ;0 sec adjust Last look
rb7ll                                   ;Time adjust Last look
rb7count                                ;Time adj guard counter
digit_posi                              ;Adj digit position data
digit_posiw                             ;Adj digit position work
digit_save                              ;Previous adj data save
digit_blink                             ;Digit blink counter
blink_cont                              ;Blink (0:ON 1:OFF)
change_st                               ;Digit change status
change_wk                               ;Digit change work

seg7_ha                                 ;7 segLED table head adr
seg70                                   ;Pattern 0 set adr
seg71                                   ;Pattern 1 set adr
seg72                                   ;Pattern 2 set adr
seg73                                   ;Pattern 3 set adr
seg74                                   ;Pattern 4 set adr
seg75                                   ;Pattern 5 set adr
seg76                                   ;Pattern 6 set adr
seg77                                   ;Pattern 7 set adr
seg78                                   ;Pattern 8 set adr
seg79                                   ;Pattern 9 set adr
seg7a                                   ;Pattern A set adr
seg7b                                   ;Pattern B set adr
        endc

ra1     equ     h'01'                   ;RA1 port designation
ra2     equ     h'02'                   ;RA2 port designation
ra3     equ     h'03'                   ;RA3 port designation
rb1     equ     h'01'                   ;RB1 port designation
rb4     equ     h'04'                   ;RB4 port designation
```

```
rb5      equ     h'05'          ;RB5 port designation
rb6      equ     h'06'          ;RB6 port designation
rb7      equ     h'07'          ;RB7 port designation

seg7_0   equ     b'01000000'    ;-gfedcba Pattern 0
seg7_1   equ     b'01111001'    ;         Pattern 1
seg7_2   equ     b'00100100'    ;         Pattern 2
seg7_3   equ     b'00110000'    ;         Pattern 3
seg7_4   equ     b'00011001'    ;         Pattern 4
seg7_5   equ     b'00010010'    ;         Pattern 5
seg7_6   equ     b'00000010'    ;         Pattern 6
seg7_7   equ     b'01111000'    ;         Pattern 7
seg7_8   equ     b'00000000'    ;         Pattern 8
seg7_9   equ     b'00010000'    ;         Pattern 9
seg7_a   equ     b'01111111'    ;         LED off
seg7_b   equ     b'00100011'    ;         Illegal int

;****************  Program Start  *********************
         org     0              ;Reset Vector
         goto    init
         org     4              ;Interrupt Vector
         goto    int

;****************  Initial Process  ********************
init

;*** Port mode initializing
         bsf     status,rp0     ;Change to Bank1
         movlw   b'00000110'    ;RA port to digital mode
         movwf   adcon1         ;Set ADCON1 register
         movlw   b'00000000'    ;RA port to output mode
         movwf   trisa          ;Set TRISA register
         movlw   b'11111101'    ;RB1:output,OTHER:input
         movwf   trisb          ;Set TRISB register
         movlw   b'00000000'    ;RC port to output mode
         movwf   trisc          ;Set TRISC register

;*** LED disply interval initializing (Timer0)
         movlw   b'00000010'    ;PBPU=on,PSA=0,PS=1:8
         movwf   option_reg     ;Set OPTION_REG register
         bcf     status,rp0     ;Change to Bank0
         movlw   d'131'         ;(256-131)x8=1000usec
         movwf   tmr0           ;Set TMR0 register

;*** Port initializing
         clrf    porta          ;Clear PORTA
         clrf    portb          ;Clear PORTB
         movlw   b'11111111'    ;Set LED off data
         movwf   portc          ;Set PORTC

;*** Work area initializing
         clrf    count          ;Clear Clock counter
         movlw   d'6'           ;Disply position = 6
         movwf   disp_p         ;Set disply position
         clrf    disp_h10       ;Clear Tens of hour
         clrf    disp_h1        ;Clear Units of hour
         clrf    disp_m10       ;Clear Tens of minute
         clrf    disp_m1        ;Clear Units of minute
         clrf    disp_s10       ;Clear Tens of second
         clrf    disp_s1        ;Clear Units of second
```

```
        clrf    mode            ;Set Adjust mode
        clrf    rb6ll           ;Clear 0 sec Last look
        clrf    rb7ll           ;Clear Time adj Last look
        clrf    rb7count        ;Clear Time adj guard
        clrf    digit_posi      ;Clear Adj digit position
        clrf    digit_blink     ;Clear Digit blink count
        clrf    blink_cont      ;Set Blink on
        clrf    change_st       ;Clear Change status

        movlw   seg70           ;Set 7seg head address
        movwf   seg7_ha         ;Save 7seg head address
        movlw   seg7_0          ;Set 7segment pattern 0
        movwf   seg70           ;Save pattern 0
        movlw   seg7_1          ;Set 7segment pattern 1
        movwf   seg71           ;Save pattern 1
        movlw   seg7_2          ;Set 7segment pattern 2
        movwf   seg72           ;Save pattern 2
        movlw   seg7_3          ;Set 7segment pattern 3
        movwf   seg73           ;Save pattern 3
        movlw   seg7_4          ;Set 7segment pattern 4
        movwf   seg74           ;Save pattern 4
        movlw   seg7_5          ;Set 7segment pattern 5
        movwf   seg75           ;Save pattern 5
        movlw   seg7_6          ;Set 7segment pattern 6
        movwf   seg76           ;Save pattern 6
        movlw   seg7_7          ;Set 7segment pattern 7
        movwf   seg77           ;Save pattern 7
        movlw   seg7_8          ;Set 7segment pattern 8
        movwf   seg78           ;Save pattern 8
        movlw   seg7_9          ;Set 7segment pattern 9
        movwf   seg79           ;Save pattern 9
        movlw   seg7_a          ;Set 7segment pattern A
        movwf   seg7a           ;Save pattern A
        movlw   seg7_b          ;Set 7segment pattern B
        movwf   seg7b           ;Save pattern B

;*** Interruption control
        movlw   b'10111000'     ;GIE&T0IE&INTE&RBIE=ON
        movwf   intcon          ;Set INTCON register

wait
        goto    $               ;Interruption wait

;***************  Interruption Process  *****************
int
        movf    intcon,w        ;Read INTCON register
        btfsc   intcon,intf     ;RB0/INT interrupt ?
        goto    clock           ;Yes. "Clock"
        btfsc   intcon,t0if     ;TMR0 overflow ?
        goto    led_disp        ;Yes. "LED disply"
        btfsc   intcon,rbif     ;RB Port Change ?
        goto    digit_change    ;Yes. "Digit change"

;***************  Illegal interruption  *****************
illegal
        movlw   h'0b'           ;Set Illegal disp digit
        addwf   seg7_ha,w       ;Seg7 H.Adr + digit
        movwf   fsr             ;Set FSR register
        movf    indf,w          ;Read seg7 data
        movwf   portc           ;Set LED data
```

```
        movlw   b'00000101'      ;Set sec1 select data
        movwf   porta            ;Write digit select data
        goto    $                ;Stop

;************  END of Interruption Process **************
int_end
        retfie

;*********  LED disply Process (1msec interval) *********
led_disp
        bcf     intcon,t0if      ;Clear T0IF
        movlw   d'131'           ;Set Time value (1msec)
        movwf   tmr0             ;Write TMR0 register
        movlw   b'11111111'      ;LED off data
        movwf   portc            ;Clear disply
        movf    disp_p,w         ;Read disply position
        movwf   disp_pw          ;Save position data
        decfsz  disp_pw,f        ;Units of second ?
        goto    led_disp0        ;No. Next

;*** Control UNITS of SECOND
        movlw   b'00000101'      ;Set units of second
        movwf   porta            ;Write PORTA register
        movf    disp_s1,w        ;Read units of sec data
        movwf   disp_data        ;Save data
        goto    led_disp8        ;Jump to LED control
led_disp0
        decfsz  disp_pw,f        ;Tens of second ?
        goto    led_disp1        ;No. Next

;*** Control TENS of SECOND
        movlw   b'00000100'      ;Set tens of second
        movwf   porta            ;Write PORTA register
        movf    disp_s10,w       ;Read tens of sec data
        movwf   disp_data        ;Save data
        goto    led_disp8        ;Jump to LED control
led_disp1
        decfsz  disp_pw,f        ;Units of minute ?
        goto    led_disp2        ;No. Next

;*** Control UNITS of MINUTE
        movlw   b'00000011'      ;Set units of minute
        movwf   porta            ;Write PORTA register
        movf    disp_m1,w        ;Read units of min data
        movwf   disp_data        ;Save data
        goto    led_disp8        ;Jump to LED control
led_disp2
        decfsz  disp_pw,f        ;Tens of minute ?
        goto    led_disp3        ;No. Next

;*** Control TENS of MINUTE
        movlw   b'00000010'      ;Set tens of minute
        movwf   porta            ;Write PORTA register
        movf    disp_m10,w       ;Read tens of min data
        movwf   disp_data        ;Save data
        goto    led_disp8        ;Jump to LED control
led_disp3
        decfsz  disp_pw,f        ;Units of hour ?
        goto    led_disp4        ;No. Next
```

```
;*** Control UNITS of HOUR
        movlw   b'00000001'     ;Set units of hour
        movwf   porta           ;Write PORTA register
        movf    disp_h1,w       ;Read units of hour data
        movwf   disp_data       ;Save data
        goto    led_disp8       ;Jump to LED control
;*** Control TENS of HOUR
led_disp4
        movlw   b'00000000'     ;Set tens of hour
        movwf   porta           ;Write PORTA register
        movlw   h'0a'           ;Set off data
        movwf   disp_data       ;Save data
        subwf   disp_h10,w      ;H10 - off data
        btfsc   status,z        ;H10 = off data ?
        goto    led_disp8       ;Jump to LED control
        movf    disp_h10,w      ;Read tens of hour data
        movwf   disp_h10w       ;Save tens of hour data
        btfss   status,z        ;AM 0x o'clock ?
        goto    led_disp5       ;No. Next
        movlw   b'11111110'     ;PM=off,Tens=off,AM=on
        goto    led_disp9       ;Jump to PORTC write
led_disp5
        decfsz  disp_h10w,f     ;AM 1x o'clock ?
        goto    led_disp6       ;No. Next
        movlw   b'11111000'     ;PM=off,Tens=1,AM=on
        goto    led_disp9       ;Jump to PORTC write
led_disp6
        decfsz  disp_h10w,f     ;PM 0x o'clock ?
        goto    led_disp7       ;No. Next
        movlw   b'11110111'     ;PM=on,Tens=off,AM=off
        goto    led_disp9       ;Jump to PORTC write
led_disp7
        movlw   b'11110001'     ;PM=on,Tens=1,AM=off
        goto    led_disp9       ;Jump to PORTC write
led_disp8
        movf    disp_data,w     ;Read disply digit data
        addwf   seg7_ha,w       ;Seg7 H.Adr + digit
        movwf   fsr             ;Set FSR register
        movf    indf,w          ;Read seg7 data
led_disp9
        movwf   portc           ;Write LED data
led_dispe
        decfsz  disp_p,f        ;End of cycle ?
        goto    int_end         ;Jump to END of interrupt
        movlw   d'6'            ;Set initial value
        movwf   disp_p          ;Write disply position
        goto    int_end         ;Jump to END of interrupt
;******  Clock count up Process (20msec interval) *******
clock
        bcf     intcon,intf     ;Clear INTF
;*** Time adjust mode check
        movf    mode,w          ;Read time adj mode data
        btfsc   status,z        ;Time adjust mode ?
        goto    adjust          ;Yes. Jump to Adjust Proc
```

```
;*** 0 second adjust check
        btfss   portb,rb6       ;0 sec adjust ?
        goto    check1          ;No.
        movf    rb6ll,w         ;Yes. Read RB6 last look
        btfss   status,z        ;Last look = 0 ?
        goto    check2          ;No. Last look = 1
        incf    rb6ll,f         ;Yes. Set last look
        clrf    disp_s1         ;Clear units of second
        clrf    disp_s10        ;Clear tens of second
        clrf    count           ;Clear clock counter
        goto    check2          ;Jump to time adj check
check1
        clrf    rb6ll           ;Clear RB6 last look
;*** Time adjust demand check
check2
        btfss   portb,rb7       ;Time adjust ?
        goto    check4          ;No.
        movlw   d'100'          ;Set guard (2sec)
        subwf   rb7count,w      ;Counter - Guard
        btfss   status,c        ;Counter >= Guard ?
        goto    check3          ;No. Counter < Guard
        clrf    digit_posi      ;Set position to H10
        movf    disp_h10,w      ;Read tens of hour
        movwf   digit_save      ;Save previous adj data
        clrf    disp_s1         ;Clear units of second
        clrf    disp_s10        ;Clear tens of second
        clrf    count           ;Clear clock counter
        incf    rb7ll,f         ;Set RB7 last look
        bsf     intcon,rbie     ;Set RBIE bit
        clrf    mode            ;Set time adjust mode
        goto    adjust          ;Jump to Adjust process
check3
        incf    rb7count,f      ;Counter + 1
        goto    clock1          ;Jump to clock count up
check4
        clrf    rb7count        ;Clear counter
        clrf    rb7ll           ;Clear RB7 last look
;*** Timer count up
clock1
        movlw   d'49'           ;Set 1 sec data
        subwf   count,w         ;Counter - 1 sec
        btfsc   status,c        ;Counter >= 1 sec ?
        goto    clock_1sec      ;Yes. Counter >= 1 sec
        incf    count,f         ;No. Counter + 1
        goto    int_end         ;Jump to END of interrupt
clock_1sec
        clrf    count           ;Clear 1 second counter
        movlw   d'9'            ;Set check data
        subwf   disp_s1,w       ;S1 - 9
        btfsc   status,c        ;S1 >= 9 sec ?
        goto    clock_10sec     ;Yes. S1 >= 9 sec
        incf    disp_s1,f       ;No. S1 + 1
        bcf     portb,rb1       ;Clear time signal
        goto    int_end         ;Jump to END of interrupt
clock_10sec
```

```
        clrf    disp_s1         ;Set xx:xx:x0
        movlw   d'5'            ;Set check data
        subwf   disp_s10,w      ;S10 - 5
        btfsc   status,c        ;S10 >= 5x sec ?
        goto    clock_1min      ;Yes. S10 >= 5x sec
        incf    disp_s10,f      ;No. S10 + 1
        goto    int_end         ;Jump to END of interrupt

clock_1min
        clrf    disp_s10        ;Set xx:xx:0x
        movlw   d'9'            ;Set check data
        subwf   disp_m1,w       ;M1 - 9
        btfsc   status,c        ;M1 >= 9 min ?
        goto    clock_10min     ;Yes. M1 >= 9 min
        incf    disp_m1,f       ;No. M1 + 1
        goto    int_end         ;Jump to END of interrupt

clock_10min
        clrf    disp_m1         ;Set xx:x0:xx
        movlw   d'5'            ;Set check data
        subwf   disp_m10,w      ;M10 - 5
        btfsc   status,c        ;M10 >= 5x min ?
        goto    clock_1hour     ;Yes. M10 >= 5x min
        incf    disp_m10,f      ;No. M10 + 1
        goto    int_end         ;Jump to END of interrupt

clock_1hour
        clrf    disp_m10        ;Set xx:0x:xx
        movf    disp_h10,w      ;Read tens of hour data
        movwf   disp_h10w       ;Save tens of hour data
        btfss   status,z        ;AM 0x o'clock ?
        goto    hour1           ;No. Next

;*** AM 0x
        movlw   d'9'            ;Set check data
        subwf   disp_h1,w       ;H1 - 9
        btfsc   status,c        ;H1 >= 9 hour ?
        goto    am09            ;Yes. H1 >= 9 hour
        incf    disp_h1,f       ;No. H1 + 1
        goto    time_check      ;Jump to Time Check
am09    clrf    disp_h1         ;Set x0:xx:xx
        incf    disp_h10,f      ;Set AM10:00:00
        goto    time_check      ;Jump to Time Check
hour1
        decfsz  disp_h10w,f     ;AM 1x o'clock ?
        goto    hour2           ;No. Next

;*** AM 1x
        decfsz  disp_h1,w       ;AM 11 o'clock ?
        goto    am10            ;No.  AM 10 o'clock
        goto    am11            ;Yes. AM 11 o'clock
am10    incf    disp_h1,f       ;H1 + 1
        goto    time_check      ;Jump to Time Check
am11    incf    disp_h1,f       ;Set x2:xx:xx
        movlw   d'3'            ;Set PM 1x
        movwf   disp_h10        ;Set PM12:00:00
        goto    time_check      ;Jump to Time Check
hour2
        decfsz  disp_h10w,f     ;PM 0x o'clock ?
        goto    hour3           ;No. Next
```

```

;*** PM 0x
        movlw   d'9'            ;Set check data
        subwf   disp_h1,w       ;H1 - 9
        btfsc   status,c        ;H1 >= 9 hour ?
        goto    pm09            ;Yes. H1 >= 9 hour
        incf    disp_h1,f       ;No. H1 + 1
        goto    time_check      ;Jump to Time Check
pm09    clrf    disp_h1         ;Set x0:xx:xx
        incf    disp_h10,f      ;Set PM10:00:00
        goto    time_check      ;Jump to Time Check

;*** PM 1x
hour3
        movlw   d'1'            ;Set check data
        subwf   disp_h1,w       ;H1 - 1
        btfsc   status,z        ;H1 = 1 hour ?
        goto    pm11            ;Yes. PM 11 o'clock
        movlw   d'2'            ;Set check data
        subwf   disp_h1,w       ;H1 - 2
        btfsc   status,c        ;H1 >= 2 hour ?
        goto    pm12            ;Yes. PM 12 o'clock
        incf    disp_h1,f       ;No. H1 + 1
        goto    time_check      ;Jump to Time Check
pm11    clrf    disp_h1         ;Set 0 o'clock
        clrf    disp_h10        ;Set AM00:00:00
        goto    time_check      ;Jump to Time Check
pm12    movlw   d'1'            ;Set data
        movwf   disp_h1         ;Set 1 o'clock
        movlw   d'2'            ;Set data
        movwf   disp_h10        ;Set PM01:00:00
        goto    time_check      ;Jump to Time Check

;*** Time signal check
time_check
        btfsc   disp_h10,1      ;AM ?
        goto    tck4            ;No. PM
        movlw   d'7'            ;Set AM 7:00 data
        subwf   disp_h1,w       ;H1 - check data
        btfss   status,z        ;AM 7:00 ?
        goto    tck1            ;No. Next
        goto    time_signal     ;Yes. Jump to time signal
tck1    movlw   d'8'            ;Set AM 8:00 data
        subwf   disp_h1,w       ;H1 - check data
        btfss   status,z        ;AM 8:00 ?
        goto    tck2            ;No. Next
        goto    time_signal     ;Yes. Jump to time signal
tck2    movlw   d'9'            ;Set AM 9:00 data
        subwf   disp_h1,w       ;H1 - check data
        btfss   status,z        ;AM 9:00 ?
        goto    tck3            ;No. Next
        goto    time_signal     ;Yes. Jump to time signal
tck3    btfss   disp_h10,0      ;AM 1x ?
        goto    no_signal       ;No. End of signal check
        movf    disp_h1,w       ;Read H1
        btfss   status,z        ;AM 10:00 ?
        goto    no_signal       ;No. End of signal check
        goto    time_signal     ;Yes. Jump to time signal

tck4    movlw   d'6'            ;Set PM 6:00 data
```

```
            subwf   disp_h1,w         ;H1 - check data
            btfss   status,z          ;PM 6:00 ?
            goto    tck5              ;No. Next
            goto    time_signal       ;Yes. Jump to time signal
tck5        movlw   d'7'              ;Set PM 7:00 data
            subwf   disp_h1,w         ;H1 - check data
            btfss   status,z          ;PM 7:00 ?
            goto    tck6              ;No. Next
            goto    time_signal       ;Yes. Jump to time signal
tck6        movlw   d'8'              ;Set PM 8:00 data
            subwf   disp_h1,w         ;H1 - check data
            btfss   status,z          ;PM 8:00 ?
            goto    tck7              ;No. Next
            goto    time_signal       ;Yes. Jump to time signal
tck7        movlw   d'9'              ;Set PM 9:00 data
            subwf   disp_h1,w         ;H1 - check data
            btfss   status,z          ;PM 9:00 ?
            goto    no_signal         ;No. End of signal check
            goto    time_signal       ;Yes. Jump to time signal

time_signal
            bsf     portb,rb1         ;Time signal ON
no_signal
            goto    int_end           ;Jump to END of interrupt

;****** Time adjust mode Process (20msec interval) ******
adjust
;*** Adjust end check
            btfss   portb,rb6         ;0 sec adjust SW = ON ?
            goto    adjust1           ;No. Next process
            movf    digit_posi,w      ;Yes. Read digit position
            movwf   digit_posiw       ;Save digit position
            btfss   status,z          ;Position = H10 ?
            goto    adj_end1          ;No. Next
            movf    digit_save,w      ;Yes. Read saved digit
            movwf   disp_h10          ;Recover digit
            goto    adj_end4          ;Jump to adj mode end
adj_end1
            decfsz  digit_posiw,f     ;Position = H1 ?
            goto    adj_end2          ;No. Next
            movf    digit_save,w      ;Yes. Read saved digit
            movwf   disp_h1           ;Recover digit
            goto    adj_end4          ;Jump to adj mode end
adj_end2
            decfsz  digit_posiw,f     ;Position = M10 ?
            goto    adj_end3          ;No. Next
            movf    digit_save,w      ;Yes. Read saved digit
            movwf   disp_m10          ;Recover digit
            goto    adj_end4          ;Jump to adj mode end
adj_end3
            movf    digit_save,w      ;Read saved digit
            movwf   disp_m1           ;Recover digit
adj_end4
            incf    rb6ll,f           ;Set last look ON
            bcf     intcon,rbie       ;Clear RBIE bit
            incf    mode,f            ;Set clock mode
            goto    int_end           ;Jump to END of interrupt

;*** Adjust position check
adjust1
```

```
        btfss   portb,rb7        ;Position SW = ON ?
        goto    adj_posi10       ;No. SW = OFF
        movf    rb7l1,w          ;Yes. Read RB7 last look
        btfss   status,z         ;Last look = 0 ?
        goto    adjust2          ;No. Last look = 1
        incf    rb7l1,f          ;Yes. Set last look
        incf    digit_posi,f     ;Change position
        movlw   d'4'             ;Set check data
        subwf   digit_posi,w     ;Position data - 4
        btfss   status,c         ;Position over ?
        goto    adj_posi1        ;No. digit proc
        clrf    digit_posi       ;Set position to H10
adj_posi1
        movf    digit_posi,w     ;Read digit position
        movwf   digit_posiw      ;Save digit position
        btfss   status,z         ;Position = H10 ?
        goto    adj_posi3        ;No. Next
        movf    blink_cont,w     ;Read blink control
        btfsc   status,z         ;LED OFF ?
        goto    adj_posi2        ;No. LED ON
        movf    digit_save,w     ;Yes. Read saved digit
        movwf   disp_m1          ;Set M1 digit
adj_posi2
        movf    disp_h10,w       ;Read digit
        goto    adj_posi9        ;Jump to digit save
adj_posi3
        decfsz  digit_posiw,f    ;Position = H1 ?
        goto    adj_posi5        ;No. Next
        movf    blink_cont,w     ;Read blink control
        btfsc   status,z         ;LED OFF ?
        goto    adj_posi4        ;No. LED ON
        movf    digit_save,w     ;Yes. Read saved digit
        movwf   disp_h10         ;Set H10 digit
adj_posi4
        movf    disp_h1,w        ;Read digit
        goto    adj_posi9        ;Jump to digit save
adj_posi5
        decfsz  digit_posiw,f    ;Position = M10 ?
        goto    adj_posi7        ;No. Next
        movf    blink_cont,w     ;Read blink control
        btfsc   status,z         ;LED OFF ?
        goto    adj_posi6        ;No. LED ON
        movf    digit_save,w     ;Yes. Read saved digit
        movwf   disp_h1          ;Set H1 digit
adj_posi6
        movf    disp_m10,w       ;Yes. Read digit
        goto    adj_posi9        ;Jump to digit save
adj_posi7
        movf    blink_cont,w     ;Read blink control
        btfsc   status,z         ;LED OFF ?
        goto    adj_posi8        ;No. LED ON
        movf    digit_save,w     ;Yes. Read saved digit
        movwf   disp_m10         ;Set M10 digit
adj_posi8
        movf    disp_m1,w        ;Read digit
adj_posi9
        movwf   digit_save       ;Save digit
        goto    adjust2
adj_posi10
        clrf    rb7l1            ;Clear RB7 last look
```

```

;*** Adjust digit blink process
adjust2
          movlw    d'10'           ;Set 200 msec data
          subwf    digit_blink,w   ;Counter - 200 msec
          btfsc    status,c        ;Counter >= 200 msec ?
          goto     adj_blk1        ;Yes. Counter >= 200 msec
          incf     digit_blink,f   ;No. Counter + 1
          goto     int_end         ;Jump to END of interrupt
adj_blk1
          clrf     digit_blink     ;Clear Blink counter
          btfsc    blink_cont,0    ;Blink ON ?
          goto     adj_blk5        ;No. Jump to ON process

;*** LED OFF process
          incf     blink_cont,f    ;Set Blink OFF data
          movf     digit_posi,w    ;Read digit position
          movwf    digit_posiw     ;Save digit position
          btfss    status,z        ;Position = H10 ?
          goto     adj_blk2        ;No. Next
          movlw    h'0a'           ;Yes. Set LED off digit
          movwf    disp_h10        ;LED off
          goto     adj_blke        ;Jump to blink end
adj_blk2
          decfsz   digit_posiw,f   ;Position = H1 ?
          goto     adj_blk3        ;No. Next
          movlw    h'0a'           ;Yes. Set LED off digit
          movwf    disp_h1         ;LED off
          goto     adj_blke        ;Jump to blink end
adj_blk3
          decfsz   digit_posiw,f   ;Position = M10 ?
          goto     adj_blk4        ;No. Next
          movlw    h'0a'           ;Yes. Set LED off digit
          movwf    disp_m10        ;LED off
          goto     adj_blke        ;Jump to blink end
adj_blk4
          movlw    h'0a'           ;Yes. Set LED off digit
          movwf    disp_m1         ;LED off
          goto     adj_blke        ;Jump to blink end

;*** LED ON process
adj_blk5
          clrf     blink_cont      ;Set Blink ON data
          movf     digit_posi,w    ;Read digit position
          movwf    digit_posiw     ;Save digit position
          btfss    status,z        ;Position = H10 ?
          goto     adj_blk6        ;No. Next
          movf     digit_save,w    ;Read saved digit
          movwf    disp_h10        ;Set H10 digit
          goto     adj_blke        ;Jump to blink end
adj_blk6
          decfsz   digit_posiw,f   ;Position = H1 ?
          goto     adj_blk7        ;No. Next
          movf     digit_save,w    ;Read saved digit
          movwf    disp_h1         ;Set H1 digit
          goto     adj_blke        ;Jump to blink end
adj_blk7
          decfsz   digit_posiw,f   ;Position = M10 ?
          goto     adj_blk8        ;No. Next
          movf     digit_save,w    ;Read saved digit
```

```
        movwf   disp_m10          ;Set M10 digit
        goto    adj_blke          ;Jump to blink end
adj_blk8
        movf    digit_save,w      ;Read saved digit
        movwf   disp_m1           ;Set M1 digit
adj_blke
        goto    int_end           ;Jump to END of interrupt
;**************** Digit change process *****************
digit_change
        bcf     intcon,rbif       ;Clear RBIF
        movf    portb,w           ;Read PORTB
        andlw   b'00110000'       ;Pick up RB4 and RB5
        movwf   change_wk         ;Save RB4/RB5 condition
        movf    change_st,w       ;Read Digit change status
        btfss   status,z          ;Status = "0" ?
        goto    change2           ;No. Next
        movf    change_wk,w       ;Read RB4/RB5 condition
        xorlw   b'00100000'       ;Check RB4/RB5 condition
        btfss   status,z          ;RB5(B)=1 RB4(A)=0 ?
        goto    change1           ;No. next
        movlw   d'1'              ;Set status to "1"
        movwf   change_st         ;Write status
        goto    int_end           ;Jump to END of interrupt
change1
        movf    change_wk,w       ;Read RB4/RB5 condition
        xorlw   b'00110000'       ;Check RB4/RB5 condition
        btfss   status,z          ;RB5(B)=1 RB4(A)=1 ?
        goto    int_end           ;Jump to END of interrupt
;*** Count up process
        movlw   d'2'              ;Set status to "2"
        movwf   change_st         ;Write status
        movf    digit_posi,w      ;Read digit position
        movwf   digit_posiw       ;Save digit position
        btfss   status,z          ;Position = H10 ?
        goto    count_up2         ;No. Next
        movlw   d'3'              ;Set check data
        subwf   digit_save,w      ;H10 - check data
        btfss   status,z          ;H10 = 3 ?
        goto    count_up1         ;No.
        clrf    digit_save        ;Set H10 = 0
        goto    count_h10         ;Jump to save check
count_up1
        incf    digit_save,f      ;H10 + 1
        goto    count_h10         ;Jump to save check
count_up2
        decfsz  digit_posiw,f     ;Position = H1 ?
        goto    count_up8         ;No. Next
        movf    disp_h10,w        ;Read H10 digit
        andlw   b'00000001'       ;Pick up 0x/1x
        btfss   status,z          ;H10 = AM 0x or PM 0x ?
        goto    count_up4         ;No. AM 1x or PM 1x
        movlw   d'9'              ;Set check data
        subwf   digit_save,w      ;H1 - check data
        btfss   status,z          ;H1 = 9 ?
```

```
        goto    count_up3       ;No.
        clrf    digit_save      ;Set H1 = 0
        goto    count_h1        ;Jump to save check
count_up3
        incf    digit_save,f    ;H1 + 1
        goto    count_h1        ;Jump to save check
count_up4
        movf    disp_h10,w      ;Read H10 digit
        andlw   b'00000010'     ;Pick up AM/PM
        btfss   status,z        ;H10 = AM ?
        goto    count_up6       ;No. PM
        movf    digit_save,w    ;Read H1 digit
        btfss   status,z        ;H1 = 0 ?
        goto    count_up5       ;No. H1 > 1
        incf    digit_save,f    ;H1 = 1
        goto    count_h1        ;Jump to save check
count_up5
        clrf    digit_save      ;H1 = 0
        goto    count_h1        ;Jump to save check
count_up6
        movlw   d'2'            ;Set check data
        subwf   digit_save,w    ;H1 - check data
        btfss   status,c        ;H1 >= 2 ?
        goto    count_up7       ;No.
        clrf    digit_save      ;Set H1 = 0
        goto    count_h1        ;Jump to save check
count_up7
        incf    digit_save,f    ;H1 + 1
        goto    count_h1        ;Jump to save check
count_up8
        decfsz  digit_posiw,f   ;Position = M10 ?
        goto    count_up10      ;No. Next
        movlw   d'5'            ;Set check data
        subwf   digit_save,w    ;M10 - check data
        btfss   status,z        ;M10 = 5 ?
        goto    count_up9       ;No.
        clrf    digit_save      ;Set M10 = 0
        goto    count_m10       ;Jump to save check
count_up9
        incf    digit_save,f    ;M10 + 1
        goto    count_m10       ;Jump to save check
count_up10
        movlw   d'9'            ;Set check data
        subwf   digit_save,w    ;M1 - check data
        btfss   status,z        ;M1 = 9 ?
        goto    count_up11      ;No.
        clrf    digit_save      ;Set M1 = 0
        goto    count_m1        ;Jump to save check
count_up11
        incf    digit_save,f    ;M1 + 1
        goto    count_m1        ;Jump to save check
change2
        movlw   d'1'            ;Set check data
        subwf   change_st,w     ;Status - check data
        btfss   status,z        ;Status = "1" ?
        goto    change4         ;No. Next
```

```
        movf    change_wk,w     ;Read RB4/RB5 condition
        xorlw   b'00010000'     ;Check RB4/RB5 condition
        btfss   status,z        ;RB5(B)=0 RB4(A)=1 ?
        goto    change3         ;No. next
        clrf    change_st       ;Set status to "0"
        goto    int_end         ;Jump to END of interrupt
change3
        movf    change_wk,w     ;Read RB4/RB5 condition
        xorlw   b'00110000'     ;Check RB4/RB5 condition
        btfss   status,z        ;RB5(B)=1 RB4(A)=1 ?
        goto    int_end         ;Jump to END of interrupt

;*** Count down process
        movlw   d'2'            ;Set status to "2"
        movwf   change_st       ;Write status
        movf    digit_posi,w    ;Read digit position
        movwf   digit_posiw     ;Save digit position
        btfss   status,z        ;Position = H10 ?
        goto    count_down2     ;No. Next

        movf    digit_save,w    ;Read H10
        btfss   status,z        ;H10 = 0 ?
        goto    count_down1     ;No.
        movlw   d'3'            ;Set data
        movwf   digit_save      ;Set H10 = 3
        goto    count_h10       ;Jump to save check
count_down1
        decf    digit_save,f    ;H10 - 1
        goto    count_h10       ;Jump to save check

count_down2
        decfsz  digit_posiw,f   ;Position = H1 ?
        goto    count_down9     ;No. Next

        movf    disp_h10,w      ;Read H10 digit
        andlw   b'00000001'     ;Pick up 0x/1x
        btfss   status,z        ;H10 = AM 0x or PM 0x ?
        goto    count_down4     ;No. AM 1x or PM 1x
        movf    digit_save,w    ;Read H1
        btfss   status,z        ;H1 = 0 ?
        goto    count_down3     ;No.
        movlw   d'9'            ;Set data
        movwf   digit_save      ;Set H1 = 9
        goto    count_h1        ;Jump to save check
count_down3
        decf    digit_save,f    ;H1 - 1
        goto    count_h1        ;Jump to save check
count_down4
        movf    disp_h10,w      ;Read H10 digit
        andlw   b'00000010'     ;Pick up AM/PM
        btfss   status,z        ;H10 = AM ?
        goto    count_down6     ;No. PM
        movf    digit_save,w    ;Read H1 digit
        btfss   status,z        ;H1 = 0 ?
        goto    count_down5     ;No. H1 = 1
        incf    digit_save,f    ;H1 = 1
        goto    count_h1        ;Jump to save check
count_down5
        clrf    digit_save      ;H1 = 0
        goto    count_h1        ;Jump to save check
```

```
count_down6
            movlw   d'3'            ;Set check data
            subwf   digit_save,w    ;H1 - check data
            btfsc   status,c        ;H1 >= 3 ?
            goto    count_down7     ;Yes.
            movf    digit_save,w    ;read H1
            btfss   status,z        ;H1 = 0 ?
            goto    count_down8     ;No.
count_down7
            movlw   d'2'            ;Set data
            movwf   digit_save      ;Set H1 = 2
            goto    count_h1        ;Jump to save check
count_down8
            decf    digit_save,f    ;H1 - 1
            goto    count_h1        ;Jump to save check
count_down9
            decfsz  digit_posiw,f   ;Position = M10 ?
            goto    count_down11    ;No. Next
            movf    digit_save,w    ;Read M10
            btfss   status,z        ;M10 = 0 ?
            goto    count_down10    ;No.
            movlw   d'5'            ;Set data
            movwf   digit_save      ;Set M10 = 5
            goto    count_m10       ;Jump to save check
count_down10
            decf    digit_save,f    ;M10 - 1
            goto    count_m10       ;Jump to save check
count_down11
            movf    digit_save,w    ;Read M1
            btfss   status,z        ;M1 = 0 ?
            goto    count_down12    ;No.
            movlw   d'9'            ;Set data
            movwf   digit_save      ;Set M1 = 9
            goto    count_m1        ;Jump to save check
count_down12
            decf    digit_save,f    ;M1 - 1
            goto    count_m1        ;Jump to save check
count_h10
            movf    blink_cont,w    ;Read blink control data
            btfss   status,z        ;Blink ON ?
            goto    int_end         ;Jump to END of interrupt
            movf    digit_save,w    ;Yes. Read H10 data
            movwf   disp_h10        ;Set H10 data
            goto    int_end         ;Jump to END of interrupt
count_h1
            movf    blink_cont,w    ;Read blink control data
            btfss   status,z        ;Blink ON ?
            goto    int_end         ;Jump to END of interrupt
            movf    digit_save,w    ;Yes. Read H1 data
            movwf   disp_h1         ;Set H1 data
            goto    int_end         ;Jump to END of interrupt
count_m10
            movf    blink_cont,w    ;Read blink control data
            btfss   status,z        ;Blink ON ?
```

```
        goto    int_end          ;Jump to END of interrupt
        movf    digit_save,w     ;Yes. Read M10 data
        movwf   disp_m10         ;Set M10 data
        goto    int_end          ;Jump to END of interrupt

count_m1
        movf    blink_cont,w     ;Read blink control data
        btfss   status,z         ;Blink ON ?
        goto    int_end          ;Jump to END of interrupt
        movf    digit_save,w     ;Yes. Read M1 data
        movwf   disp_m1          ;Set M1 data
        goto    int_end          ;Jump to END of interrupt

change4
        movf    change_wk,w      ;Read RB4/RB5 condition
        xorlw   b'00000000'      ;Check RB4/RB5 condition
        btfss   status,z         ;RB5(B)=0 RB4(A)=0 ?
        goto    int_end          ;No. END of interrupt
        clrf    change_st        ;Yes. Set status to "0"
        goto    int_end          ;Jump to END of interrupt

;********************************************************
;                    END of Digital Clock
;********************************************************

        end
```

HEX CODE

```
:020000000528D1
:080008004528831606309F0015
:1000100000308500FD308600003087000230810000E
:10002000831283308100850186O1FF308700A001A3
:100030000630A100A501A601A701A801A901AA01F6
:10004000AB01AC01AD01AE01AF01B201B301B4012E
:100050003730B6004030B7007930B8002430B900EE
:100060003030BA001930BB001230BC000230BD0085
:100070007830BE000030BF001030C0007F30C100BB
:100080002330C200B8308B0044280B088B18A028FE
:100090000B1955280B18C7290B30360784000008A8
:1000A000870005308500532809000B11833081003B
:1000B000FF3087002108A200A20B632805308500CD
:1000C0002A08A3009628A20B6A2804308500290874
:1000D000A3009628A20B71280330850028 08A300EE
:1000E0009628A20B78280230850027 08A3009628BE
:1000F000A20B7F28013085002608A3009628003037
:10010000085000A30A3002502031996282508A400BB
:1001100031D8C28FE309A28A40B9028F8309A28CA
:10012000A40B9428F7309A28F1309A282308360730
:100130008400000887 00A10B54280630A100542831
:100140008B102B0803194929061FAE282C08031D04
:10015000AF28AC0AAA01A901A001AF28AC01861FF3
:10016000C12864302E02031CBF28AF012508B1004E
:10017000AA01A901A001AD0A8B15AB014929AE0A5C
:10018000C328AE01AD01313020020318C928A00AEE
```

:100190005428A00109302A020318D128AA0A86107F
:1001A0005428AA01053029020318D828A90A54287E
:1001B000A901093028020318DF28A80A5428A80139
:1001C000053027020318E628A70A5428A7012508A6
:1001D000A400031DF428093026020318F128A60AFA
:1001E0001C29A601A50A1C29A40BFF28260BF92807
:1001F000FB28A60A1C29A60A0330A5001C29A40B6B
:100200000A29093026020318 0729A60A1C29A60173
:10021000A50A1C290130260203191429023026 02DE
:1002200003181729A60A1C29A601A5011C290130BB
:10023000A6000230A5001C29A51833290730260284
:10024000031D2329472908302602031D2829472991
:1002500009302602031D2D294729251C4829260877
:10026000031D4829472906302602031D382947293E
:1002700007302602031D3D29472908302602031DA9
:1002800042294729093026020 31D48294729861497
:100290005428061F62292F08B000031D5229310877
:1002A000A5005E29B00B57293108A6005E29B00BC6
:1002B0005C293108A7005E293108A800AC0A8B111F
:1002C000AB0A5428861F93292D08031D9429AD0AD3
:1002D000AF0A04302F02031C6E29AF012F08B000B3
:1002E000031D79293308031977293108A800250847
:1002F0009129B00B82293308031980293108A50000
:1003000026089129B00B8B29330803198929310854
:10031000A600270891293308031990293108A7005E
:100320002808B1009429AD010A30320203189A2935
:10033000B20A5428B2013318B229B30A2F08B00008
:10034000031DA5290A30A500C629B00BAA290A3029
:10035000A600C629B00BAF290A30A700C6290A306B
:10036000A800C629B3012F08B000031DBA2931081F
:10037000A500C629B00BBF293108A600C629B00BBD
:10038000C4293108A700C6293108A80054280B1039
:1003900006083039B5003408031D1A2A3508203AFA
:1003A000031DD5290130B40054283508303A031D07
:1003B00054280230B4002F08B000031DE729033091
:1003C0003102031DE529B1016D2AB10A6D2AB00B76
:1003D000082A25080139031DF52909303102031DBA
:1003E000F329B101732AB10A732A25080239031DC2
:1003F000002A3108031DFE29B10A732AB101732AAC
:1004000002303102031C062AB101732AB10A732A91
:10041000B00B122A05303102031D102AB101792ACE
:10042000B10A792A09303102031D182AB1017F2A45
:10043000B10A7F2A01303402031D852A3508103A9B
:10044000031D242AB40154283508303A031D5428CA
:100450000230B4002F08B000031D362A3108031DF6
:10046000342A0330B1006D2AB1036D2AB00B5B2A28
:1004700025080139031D442A3108031D422A093089
:10048000B100732AB103732A25080239031D4F2ACC
:100490003108031D4D2AB10A732AB101732A0330B2

:1004A00031020318562A3108031D592A0230B100BF
:1004B000732AB103732AB00B652A3108031D632A1E
:1004C0000530B100792AB103792A3108031D6B2A5E
:1004D0000930B1007F2AB1037F2A3308031D542855
:1004E0003108A50054283308031D54283108A600FC
:1004F00054283308031D54283108A7005428330812
:10050000031D54283108A80054283508003A031D5B
:060510005428B401542838
:02400E00723FFF
:00000001FF