



NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

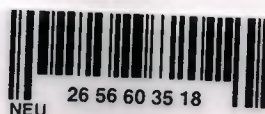
**MULTITASKING APPLICATION ON
MICROCONTROLLERS**

MASTER THESIS

Student: Majdi Rasmi Khalil Tibi (20033596)

Supervisor: PROF.DR.DOGAN IBRAHIM

Nicosia – 2005





Majdi Tibi: Multitasking application on microcontrollers.

**Approval of the Graduate School of Applied and
Social Sciences**

**Prof. Dr. Fakhraddin Mamedov
Director**

**We certify this thesis is satisfactory for the award of the
Degree of Master of Science in Computer Engineering**

Examining Committee in charge:

Assoc. Prof. Dr. Rahib Abiyev, Committee Chairman, Electrical
Engineering Department, NEU

Assist. Prof. Dr. Kadri Bürüncük, Committee Member, Computer
Engineering Department, NEU

Assist. Prof. Dr. Adil Amircanov, Committee Member, Computer
Engineering Department, NEU

Mr. Kaan Uyar, Committee Member, Computer
Engineering Department, NEU

Prof. Dr. Doğan İbrahim, Supervisor, Chairman of Computer
Engineering Department, NEU

*Dedicated to my Parents,
Sisters, Brothers, Uncles
And my Fiancée*

ACKNOWLEDGEMENTS

Firstly, I would like to present my special appreciation to my supervisor *Prof. Dr. Doğan Ibrahim*, without whom it would have not been possible for me to complete my thesis. His trust in my work and me and his priceless awareness for the project has made me do my work with full interest. His friendly behavior with me and his words of encouragement kept me doing my thesis.

Secondly, I offer special thanks to my uncle Dr. Jawad Tibi, who encouraged me in every field of life and tried to help whenever I needed. He enhanced my confidence in myself to make me able to face every difficulty easily. I am also grateful to my sisters, brothers and my fiancée. And because of them I am able to complete my work.

Finally, I would also like to pay my special thanks to all of my friends who helped me and encouraged me for doing my work. Their continuous encouragement and friendly environment has helped me to complete this thesis successfully. I wish to express my sincere thanks to them as they spent their time and provided very helpful suggestions to me.

ABSTRACT

A microcontroller is a single chip computer which contains the CPU, data and program memory, serial and parallel I/O, timers, and interrupt logic. About 40% of microcontroller applications are in office automation, such as PCs, laser printers, fax machines, intelligent telephones, and so forth. About one-third of microcontrollers are found in consumer electronics goods. Products like CD players, hi-fi equipment, video games, washing machines, cookers and so on fall into this category. The communications market, automotive market, and the military share the rest of the microcontroller application areas.

Basically, a microcontroller executes a user program which is loaded in its program memory. Under the control of this program, data is received from external devices (input devices), manipulated, and then sent to external devices (output devices).

Microcontrollers have traditionally been used to control a single device. But as the demand for complex control operations have increased, the need to control multiple devices at the same time has also increased. This is known as multitasking where a single microcontroller is used to control more than one task at the same time.

This thesis describes the various multitasking algorithms and develops simple multitasking algorithms which can be implemented on low-cost microcontrollers. The PIC family of microcontrollers is chosen as the target microcontroller in this thesis. PIC is currently one of the most popular microcontrollers, used extensively by many engineers, students, and hobbyists.

It is shown in the thesis that simple, but effective multitasking algorithms can be developed on the PIC microcontrollers using the popular PIC Basic language, and the native PIC assembler language.

TABLE OF CONTENTS

DEDICATED	i
ACKNOWLEDGEMENTS	ii
ABSTRACT	iii
TABLE OF CONTENTS	iv
LIST OF FIGURES	viii
LIST OF TABLES	x
INTRODUCTION	1
1. MULTITASKING	4
1.1 Overview	4
1.2 Single Tasking	4
1.3 What is Multitasking	5
1.3.1 Comparing between StateMachines and TimeSlicing	8
1.4 Deterministic Multitasking	9
1.5 How does Multitasking work?	12
1.6 Types of Multitasking	14
1.6.1 Cooperative Multitasking	14
1.6.2 Time Slice Multitasking	16
1.6.3 Preemptive Multitasking	17
1.7 The concept of tasking	18
1.7.1 Scheduler	18
1.7.1.1 First-Come, First-Served (FCFS)	19
1.7.1.2 Shortest-Job-First (SJB)	20
1.7.1.3 Shortest Remaining Time First	21
1.7.1.4 Round Robin Task Scheduling	22
1.7.1.5 highest runnable task	22
1.7.1.6 Priority and Round Robin Tasking	23
1.8 Why Multitasking on Microcontrollers is important?	24
1.9 Summary	25

2. MICROCONTROLLERS	26
2.1 Overview	26
2.2 What is a microcontroller?	26
2.3 Microprocessors and Microcontroller	29
2.4 Basic elements of Microcontroller	30
2.4.1 Memory Unit	30
2.4.2 Central Processing Unit	31
2.4.3 Bus	32
2.4.4 Input-Output Unit	33
2.4.5 Serial Communication	33
2.4.6 Timer Unit	35
2.4.7 Watchdog	35
2.4.8 Analog To Digital Converter	36
2.5 Embedded Controller	38
2.6 Microcontroller Applications	39
2.6.1 Environmental Applications	39
2.6.2 Industrial Applications	40
2.7 PIC Microcontroller	40
2.7.1 The PIC16F84 Microcontroller	41
2.7.1.1 Internal Components Of PIC16F84	42
2.7.2 The PIC16F877 Microcontroller	51
2.8 Summary	52
3. MICROCONTROLLER SYSTEM DEVELOPMENT CYCLE	53
3.1 Overview	53
3.2 Basic elements of PIC BASIC language	53
3.2.1 Identifiers	53
3.2.2 Labels	54
3.2.3 Constants	54
3.2.4 Variables	55
3.2.5 Sequences	56
3.2.6 Modifiers	57

3.2.7 Symbols	57
3.2.8 Direction Include	57
3.2.9 Comments	58
3.2.10 Programming ling with more instructions	59
3.2.11 Transfer of instruction into another line	59
3.2.12 Define	59
3.2.13 Disable	62
3.2.14 Enable	62
3.2.15 On Interrupt	63
3.2.16 Resume	63
3.3 PIC BASIC Compiler	63
3.4 Writing and Compilation of a BASIC program	64
3.5 Loading a program into the Microcontroller memory	65
3.6 Running your program	67
3.7 Summary	69
4. MULTITASKING APPLICATION ON PIC MICROCONTROLLER	70
4.1 Overview	70
4.2 Programming Languages Of Microcontroller	70
4.3 Single Tasks On PIC Microcontroller Using PIC BASIC	71
4.3.1 Led Diode Example	71
4.3.2 Button Example	74
4.3.3 Seven-Segment Displays Example	78
4.3.4 Building Light Control Example	85
4.4 Multitasking Application On PIC Microcontroller	87
4.4.1 State Machine Multitasking Circuit Block Diagram	87
4.4.2 Analogue Temperature Sensor (LM35)	88
4.4.3 Analogue To Digital Converter (ADC0804)	88
4.4.4 BCD To 7segment Decoder (74LS47)	89
4.4.5 The Circuit Diagram	90
4.4.6 Implementing The State Machine Algorithm	91
4.4.7 Implementing The Time Sliced Multitasking Algorithm	94

4.4.8 Implementing The Cooperative Multitasking Algorithm	96
4.4.8.1 Generalising To Many Tasks	101
4.5 Summary	109
CONCLUSION	110
REFERENCES	111

LIST OF FIGURES

1.1 Single tasking and multitasking	13
1.2 First-come, first-served scheduling	20
1.3 Shortest-job-first scheduling	21
1.4 Round robin task scheduling.	22
1.5 Tasks running with this priority.	23
2.1 Example of simplified model of a memory unit.	30
2.2 Example of simplified central processing unit with three registers.	31
2.3 Connecting memory and central processing unit.	32
2.4 Example of a simplified input-output.	33
2.5 Serial unit used to send data, but only by three lines.	34
2.6 Timer unit generates signals in regular time intervals.	35
2.7 Watchdog reset.	35
2.8 Blocks for converting an analogue to a digital form.	36
2.9 Physical configuration of the interior of a microcontroller.	37
2.10 Microcontroller outline with its basic elements and internal connections.	37
2.11 PINs of PIC19F84A.	42
2.12 Flash program memory.	43
2.13 Special function registers.	44
2.14 Block diagram of PIC16F84A.	45
2.15 Call instruction.	47
3.1 The PIC BASIC compiler.	64
3.2 The connection between PC, programming device and the microcontroller.	67
3.3 LM7805 regulator circuit.	68
3.4 Switching on LED.	68
4.1 Diagram of LED diode.	72
4.2 LED diode is turned on by a logical one.	72
4.3 LED diode is turned on by a logical zero.	73
4.4 LED diodes are connected to portB and are turned on by a logical one.	73
4.5 Button with "PULL-UP" resistor.	75
4.6 Button with "PULL-DOWN" resistor.	76

4.7 Four buttons connected to the microcontroller using the pull-up resistors.	76
4.8 Seven-Segment digits.	79
4.9 Connecting seven segment displays in multiplex mode with the microcontroller.	79
4.10 Building light control.	85
4.11 Block diagram of multitasking circuit.	88
4.12 LM35 Pin configuration.	88
4.13 ADC0804 Pin configuration.	89
4.14 74LS47 pin configuration.	89
4.15 The circuit diagram.	90
4.16 The Basic program.	94
4.17 Time sliced multitasking code	96
4.18 Cooperative multitasking with 2 tasks.	99
4.19 Screen shot of the simulation.	101
4.20 Generalised Cooperative multitasking algorithm.	102
4.21 Multitasking Cooperative n tasks.	106
4.22 Multitasking Cooperative 5 tasks.	109

LIST OF TABLES

1.1 Comparison between state machines and timeslicing.	8
2.1 Pin descriptions.	42
2.2 SFR functions.	45
2.3 Initialization circuits.	51
2.4 Comparison of some popular microcontrollers.	52
3.1 The size of the sequence.	56
3.2 The use of a direction DEFINE.	60
4.1 Contains corresponding mask values for numbers 0-9.	81

INTRODUCTION

From microwave ovens to alarm systems to industrial programmable logic controllers (PLCs) and distributed systems (DCSs), embedded controllers are running our world. Embedded controllers are used in most items of electronic equipment today. They can be thought of as intelligent electronic devices used to control and monitor devices connected to the real world. This can be a PLC, DCS or a smart sensor. These devices are used in almost every walk of life today. Most automobiles, factories and even kitchen appliances have embedded controllers in them.

The microcontrollers that are at the heart of these and many more devices are becoming easier and simpler to use. The sheer volume of embedded controllers used in the world drives us to understand how they work and then how to troubleshoot and repair them. The support chips used in these controllers are becoming smarter and easier to use. This is bringing the design and use of embedded controllers to more and more engineers hence the need for a good understanding of what embedded controllers are and how to troubleshoot them.

Microcontrollers are intelligent electronic devices used to control and monitor devices connected to the real world. This can be a microwave oven, programmable logic controller, distributed control system, car braking system, cruise missile control system, or a smart sensor. As time goes on electronic devices get smarter and smaller, the embedded controller will be in or associated with everything we touch throughout the day. Early embedded controllers contained a CPU and a multitude of support chips. As time went on, support chips were included in the CPU chip until it became a microcontroller. A microcontroller is defined as a CPU plus random access memory (RAM), electrically erasable programmable read only memory (EEPROM), input-outputs (I/O), and communication circuits. The embedded controller is a microcontroller with peripherals such as keypads, displays, and relays connected to it and is often connected to other embedded controllers by way of some type of communications system.

The microcontroller is a direct descendent of the CPU, in fact every microcontroller has a CPU as the heart of the device. It is therefore important to understand the CPU in order to ultimately understand the microcontroller and embedded controller.

The central processor unit is the brain of the microcontroller. The CPU controls all functions and uses the program that resides in RAM, EEPROM or EPROM to function. The program may reside in one or more of these devices at the same time. Part of the program might be in RAM while another might be in EEPROM. A program is a sequence of instructions that tell the CPU what to do. These instructions could be compared to instructions a teacher may give to a student to get a desired result. The instructions sent to the CPU are very, very simple and it usually takes many instructions to get the CPU to do what is necessary to accomplish a task.

Microcontrollers have traditionally been programmed using the native assembly language of the target processor. It is very common nowadays to use high-level languages such as Basic, Pascal, and C in programming microcontrollers. Assembly language has the advantage that the execution speed is very fast. On the other hand, developing an assembly language based program is a complex task. High-level languages have the advantage that it is much easier to develop and maintain programs developed using these languages. The main disadvantage of the high-level languages is that the speed of execution is not as fast as the programs developed using the assembly language.

Microcontrollers have traditionally been programmed to control a single device and multi-processors, consisting of several microcontrollers are generally used to control more than one device at the same time. It is also possible to develop multitasking algorithms on microcontrollers such that a single microcontroller can be used to control a number of devices all at the same time.

This thesis is about the use of multitasking algorithms on simple low-cost microcontrollers, such as the PIC family of microcontrollers. The thesis describes the development of several multitasking algorithms which can be implemented on PIC microcontrollers. The PIC16F84 microcontroller is taken as an example in the thesis. Multitasking algorithms have been developed in the thesis using the Basic high-level

programming language and the native assembly language of the PIC microcontrollers. It is shown in the thesis that the low-cost microcontrollers can be programmed to operate as multitasking processors.

The thesis consists of the introduction and four chapters:

Chapter 1 presents the principles of multitasking and describes the various multitasking algorithms used in practice. This chapter also explains the importance of multitasking when used on microcontrollers.

Chapter 2 provides an introduction to the architecture of the PIC microcontrollers and describes the important features of the popular PIC16F84 microcontroller.

Chapter 3 describes the microcontroller system development cycle, the use of program description language, and the important features of the PIC Basic compiler.

Chapter 4 presents the principles of single task operation, and simple practical examples are provided to demonstrate the single task operation. The principles of multitasking are also described in this chapter and various simple multitasking algorithms are developed using the high-level Basic language and the native PIC assembler language.

A conclusion and a list of references are provided at the end of the thesis.

1. MULTITASKING

1.1 OVERVIEW

For the majority of embedded systems, a single tasking operating system is too restrictive. What is required is an operating system that can run multiple applications simultaneously and provide inter task control and communication. The facilities once only available to mini and mainframe computer users are now required by 8, 16 and 32 bit microprocessors and microcontrollers.

Multitasking is the process of letting the operating system perform multiple tasks at what seems to the user simultaneously. Multitasking enables a complex task to be implemented by designing separate tasks that operate independently or cooperate with each other. This Chapter describes the principles of various multitasking and scheduling algorithms.

1.2 SINGLE TASKING

The first widely used operating system was CP/M [17], developed for the Intel 8080 microprocessor and 8" floppy disk systems. It supported I/O calls by jump tables and quickly became standard within the industry and a large amount of application software became available for it. Many of the micro-based business machines of the late 1970s and early 1980s were based on CP/M. Its ideas even formed the basis of the popular MSDOS operating system, chosen by IBM for its personal computers.

CP/M is a good example of a single tasking operating system. Only one task or application can be executed at any one time and therefore it only supports one user at a time. With a single tasking operating system, it is not possible to run multiple tasks simultaneously. Large applications have to be run sequentially and can not support concurrent operations. There is no support for message passing or task control, which would enable applications to be divided into separate entities. If a system needs to take log data and store it on disk and, at the same time, allow a user to process that data using an online database package, a single tasking operating system would need everything to be integrated. With a multitasking operating system, the data logging task can run at the same time as the

database. Data can be passed between each element by a common file on disk, and neither task need have any direct knowledge of the other. With a single tasking system, it is likely that the database program would have to be written from scratch. With the multitasking system, a commercially available program can be used, and the logging software interfaced to it. These restrictions forced many applications to interface directly with the hardware and therefore lose the hardware independence that the operating system offered. Such software would need extensive modifications to port it to another configuration.

1.3 WHAT IS MULTITASKING?

On a typical microcontroller [1] the CPU usually performs only one task at a time and when that task is completed the next task can start. This is how the processor performs operations in a typical real-time application. Most complex processors are designed to operate in a multi-tasking manner (e.g. a PC) where the processor can execute a number of tasks concurrently.

Multitasking is, on single-processor machines, implemented by letting the running process own the CPU for a while (a timeslice) and when required gets replaced with another process, which then owns the CPU. The two most common methods for sharing the CPU time is either cooperative multitasking or preemptive multitasking.

Multitasking is the capability of performing many functions in a simultaneous or quasi-simultaneous manner. State machines and timeslicing are two popular multitasking methods with long traditions. State machines have been used to design complex systems with high reliability requirements.

In the early ages of the computers, microprocessors and microcontrollers were designed so that they can be used in real-time applications. Each processor was designed so that it could be used in a single standalone application. The advances in electronic engineering increased the processing power many times and made it possible to design multitasking real-time applications. In SMP (symmetric Multi Processor systems) this is the case, since there are several CPU's to execute programs on - in systems with only a single CPU this is

done by switching execution very rapidly between each program, thus giving the impression of simultaneous execution. This process is also known as task switching or timesharing. Practically all modern operating systems have this ability.

A multitasking operating system works by dividing the processor's time into discrete time slots. Each application or task requires a certain number of time slots to complete its execution. The operating system kernel decides which task can have the next slot, so instead of a task executing continuously until completion, its execution is interleaved with other tasks. This sharing of processor time between tasks gives the illusion to each user that he is the only one using the processor.

Multitasking operating systems are based around a multitasking kernel which controls the time slicing mechanisms. A time slice is the time period each task has for execution before it is stopped and replaced during a context switch. This is periodically triggered by a hardware interrupt from the system timer. This interrupt may provide the system clock and several interrupts may be executed and counted before a context switch is performed. When a context switch is performed, the current task is interrupted, the processor's registers are saved in a special table for that particular task and the task is placed back on the ready list to await another time slice. Special tables, often called task control blocks store all the information the system requires about the task, for example its memory usage, the priority level within the system and the error handling. It is the context information that is switched when one task is replaced by another. The ready list contains all the tasks and their status and is used by the scheduler to decide which task is allocated the next time slice.

The scheduling algorithm determines the sequence and takes into account a task's priority and present status. If a task is waiting for an I/O call to complete, it will be held in limbo until the call is complete. Once a task is selected, the processor registers and status at the time of its last context switch are loaded back into the processor and the processor is started. The new task carries on as if nothing had happened until the next context switch takes place.

The kernel controls memory usage and prevents tasks from corrupting each other. If required, it also controls memory sharing between tasks, allowing them to share common program modules, such as runtime libraries. A set of memory tables is maintained, which is used to decide if a request is accepted or rejected. This means that resources, such as physical memory and peripheral devices, can be protected from other tasks without using memory management provided the task is disciplined enough to use the operating system and not access the resources directly.

Message passing and control can be implemented in such systems by using the kernel to act as a message passer and controller between tasks. If task A wants to stop task B, then by executing a call to the kernel, the status of task B can be changed and its execution halted. Alternatively, task B can be delayed for a set time period or forced to wait for a message.

With a typical real-time operating system, there are two basic types of messages that the kernel will deal with:

- Flags that can control but can not carry any implicit information
- Messages which can carry information and control tasks

The kernel maintains the tables required to store this information and is responsible for ensuring that tasks are controlled and receive the information. With the facility for tasks to communicate between each other, system call support for accessing I/O, loading tasks etc can be achieved by running additional tasks, with a special system status. These system tasks provide additional facilities and can be included as required.

Many multitasking operating systems available today are also described as real-time. These operating systems provide additional facilities allowing applications that would normally interface directly with the microprocessor architecture to use interrupts and drive peripherals to do so without the operating system blocking such activities.

The choice of a scheduler algorithm can play an important role in the design of an embedded microcontroller and can dramatically affect the underlying design of the software. There are many different types of scheduler algorithms that can be used, each

with either different characteristics or different approaches to solving the same problem of how to assign priorities to schedule tasks so that correct operation is assured.

1.3.1 COMPARING BETWEEN STATE MACHINES AND TIMESLICING

A **state machine** requires that the task is split into states. The state machine stays in one state at a time, and switch to another state when the specified conditions are met. Actions are performed during the transitions. States represent a situation that is stable for some time interval. State machines can be implemented on all types of microcontrollers, also on the smallest microcontrollers.

Timeslicing means that the kernel interrupts each process after some milliseconds and gives control to another task. Thus, each task is given CPU processing at regular intervals. This mechanism is general but requires that the kernel can pop and push program addresses on the stack, and is thus not available on the midrange microcontroller devices.

Designing processes for timeslicing is very similar to designing independent programs. Programmers find this to be natural and straight forward. State machines are usually more complex to design than a process.

Table 1.1 shows a comparison of the state machines and the timeslicing systems.

Table 1.1: Comparison between state machines and timeslicing

	State Machines	Timeslicing Systems
Fast response	+	
Code readability		+
Ease of design		+
Predictability	+	
Max response delay	+	
Reliability	+	
Debugging	+	

Faster response: The State machines have faster response than a timeslicing system. One of the reasons for this is that a state machine does not require the context to be saved during a task switch. Timeslicing is done by interrupt and requires that CPU registers, stack and local workspace are saved before a new task can start execution. The second reason for the slower response is that each task executes for a predefined time period before control is given to the next task. So, when many tasks are executing, the delay becomes longer. Preemptive scheduling may cure much of the slow response and make a timeslicing system more predictive. Still, state machines are faster because each task is divided into small fragments by the programmer.

Debugging: a timeslicing system can be extremely difficult. State machines are significantly easier to debug. A state machine operates independently of interrupts, and the individual states can be transmitted to the outside world and displayed on a monitor. This have shown to be very efficient during debugging.

Reliability: The State machines score highest on the reliability issues also. Designing a RTOS is a very complex task, and RTOS kernel bugs can be a problem. Because of the interrupt mechanisms, both application and RTOS bugs are often not easily reproduced. Using the reset button is not satisfactory on a real-time system. A state machine operates without a RTOS kernel and without the interrupt mechanism.

Easier code writing seems to be the only advantage of a timeslicing system over state machines.

1.4 DETERMINISTIC MULTITASKING

The idea of Deterministic Multitasking is to make state machines look more like system processes. This enhances readability and makes code writing easier. Still, the concept is fully compatible with state machines, and inherits all the advantages.

An example state machine implementation is given below. In this example, the task *ventilationControl()* controls ventilation on/off depending on the temperature. The behaviour of the code is:

- Switch ventilation off
- Wait until the temperature is high
- Switch on ventilation
- Wait until the temperature is low
- Delay 100 seconds until stopping ventilation
- Keep ventilation on if the temperature goes high again while waiting
- Go to step 1 (turn off ventilation)

An implementation of the above example as a state machine may look like:

```

Char stateVC;

Void ventilationControl(void)
{
    switch (stateVC)
    {
        case 0:
            ventilation = off;
            stateVC = 1;
            break;

        case 1:
            if(!highTemperature) break;
            stateVC = 2;
            ventilation = on;

        case 2:
            if(highTemperature) break;
            stateVC = 3;
            startTimer(100);

        case 3:
    
```

```

        if(!timeout)
        {
            if(highTemperature)stateVC = 2;
            break;
        }
        stateVC = 0;
        break;
    }
}

```

Note that the state machine is implemented using a case block and there are no wait loops in a state machine. At each iteration, a short check is made to determine if it is time to change state. The iterations are performed by making calls to the function regularly.

The same code could be implemented as a deterministic multitasking by allowing the task to be written like a procedure or process. The equivalent deterministic multitasking implementation is given below:

```

Task ventilationControl(void)
{
    ventilation = off;
    while(!highTemperature)waitState();
    ventilation = on;
    VENTILATION_ON:
        While(highTemperature)waitState();
        StartTimer(100);
        While(!timeout)
        {

```



```

        if(highTemperature)goto VENTILATION_ON;

        waitState();

    }

    restartTask();

}

```

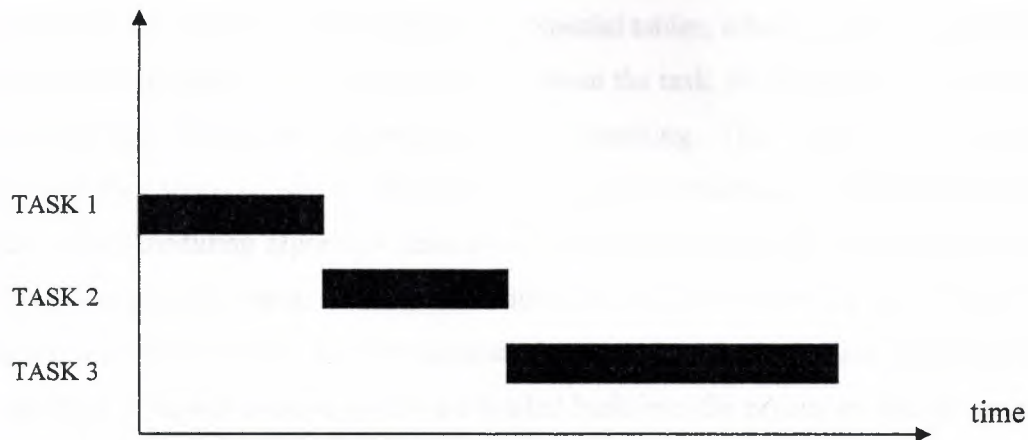
The main difference from the state machine definition is the hiding of state information and insertion of waitState(). This enables the compiler to select the state during compilation.

1.5 HOW DOES MULTITASKING WORK?

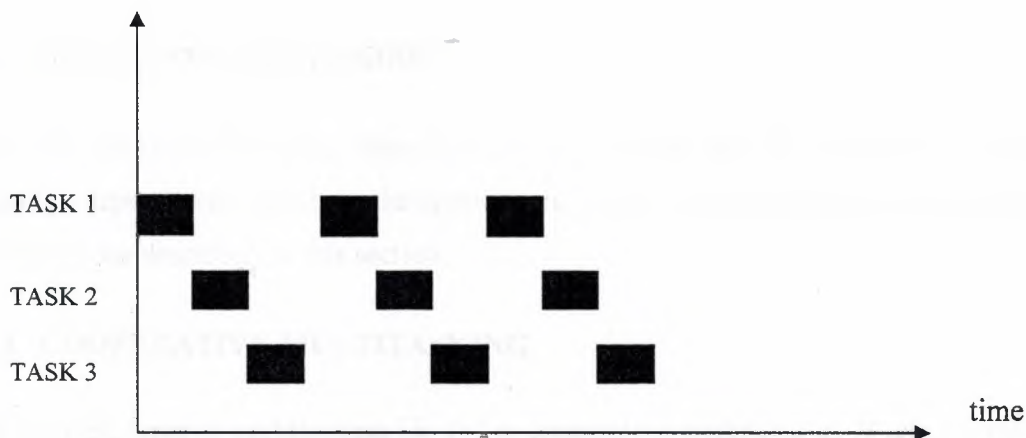
The idea of multitasking [2] is to run more than one task “at the same time.” In some sense, we want it to look like we have divided our computer into many (slower) computers. Each (slower) computer is performing a different task. One important property is that different tasks can go through their lives without knowing that the other tasks even exist. If we had separate CPUs, separate memory, separate disks, (separate computers really), this would be easy. However, we want to be able to run more tasks than we have CPUs. In the simplest case, we only have one CPU, one memory bank, and one set of disks. Given these resources, we want to be able to run more than one task.

A conventional processor can only execute a single task at a time, but by rapidly switching between tasks a multitasking operating system can make it appear as if each task is executing concurrently. This is shown in Figure 1.1 with three tasks. The upper diagram shows the three tasks executing on a single tasking kernel where each task runs to completion before the next one starts. This is the behaviour of most real-time microcontroller systems. In the lower diagram the multitasking behaviour is shown where the tasks share the available processor time. In this diagram, each task is given a certain amount of processor time and the tasks give-up the processor when they use their times. Although still the processor executes one code at any time, it appears to the user that all the

three tasks are running at the same time. I.e. as if there are three processors, each running a different task.



Running 3 tasks on a single tasking processor



Running 3 tasks on a multitasking processor

Figure 1.1 Single tasking and multitasking

Multitasking operating systems are based around a multitasking kernel which controls the time slicing mechanisms. A time slice is the time period each task has for execution before it is stopped and replaced during a context switch. This is periodically triggered by a hardware interrupt from the system timer. This interrupt may provide the system clock and several interrupts may be executed and counted before a context switch is performed.

When a context switch is performed, the current task is interrupted, the processor's registers are saved in a special table for that particular task and the task is placed back on the "ready" list to await another time slice. Special tables, often called task control blocks, store all the information the system requires about the task, for example, its memory usage, its priority level within the system and its error handling. The "ready" list contains all the tasks and their status is used by the scheduler to decide which task is allocated the next time slice. The scheduling algorithm determines the sequence and takes into account a task's priority and present status. If a task is waiting for an I/O to complete, it will be held until the call is complete. Once a task is selected for execution, the processor registers and status at the time of its last context switch are loaded back into the processor and the processor is started. The new task carries on as if nothing had happened until the next context switch takes place. This is the basic methods behind all multitasking operating systems.

1.6 TYPES OF MULTITASKING

There are many multitasking algorithms [18] available and the choice of a particular algorithm depends very much on the application. Some of the commonly used multitasking algorithms are described in this section.

1.6.1 COOPERATIVE MULTITASKING

The simplest form of multitasking [3, 18] is cooperative multitasking. It lets the programs decide when they wish to let other tasks run. This method is not good since it lets one process to monopolise the CPU and never let other processes run. This way a program may be reluctant to give away processing power in the fear of another process hogging all CPU-time. Early versions of the MacOS (up till MacOS 8) and versions of Windows earlier than Win95/WinNT used cooperative multitasking (Win95 when running old applications).

Cooperative multitasking has the advantages that it is very simple to implement and it can be implemented nearly on all microcontrollers. Since the tasks must all cooperate for context switching to occur, the scheduler is less dependent on interrupts and can be small.

Also, the programmer knows exactly when context switching will occur, and can protect critical regions of code simply by keeping a context-switching call out of that part of the code. One disadvantage of the cooperative multitasking is that a task may never give-up the CPU time and when this happens only one task can run in the system.

Salvo from the Pumpkin systems [19] is a cooperative multitasking kernel developed for the PIC microcontrollers. This kernel provides support for event and timer services and it works on a priority basis. Tasks that share the same priority execute in a round-robin fashion.

The operation of a cooperative multitasking processor is summarized below by using a program description language and by considering 3 simple tasks. In this example, each task starts and runs independent of each other and they give-up the CPU time whenever they wish.

TASK1:

BEGIN

DO FOREVER

.....

Give-up CPU

.....

Give-up CPU

.....

ENDDO

END

TASK2:

BEGIN

DO FOREVER


```

.....
Give-up CPU
.....
Give-up CPU
.....
ENDDO
END

```

TASK3:

```

BEGIN
DO FOREVER
.....
Give-up CPU
.....
Give-up CPU
.....
ENDDO
END

```

1.6.2 TIME SLICE MULTITASKING

This type of multitasking has been described earlier. Time slicing multitasking works by making the tasks to switch at regular periodic points in time. This means any task that needs to run next will have to wait until the current time slice is completed or until the current task suspends its operation deliberately. Time slicing is usually implemented using timer interrupts such that a context switching occurs whenever a timer interrupt is generated. i.e. when a timer interrupt is generated the processor stores the context of the running task and then selects the next task to run. The choice of which task to run next is

determined by the scheduling algorithm and thus is nothing to do with the time slicing mechanism itself. It just happens that many time slice based systems use a round-robin scheduler to distribute the time slices across all tasks that need to run. One commonly used technique when selecting the next task is to use priorities and always select a task with the highest priority.

For real-time systems where the speed of execution is very critical, the time slice period plus the context switching time determine the context switch time of the system. With most time slice periods in the order of milliseconds, it is the dominant factor in the system response.

1.6.3 PREEMPTIVE MULTITASKING

The alternative to the cooperative multitasking is to use preemption where a currently running task can be stopped and switched out by a higher priority task. The main difference between preemptive multitasking and other forms of multitasking is that with the preemptive multitasking the context switching does not need to wait for the end of a time slice or for a task to give-up the CPU.

Preemptive multitasking moves the control of the CPU to the operating system, letting each process run for a given amount of time (a timeslice) and then switching to another task. This method prevents one process from taking complete control of the system and thereby making it seem as if it has crashed. This method is most common today, implemented by among others OS/2, Win95/98, WinNT, UNIX, Linux, BeOS, QNX, OS9 and most mainframe operating systems. The assignment of CPU time is taken care of by the scheduler.

Preemptive multitasking is very stack-intensive. The scheduler maintains a separate stack for each task so that when a task resumes execution after a context switch, all the stack values that are unique to the task are properly in place. These would normally be return addresses from subroutine calls, and parameters and local variables.

Preemptive schedulers are generally quite complex because of the myriad of issues that must be addressed to properly support context switching at any time. This is especially true with regard to the handling of interrupts.

1.7 THE CONCEPT OF TASKING

1.7.1 SCHEDULER

The scheduler is the component of the operating system, which is responsible for the assignment of CPU time to tasks. It usually implements a priority engine which lets it assign more CPU time to high priority tasks.

Many different schemes are used to implement the multitasking architecture. The scheduling algorithm depends on the following factors:

- Fairness: each processor gets its share of the CPU
- Efficiency: CPU utilization
- Throughput: number of processes/time unit
- Turnaround: time it takes to execute a process from start to finish
- Waiting time: total time spent in the ready queue
- Response time: amount of time it takes to start responding

It is desirable that all tasks get the CPU time they need and to maximize the CPU utilization and throughput, and to minimize the turnaround time, waiting time, and response time. Scheduling of preemptive systems is more complex than the scheduling of cooperative or time slicing systems. Generally in preemptive systems a scheduling is done when one or more of the following conditions are satisfied:

- A new task is created
- A task is destroyed
- A task waits for I/O

- A task blocks and waits for some event to happen (e.g. a timer event)
- An I/O or an interrupt occurs

Some of the commonly used scheduling algorithms are given below.

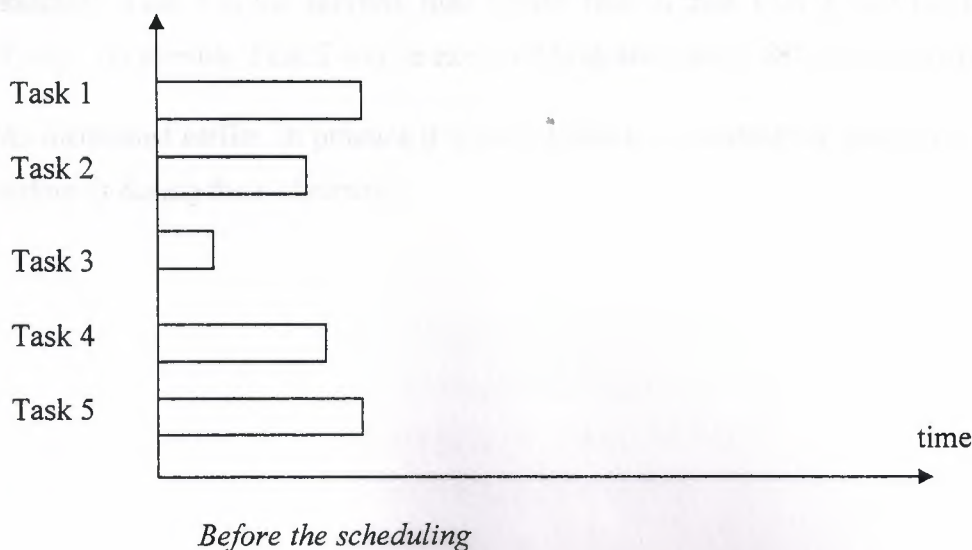
1.7.1.1 FIRST-COME, FIRST-SERVED (FCFS)

In this algorithm, the scheduling depends on the arrival times of the tasks. A task which is eligible to run first is executed first. In FCFS each task is placed in a single queue, the first task in the queue is selected, and allowed to run as long as it wants. If the task blocks, the next task in the queue is selected to run. When a blocked task becomes eligible to run, it is placed at the end of the queue. This type of scheduling has the disadvantage that it is difficult to manage the running of the tasks in an orderly manner.

FCFS scheduling has the following drawbacks:

- A process that does not perform any I/O can monopolize the CPU
- I/O bound processes have to wait until CPU-bound process completes

Figure 1.2 shows an example first-come, first-served scheduling. In this example there are 5 tasks and task 1 arrives slightly earlier than the other tasks. In this example it is assumed that all tasks are immediately eligible to run and the execution time is indicated on the x-axis.



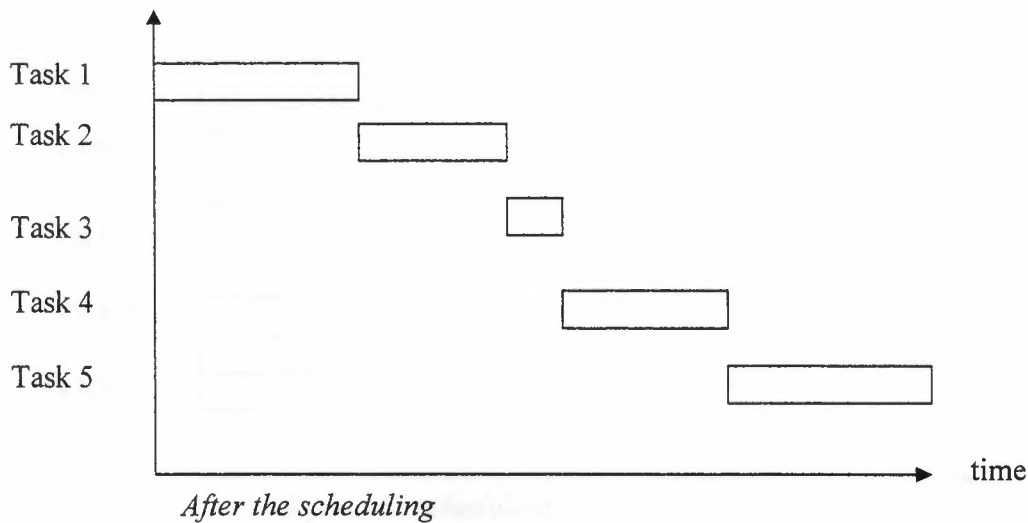


Figure 1.2 First-come, first-served scheduling

1.7.1.2 SHORTEST-JOB-FIRST (SJB)

In this algorithm, the task with the shortest execution time is run first. The scheduler will have to know the execution times of the tasks in the system and it then selects the tasks with the shortest execution times. SJB is simple but it has the disadvantage that the longer tasks may never be called to run. Also, it is not always possible to predict the shortest job in advance.

Figure 1.3 shows an example shortest-job-first scheduling with 5 tasks. As shown in this example, Task 3 is the shortest, then comes Task 2, then Task 4, and the longest task is Task 5. As a result, Task 3 will be executed first, and Task 5 will be executed last.

As mentioned earlier, in practice it is very difficult to estimate the execution times of tasks before or during their lifetimes.

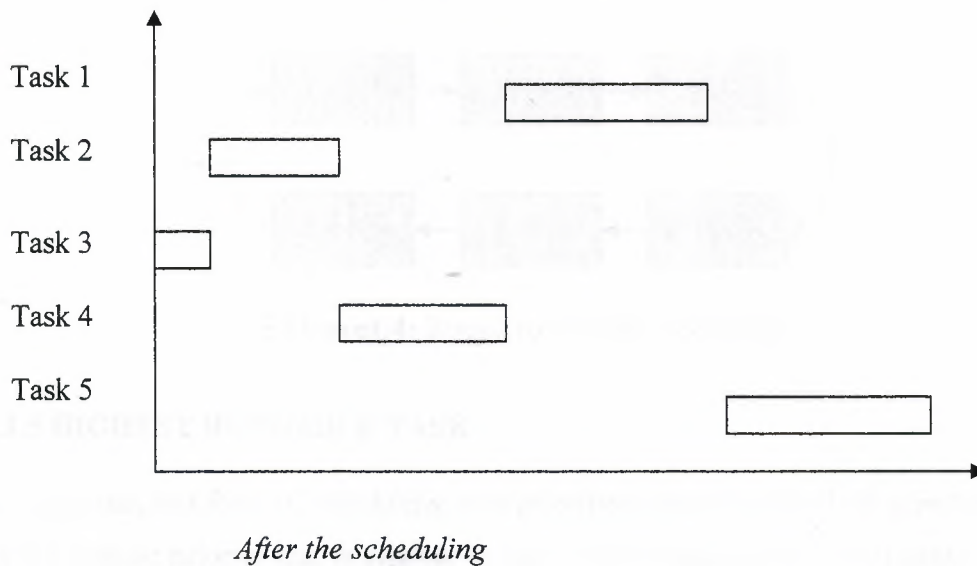
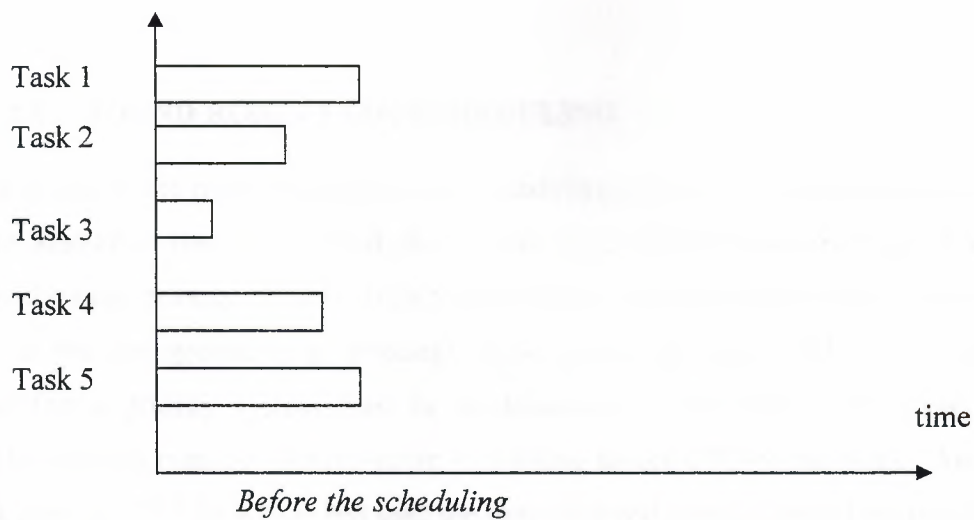


Figure 1.3 Shortest-job-first scheduling

1.7.1.3 SHORTEST REMAINING TIME FIRST

This scheduling algorithm is similar to the shortest-job-first. But here, whenever a task arrives, we choose the one with the shortest remaining time to execute first. In this algorithm new short jobs get good services. The disadvantage of this algorithm is that it is not always easy to predict the task with the shortest remaining execution time.

1.7.1.4 ROUND ROBIN TASK SCHEDULING

This is one of the most commonly used scheduling techniques. The scheduler assigns the same amount of time to each task that is ready in a ring as shown in Figure 1.4. All tasks have the same priority. This is slightly problematic, since one often wish certain tasks to be put in the background (e.g. printing) while others get more CPU time (like games). Therefore a priority system must be implemented. Round-robin scheduling is easy to implement and manage. Round-robin scheduling favors CPU-bound tasks. An I/O bound task uses the CPU for a time less than the time slice and then is blocked waiting for I/O.

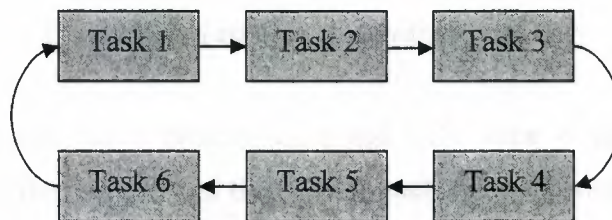


Figure1.4: Round robin task scheduling

1.7.1.5 HIGHEST RUNNABLE TASK

This is the simplest form of scheduling with priorities where the idea is to select the process with the highest priority that is eligible to run. This method is very predictable and yields satisfactory results.

Highest unable task algorithm has the disadvantage that if the tasks with the high priorities take too much CPU time then the lower priority tasks may never run. This is especially true if the higher priority task is in a loop and is executing continuously. When this happens, the lower priority tasks will be denied any CPU time.

1.7.1.6 PRIORITY AND ROUND ROBIN TASKING

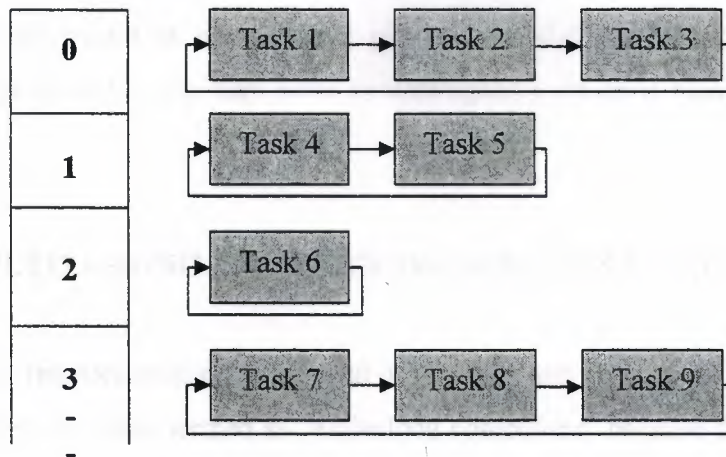


Figure 1.5: Tasks running with this priority

In Figure 1.5 each task has a priority class and CPU time is assigned to each class corresponding to its priority level and then within each class the round-robin scheduling is used. This technique is commonly used by many operating systems. Imagine that a total of 100 timeslices are available and that 2 is the lowest class. Now, if we say that a class-0 task gets 3 times as much time as a class-2 task and twice as much as a class-1, the division of slices is 3:2:1 for each task. If we imagine all tasks running at class-2, class-1 tasks would be twice in a round and class-0 would be 3 times in a round (thus ensuring the priority). Therefore the total timeslices per round would be $3*3+2*2+1=14$ slices/round. The total slice count was 100 leaving a total of $3*100/14$ to a class-0 task, $2*100/14$ to a class-1 task and $100/14$ to a class-2 task.

If we want to know in which order they are to be executed, it must be something like (priority-levels): 0-1-0-1-0-2-.

One implementation of this would be that for each class, one has a counter that tells us how many timeslices a class must be given before the system is reset to standards (in our example c0:3, c1:2, c2:1). Then, one starts at the highest level, gives one timeslice to c0, finds the first class below, that earns a slice and gives it, and then going to the top again. Repeating this would give us a sharing in our example 0-1-0-1-0-2-, making a quite even

assignment between priority classes. An alternate method will be starting at c0 and run down the class-list and restarting when reaching the button. This is repeated until all classes counters is 0 (this would in our example give 0-1-2-0-1-0-). This algorithm has one problem - a high-priority task may have to wait quite a while if many tasks are being multitasked.

1.8 WHY MULTITASKING ON MICROCONTROLLERS IS IMPORTANT?

Multitasking on microcontrollers [4] is an important part of a majority of electronic applications. They are often termed as "embedded controllers" because they are implanted in an electronic device and they control the actions or features of this device. Also, as a result of their compact size, low cost, low power consumption and ease in implementation, multitasking on microcontroller is extensively used in the current electronic products.

Multitasking on microcontrollers mean that one can make more than one task to run on a microcontroller at the same time. Most microcontrollers are inexpensive single chip computers. The microcontroller's ability to store and run unique programs makes it extremely versatile. For instance, one can program a microcontroller to make decisions (perform functions) based on predetermined situations (I/O-line logic) and selections. The microcontroller's ability to perform math and logic functions allows it to mimic sophisticated logic and electronic circuits.

Microcontrollers are responsible for the "intelligence" in most smart devices on the consumer market and the use of multitasking into these devices provides many benefits, such as cost saving and simplicity in design.

1.9 SUMMARY

Multitasking enables a complex job to be implemented by designing separate tasks that operate independently or cooperate with each other. Each task is usually so simple that it can be described by some sequential behavior. Each task is simple, but the total collection of tasks enables a complex job to be solved relatively easily. In this chapter we have discussed the basic concepts of multitasking, and the types of multitasking algorithms.

2. MICROCONTROLLERS

2.1 OVERVIEW

This chapter describes the principles of microcontrollers and also explains the uses and benefits of microcontroller applications in industrial and environmental fields. Moreover, it discusses the basic elements and the architecture of a microcontroller. In this chapter the following topics will be discussed:

- What is a microcontroller?
- Microprocessors and Microcontrollers
- Basic elements of a microcontroller
- Embedded controllers
- Microcontroller applications
- PIC microcontrollers

2.2 WHAT IS A MICROCONTROLLER?

A microcontroller [4,5,7] is a computer on a chip. All computers, whether it is a microcomputer, a personal computer, a mini computer, or a large mainframe computer have several things in common:

- All computers have a CPU (central processing unit) that executes programs. If you are sitting at a desktop computer right now reading this thesis, the CPU in that machine is probably executing a program that implements the Web browser.
- The CPU loads the program from somewhere. On your desktop machine, the browser program is loaded from the hard disk.
- The computer has some RAM (random-access memory) where it can store "variables."
- And the computer has some input and output devices so it can talk to the outside world. On your desktop machine, the keyboard and mouse are input devices and the monitor and printer are output devices. A hard disk

is an I/O device -- it handles both input and output, and it is also a storage device which stores data.

The desktop computer you are using is a "general purpose computer" that can run any of thousands of commercially available programs. Microcontrollers are "special purpose computers." There are a number of other common characteristics that define microcontrollers. If a computer matches a majority of the following characteristics, then one can call it a "microcontroller":

- Microcontrollers are generally "embedded" inside some other device (often a consumer product) so that they can control the features or actions of the product. Another name for a microcontroller, therefore, is "embedded controller."
- Microcontrollers are dedicated to one task and run one specific program. The program is stored in ROM (read-only memory) and generally does not change.
- Microcontrollers are often low-power devices. A desktop computer is almost always plugged into a wall socket and might consume 50 watts of electricity. A battery-operated microcontroller on the other hand might only consume 50 milliwatts or less.
- A microcontroller has a dedicated input device and often (but not always) has a small LED or LCD display for output. A microcontroller also takes input from the device it is controlling and controls the device by sending signals to different components within the device. A program runs on the microcontroller which controls all actions of the microcontroller at any moment in time.

For example, the microcontroller inside a TV takes inputs from the remote control unit and displays outputs on the TV screen. The controller controls the channel selector, the speaker system and certain adjustments on the picture tube electronics such as tint and brightness. The engine controller in a car takes inputs from sensors such as the oxygen and knock sensors and controls things like fuel mix and spark plug timing. A

microwave oven controller takes input from a keypad, displays outputs on an LCD display and controls a relay that turns the microwave generator on and off.

- A microcontroller is often small and low cost. The components are chosen to minimize size and to be as inexpensive as possible.
- A microcontroller is often, but not always, ruggedized in some way.

The microcontroller controlling a car's engine, for example, has to work in temperature extremes that a normal computer generally cannot handle. A car's microcontroller in Alaska has to work fine in -30 degree F (-34 C) weather, while the same microcontroller in Nevada might be operating at 120 degrees F (49 C). When you add the heat naturally generated by the engine, the temperature can go as high as 150 or 180 degrees F (65-80 C) in the engine compartment. On the other hand, a microcontroller embedded inside a VCR need not be ruggedized at all.

The actual processor used to implement a microcontroller can vary widely. For example, a Digital Cell Phone can contain a Z-80 processor. The Z-80 [6] is an 8-bit microprocessor developed in the 1970s and originally used in home computers of the time. The Garmin GPS (Global Positioning System) contains a low-power version of the Intel 80386. The 80386 was originally used in desktop computers.

In many products, such as microwave ovens, the demand on the CPU is fairly low and price is an important consideration. In these cases, manufacturers turn to dedicated microcontroller chips -- chips that were originally designed to be low-cost, small, low-power, embedded CPUs. The Motorola 6811 [15], and Intel 8051 [16] are both good examples of such chips. There is also a line of popular controllers called "PIC microcontrollers" created by a company called Microchip [12]. By today's standards, these CPUs are incredibly minimalistic; but they are extremely inexpensive when purchased in large quantities and can often meet the needs of a device's designer with just one chip.

Any microcontroller chip might have X bytes of ROM and Y bytes of RAM on the chip, along with Z number of I/O pins. In large quantities, the cost of these chips can

sometimes be just pennies. You certainly are never going to run Microsoft Word on such a chip -- Microsoft Word requires perhaps 30 megabytes of RAM and a processor that can run millions of instructions per second. But then, you don't need Microsoft Word to control a microwave oven, either. With a microcontroller, usually one has a specific task to accomplish, and low-cost, low-power performance is what is important in such applications.

2.3 MICROPROCESSORS AND MICROCONTROLLERS

A digital computer [7] typically consists of three major components: the Central Processing Unit (CPU), program and data memory, and an Input/Output (I/O) system. The CPU controls the flow of information among the components of the computer. It also processes the data by performing digital operations. Most of the processing is done in the Arithmetic Logic Unit (ALU) within the CPU. When the CPU of a computer is built on a single printed circuit board, the computer is usually called a microprocessor based computer. A microprocessor is a CPU that is compacted into a single chip semiconductor device. Microprocessors are general-purpose devices, suitable for many applications. A computer built around a microprocessor is called a microcomputer. The choice of I/O and memory devices of a microcomputer depends on the specific application- for example, most personal computers contain a keyboard and monitor as standard input and output devices.

A microcontroller is an entire computer manufactured on a single chip. The I/O and memory subsystems contained in a microcontroller specialize these devices so that they can be interfaced with hardware and control functions of the applications. Since microcontrollers are powerful digital processors, the degree of control and programmability they provide significantly enhances the effectiveness of the application. Microcontrollers are usually dedicated devices embedded within an application. For example, microcontrollers are used as engine controllers in automobiles and as exposure and focus controllers in cameras. In order to serve these applications, they have a high concentration of on-chip facilities such as serial ports, parallel input-output ports, timers, counters, interrupt control, analog-to-digital converters, random access memory, and read only memory.

Embedded control applications also distinguish the microcontroller from its relative, the general-purpose microcontroller. Embedded systems often require real-time operation and multitasking capabilities. Real-time operation refers to the fact that the embedded controller must be able to receive and process the signals from its environment when they are available, that is, the environment must not wait for the controller to become available. Similarly, the controller must perform fast enough to output control signals to its environment when they are needed. Again, the environment must not wait for the controller. In other words, the embedded controller should not be a bottleneck in the operation of the system.

Multitasking is the capability of performing many functions in a simultaneous or quasi-simultaneous manner. The embedded controller is often responsible for monitoring several aspects of a system and responding appropriately when the need arises.

2.4 BASIC ELEMENTS OF A MICROCONTROLLER

2.4.1 MEMORY UNIT

Memory [8] 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, as shown in Figure 2.1.

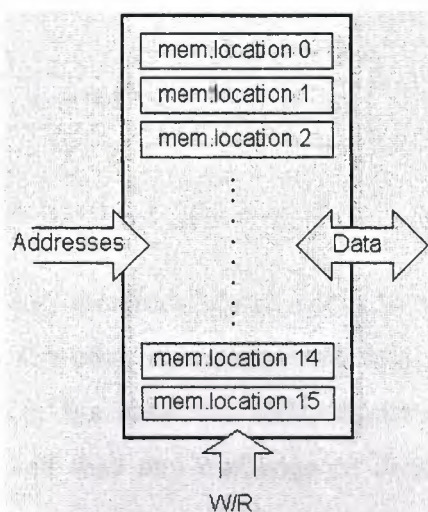


Figure 2.1 Example of simplified model of a memory unit.

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. Besides 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 desired the memory location. Memory is the first element, and we need a few more components for the operation of our microcontroller.

2.4.2 CENTRAL PROCESSING UNIT

Let's add 3 more memory locations to a specific block (see Figure 2.2) 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.

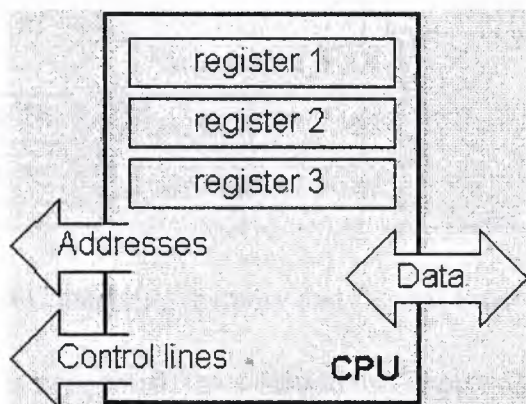


Figure 2.2 Example of simplified central processing unit with three registers.

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.

2.4.3 BUS

A "Bus" physically represents a group of 8, 16, or more wires. As shown in Figure 2.3, there are usually two types of buses: address bus 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.

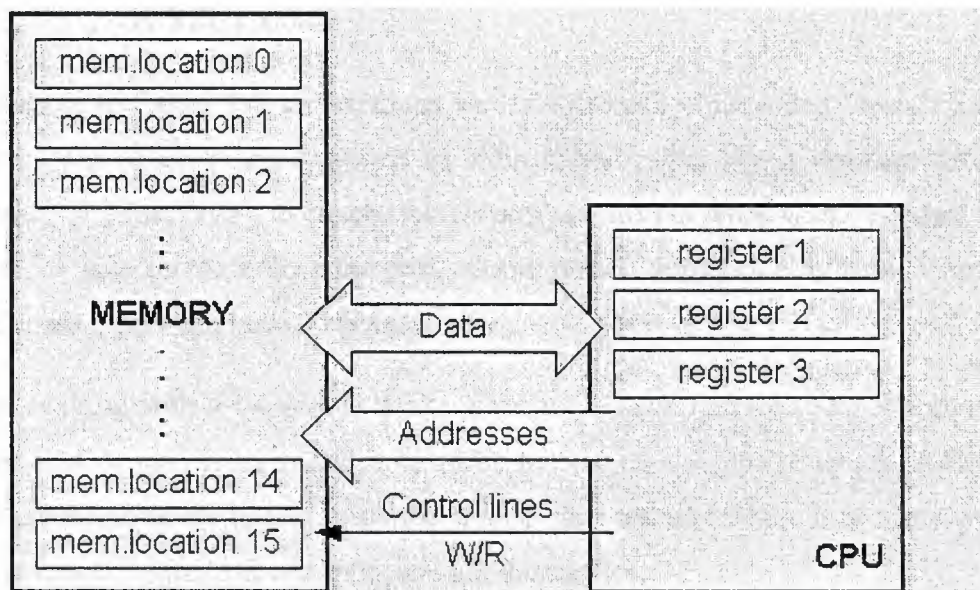


Figure 2.3 Connecting memory and Central Processing Unit.

As far as functionality is concerned, the situation has improved, but a new problem has also appeared: we have a unit that is 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, we can 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 (see Figure 2.4).

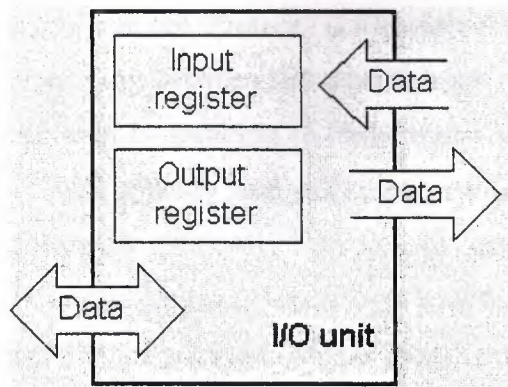


Figure 2.4 Example of a simplified input-output

2.4.4 INPUT-OUTPUT UNIT

In reference to Figure 2.4, the locations we have just added are called "ports". There are several types of ports: input, output or bidirectional 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 the data from the port. Some microcontrollers only have a few ports, while some others can have 30 or more ports.

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. Input-output ports are usually used for parallel data transfer where 8 or more wires are used to transfer data from one device to another device.

2.4.5 SERIAL COMMUNICATION

Beside stated above we have added to the already existing unit the possibility of communication with an outside world. However, this way of communication 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 known as the 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).

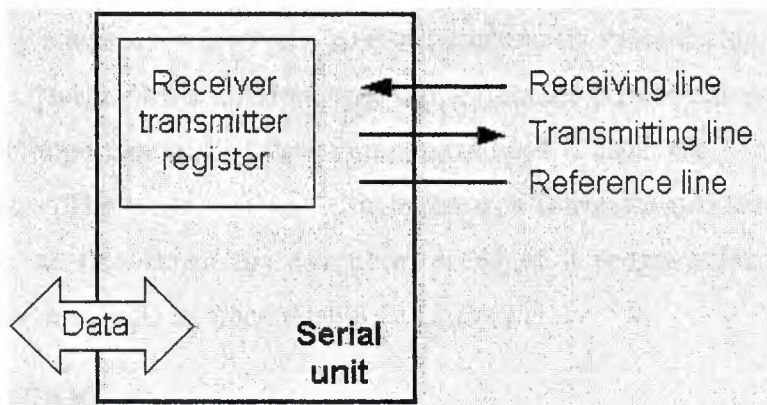


Figure 2.5 Serial unit used to send data, but only by three lines.

As we have separate lines for receiving and sending, it is possible to receive and send data (information) 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.

2.4.6 TIMER UNIT

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

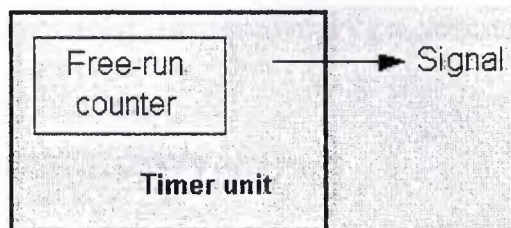


Figure 2.6 Timer unit generates signals in regular time intervals.

However, in order to utilize it in industrial and commercial applications, we need a few additional 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-running counter which is in fact a register whose numeric value increments (or decrements) 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 as it establishes the timing of the microcontroller. The timer unit can also be used to introduce artificial delays to our programs. i.e. to slow-down the execution speed of a program for example when displaying data on a LCD or when flashing a LED.

2.4.7 WATCHDOG

One more thing requiring our attention is the flawless functioning of the microcontroller during its run-time. To overcome the obstacle of some interference which often does occur in industry we need to introduce one more block called watchdog.

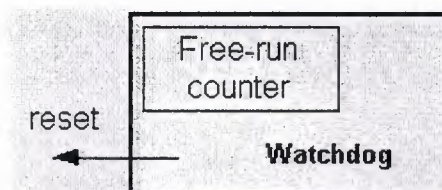


Figure 2.7 Watchdog reset.

This block is in fact another free-run counter (Figure 2.7) 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 any external intervention.

2.4.8 ANALOG TO DIGITAL CONVERTER

As the real-world 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. As shown in Figure 2.8, this task is performed by a block called the analog to digital converter or by an ADC. This block is responsible for converting an information about some analog value to a binary number and to send it to a CPU block so that CPU block can further process it.

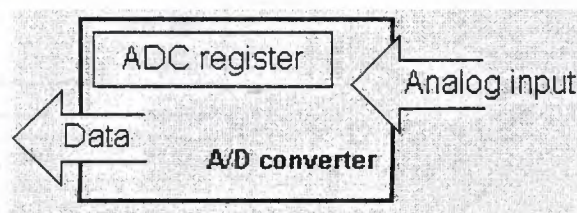


Figure 2.8 Blocks for converting an analogue to a digital form.

Finally, 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. Figures 2.9 and Figure 2.10 below show what a typical microcontroller looks like inside.

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.

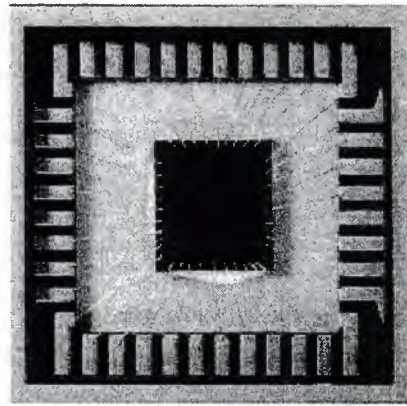


Figure 2.9 Physical configuration of the interior of a microcontroller.

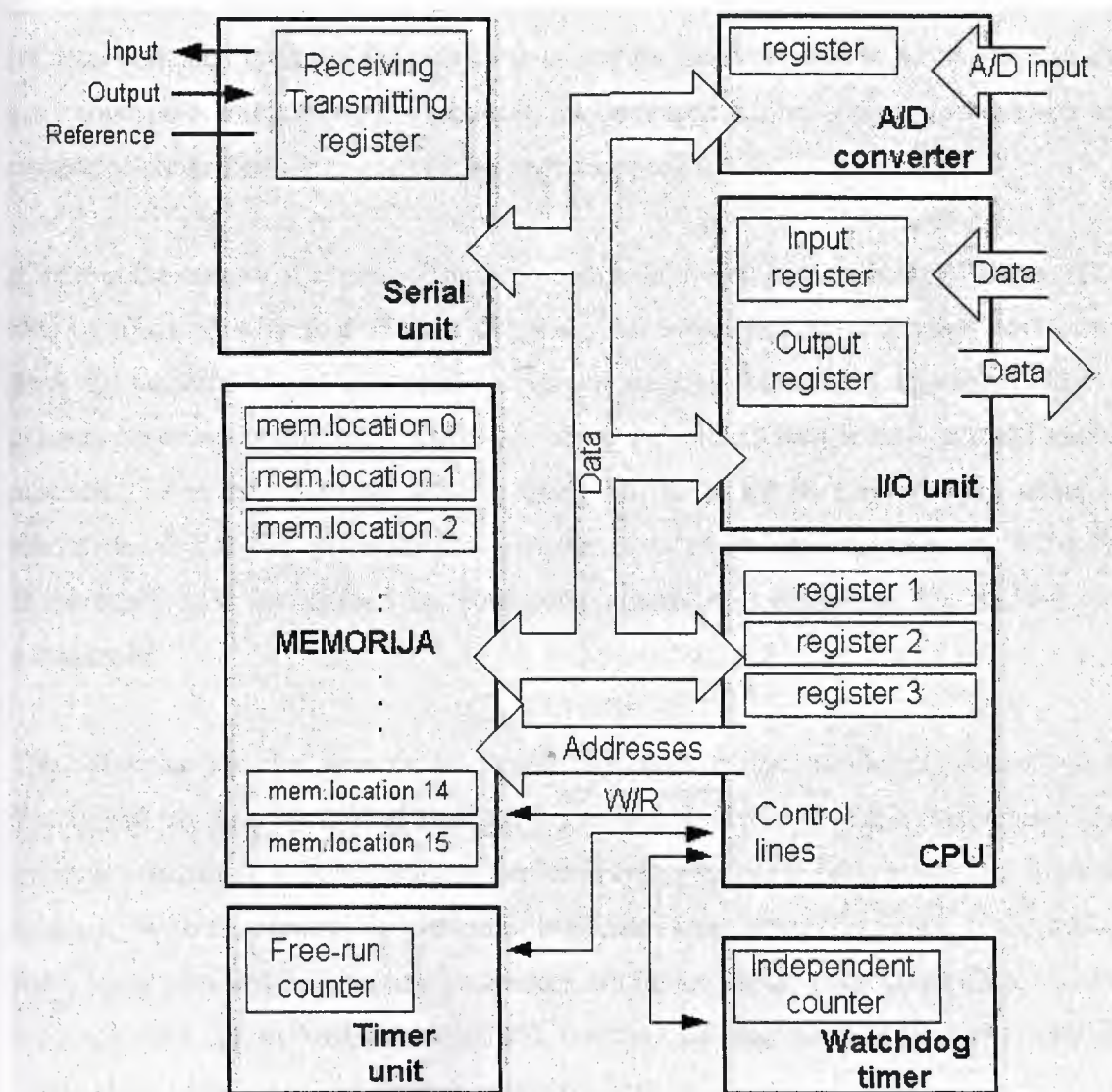


Figure 2.10 Microcontroller outline with its basic elements and internal connections.

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 are called the interface logic. The interface logic could be a simple LED, a 7-segment LED display, a buzzer, a push-button switch, a motor, or some other form of input or output devices.

2.5 EMBEDDED CONTROLLER

An embedded controller [9] can be defined in many ways. An embedded controller is a controller that is embedded in a greater system. An embedded controller is a controller (or computer) that is embedded into some device for some purpose other than to provide general purpose computing. For example, the television remote control unit contains an embedded controller which controls all operations of the unit.

A common example of a general purpose computer would be a typical PC clone. The x86 processor in a typical PC can not really be considered an embedded controller, since the machine is typically used for general purpose computing. However, what is general purpose computing? Take this same PC clone, turn it into a multi-media machine. You have an appliance - much on the order of a microwave oven or television. Is the x86 processor now considered an embedded controller or, is the PC clone itself now considered an embedded controller, controlling the multi-media peripherals?

The difference between an embedded controller and a microcontroller is not much [10]. We might be safe by saying that an embedded controller controls something (for example controlling a device such as a microwave oven, car braking system, or a cruise missile). With the continuing process of high scale integration continuing at a dizzying pace, many standard architecture processors are turning up as microcontrollers. A few such examples are the Motorola 68EC300, Intel 386 EX, and the IBM PowerPC 403GB. These chips could be called super-microcontrollers.

Embedded controllers adhere to a philosophy similar to that of microcontrollers, high integration. By including features necessary for the task at hand, an embedded

controller (processor) can be a powerful yet cost effective solution. However, where a microcontroller is a computer on a chip, an embedded controller might need external components before it is considered a "computer." This is especially true regarding RAM. Since including large amounts of RAM (megabytes) on a processor is not really practical (due to cost and available silicon real estate) and because many embedded controllers are real powerhouses requiring large amounts of RAM, the RAM is often external to the processor.

2.6 MICROCONTROLLER APPLICATIONS

2.6.1 ENVIRONMENTAL APPLICATIONS

Embedded processors and microcontrollers [11] are frequently found in: appliances (microwave oven, refrigerators, television and VCRs, stereos), computers and computer equipment (laser printers, modems, disk drives), automobiles (engine control, diagnostics, climate control), environmental control (greenhouse, factory, home), instrumentation, aerospace, and thousands of other uses. In many items, more than one processor can be found.

Microcontrollers are typically used where processing power is not so important. Although one might find a microwave oven controlled by a Unix system an attractive idea, controlling a microwave oven is easily accomplished with the smallest of microcontrollers.

Embedded processors and microcontrollers are used extensively in robotics. In this application, many specific tasks might be distributed among a large number of controllers in one system.

Communications between each controller and a central, possibly more powerful controller (or micro/mini/mainframe) would enable information to be processed by the central computer, or to be passed around to other controllers in the system.

A special application that microcontrollers are well suited for is data logging. Stick one of these chips out in the middle of a corn field or up in a balloon, and monitor and

record environmental parameters (temperature, humidity, rain, etc). Small size, low power consumption, and flexibility make these devices ideal for unattended data monitoring and recording.

2.6.2 INDUSTRIAL APPLICATIONS

The automotive market is the most important single driving force in the microcontroller market, especially at its high end. Several microcontroller families were developed specifically for automotive applications and were subsequently modified to serve other embedded applications. The automotive market is demanding. Electronics must operate under extreme temperatures and be able to withstand vibration, shock, and EMI. The electronics must be reliable, because a failure that causes an accident can (and does) result in multi-million dollar lawsuits. Reliability standards are high - but because these electronics also compete in the consumer market - they have a low price tag.

2.7 PIC MICROCONTROLLER

PIC (Peripheral Interface Controller) [12] is the integrated circuit which was originally developed to control the peripheral devices, dispersing the function of the main CPU. When compared to a human being, the brain is the main CPU and the PIC shares the part which is equivalent to the autonomic nervous system. PIC has the calculation function and the memory like the CPU, and is controlled by the software.

There are over 150 types of PIC microcontrollers available in the market place. Some devices are small with only 8-pins, some are bigger with 18 or 24-pins and some devices have up to 64-pins. All PIC microcontrollers are RISC type controllers and their instruction sets consists of 33 carefully chosen instructions. The small devices in the family have only a few digital input-output pins and small data and program memories. Larger devices are equipped with larger amounts of memory, timer circuits, interrupt facilities, watchdog timers, and internal USART circuitry for serial communications. Even larger devices have analog-to-digital converter circuits, pulse-width-modulation (PWM) ports, several general purpose timers, several interrupt sources, and much larger data and program memories. But all different models in the family have similar instruction sets and are compatible with each other.

However, the throughput and the memory capacity are not big. It depends on the kind of PIC but the maximum operation clock frequency is about 20 MHz and the memory capacity to write the program is about 1K to 4K words. The clock frequency is related to the speed to read the program and to execute the instruction. Only at the clock frequency, the throughput can not be judged. It changes with the architecture in the processing part. As for the same architecture, the one with the higher clock frequency gives higher throughput. The instruction set of the PIC16F84A microcontroller is composed of 14 bits. It is $1 \times 1,024 \times 14 = 14,336$ bits when converting the 1K words to bits. It is $14,336 / (8 \times 1,024) = 1.75\text{K}$ bytes when converting this to bytes.

The point where the PIC microcontroller is convenient for is that the calculation part, the memory, the input/output part and so on are all incorporated into the same one piece of the integrated circuit.

2.7.1 THE PIC16F84 MICROCONTROLLER

This is the microcontroller used in this thesis. The reason for using the PIC16F84 is because currently this is one of the most popular microcontrollers available in the market, used widely in many commercial and industrial applications. The multi-tasking algorithms developed in the thesis are based on the PIC16F84 microcontroller, but it should be an easy task to modify these algorithms for other members of the PIC family, or for other types of microcontrollers.

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 in size than 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. Figure 2.11 shows the pin configuration of the PIC16F84 microcontroller.

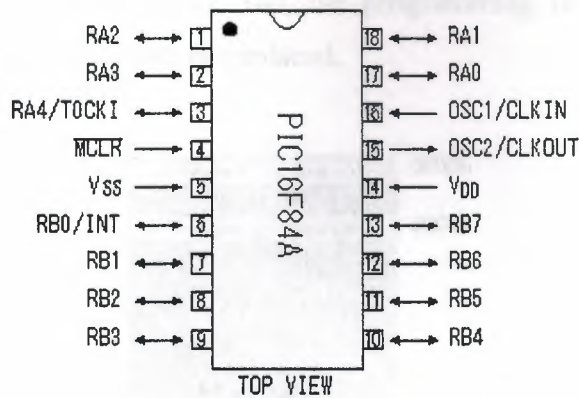


Figure 2.11 Pin configuration of PIC16F84.

Table 2.1: Pin Descriptions

OSC1/CLKIN	Oscillator crystal input. External clock source input.
OSC2/CLKOUT	Oscillator crystal output. Connects to crystal or resonator in crystal oscillator mode.
MCLR(inv)	Master clear (reset) input. Programming voltage input. This pin is an active low reset to the device.
RA0 - RA3	Bi-directional I/O port.
RA4/T0CKI	Bi-directional I/O port. Clock input to the TMR0 timer/counter.
RB0/INT	Bi-directional I/O port. External interrupt pin.
RB1 - RB7	Bi-directional I/O port.
V _{SS}	Ground.
V _{DD}	Positive supply (+2.0V to +5.5V).

The functions of each pin are described in Table 2.1

2.7.1.1 INTERNAL COMPONENTS OF PIC16F84

As we see in the block diagram (Figure 2.14) the PIC 16F84 microcontroller consists of:-

1. Flash Program Memory

The flash memory (Figure 2.12) is used for the memory which stores the user program. One word is composed of 14 bits and 1,024 words (the 1K words) are installed in this microcontroller. Even if the power supply is switched off, the content which is stored in the flash memory do not disappear. The contents of the flash memory can be rewritten

using a suitable programmer device. But, the programming is limited to about 1000 times, after which the device must be replaced.

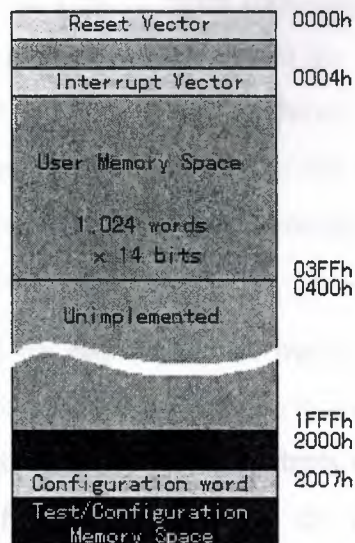


Figure 2.12 Flash program memory.

Reset Vector (0000h)

When the reset is executed by applying power to the microcontroller, or when the reset button (MCLR) is activated, the user program starts from address 0 of the program memory.

Peripheral Interrupt Vector (0004h)

This is the interrupt service routine (ISR) of the microcontroller. When there is time-out interrupt from the timer (TMR0), or when the external interrupt pin (INT0) is activated, the processor jumps to address 4 of the program memory to handle the interrupt.

Configuration word (2007h)

The basic operation of the PIC is specified by this memory location. The enable bit of the Power-up timer, the enable bit of the Watchdog timer, the Oscillator Selection bits can be set. The configuration word can either be set during the programming of the device, or as part of the program code.

2. RAM (Random Access Memory) File Registers

Figure 2.13 shows the structure of the RAM memory. The memory is 80 bytes long and is organized in two banks. The first 12 bytes (00h-0Bh) of each bank are called SFR (Special Function Registers) and are used to record the operating states of the PIC, the conditions of the input/output ports, the other conditions. Each use is decided.

The 68 bytes (0Ch-4Fh) of the 13th byte are called GPR (General Purpose Registers) and are possible to make record the results and the conditions on the way which executes the program temporarily.

The contents which depend on each bank are managed and there are 16 kinds of registers in SFR. But, some part of SFR is common to both banks of memory.

When the power supply is switched off, the contents of RAM are lost. There is not limitation on the number of times to rewrite to the RAM during the running of a program.

SFR Registers

SFR (Special Function Registers) can specify 16 kinds of the registers by the bank changing. The whole memory capacity is the 160 bytes. But, the contents of the left arrow are the same on any bank. As for the part of SFR, the contents change with the bank changing. There are not memories in the gray part.

Address	Bank 0	Bank 1	Address
00h	INDF	←	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	←	82h
03h	STATUS	←	83h
04h	FSR	←	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	Unimplemented	←	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2	89h
0Ah	PCLATH	←	8Ah
0Bh	INTCON	←	8Bh
0Ch - 4Fh	GPR	←	8Ch - CFh

Figure 2.13 Special function registers.

Each SFR has the function as listed in Table 2.2.

Table 2.2: SFR Functions.

INDF	Data memory contents by indirect addressing
TMR0	Timer counter
PCL	Low order 8 bits of the program counter
STATUS	Flag of the calculation result
FSR	Indirect data memory address pointer
PORTA	PORTA DATA I/O
PORTB	PORTB DATA I/O
EEDATA	Data for EEPROM
EEADR	Address for EEPROM
PCLATH	Write buffer for upper 5 bits of the program counter
INTCON	Interruption control
OPTIN_REG	Mode set
TRISA	Mode set for PORTA
TRISB	Mode set for PORTB
EECON1	Control Register for EEPROM
EECON2	Write protection Register for EEPROM

3. EEPROM (Electrically Erasable Programmable Read Only Memory)

This memory is the type which maintains its contents even if the power supply is switched off (i.e. non-volatile). The contents of this memory can be rewritten by the program. The memory capacity is the 64 bytes. As for this memory, the rewriting number of times is limited. It is the about one million time. So, it is not possible to use to store the data on the processing way and so on. It is used to store the data with few change frequencies.

This memory is said to be able to maintain the memorized contents for the about 40 years.

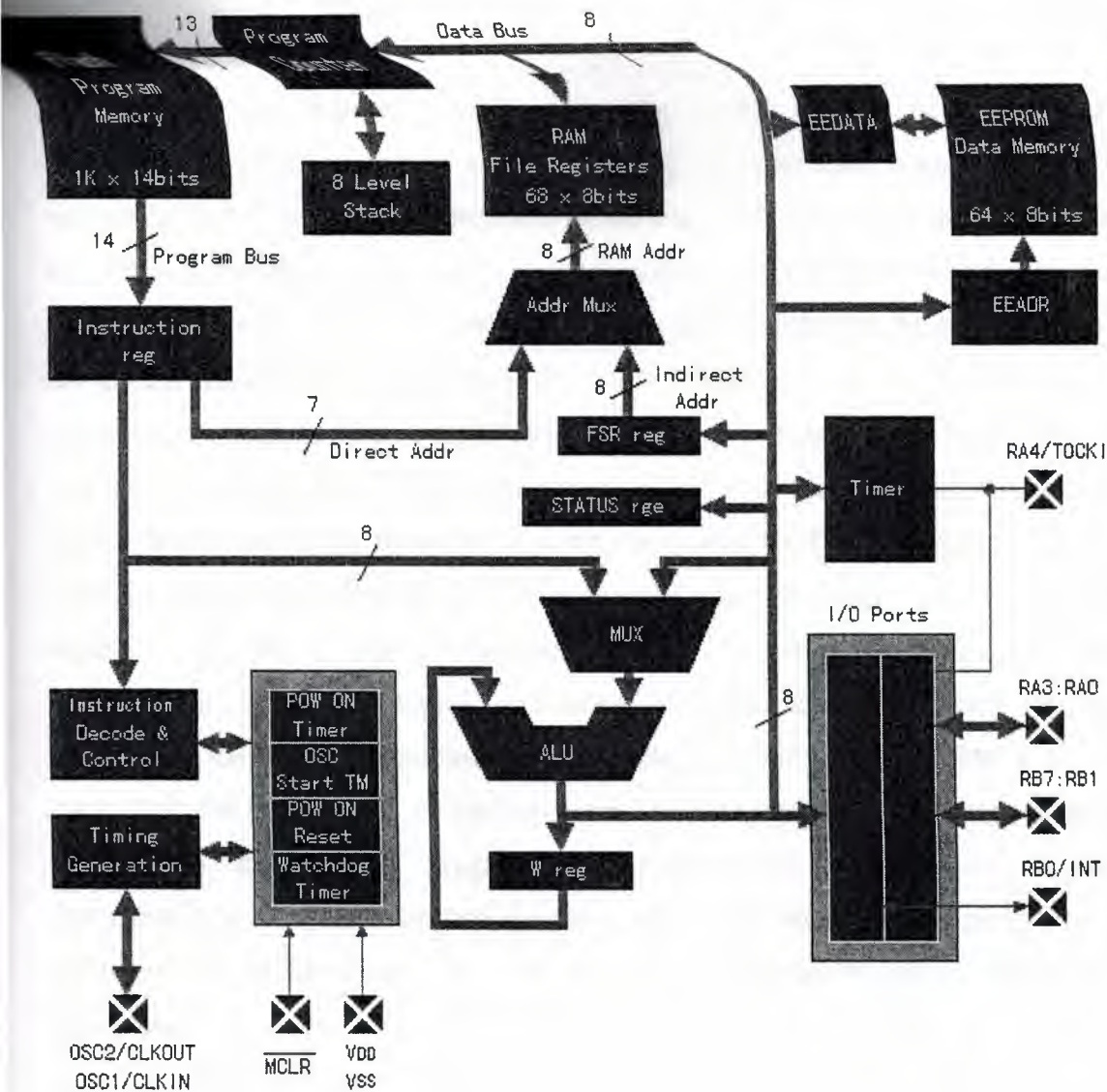


Figure 2.14 Block diagram of PIC16F84A.

4. Program Counter

This is the counter which shows the reading address (Fetch address) of the program which is written into the flash memory. It is basically a 13 bits up counter. Generally, the count increments by one every time an instruction is executed and the position of the following instruction is shown. But, when JUMP is executed, the contents of this counter are rewritten to the jump address.

5. Level Stack

The stack is the memory which stores the return address of the program. For example, when doing the same processing at more than once, it makes the processing in the form of the subroutine. At the end of the subroutine, the instruction of RETURN is written. The memory is saved in making the processing the subroutine. As shown in Figure 2.15, a program which uses a subroutine jumps to the subroutine using the CALL instruction. At this time, the return address is stored in the stack. This operation is sometimes called the PUSH. When the processing by the subroutine ends and the instruction of RETURN is executed, it jumps to the return address which is written at the stack. This operation is sometimes called the POP. If being in this way, even if CALL to the subroutine is written at more than one part, the processing can be returned to the original program which jumped to the subroutine. The fact that there are eight stacks can do the subroutine the eight times at the serial. When doing the subroutine the nine times, the return address has been written at the first stack. Because the contents of the first stack are rewritten, the processing can not be returned to the original program. The eight times can not be exceeded. The subroutine must always return the processing to the called original program using the instruction of RETURN. A JUMP instruction must not be used to return from a subroutine.

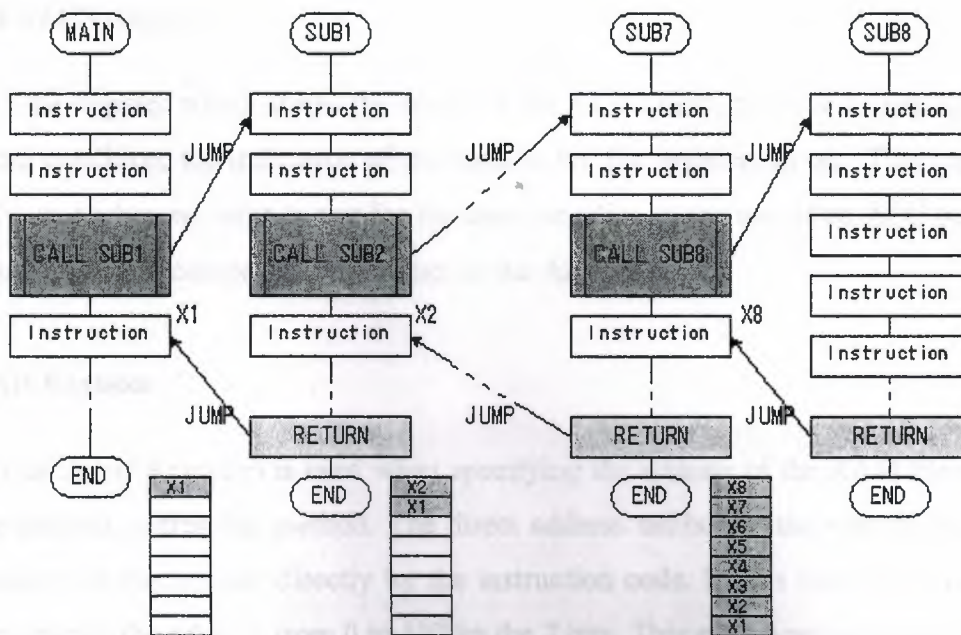


Figure 2.15 CALL instructions.

6. Instruction Register

The instruction of the program which is specified by the program counter is read to this register. This operation is called the FETCH.

7. Instruction Decode & Control

The instruction which was fetched to the instruction register is analyzed here and the operation according to the contents is done by the decode and control logic.

8. Multiplexer and Arithmetic Logic Unit

The calculation operation is done by the Multiplexer and the Arithmetic Logic Unit (ALU). It is not the computer when there are not these.

9. W Register

This is the “work” register, also known as the accumulator register. It is used to keep the calculation result of the ALU temporarily. For the calculation operation, it is the indispensable register. The contents of this register are stored in the various register and are utilized. It is also used for the output control of the input-output port.

10. STATUS Register

This is the register which stores the result of the ALU (Zero, positive or negative), the time-out condition, the indication of the bank of the file register, so on. This register is usually used when we wish to test for the zero condition at the end of an ALU operation (for example, after comparing two values in the ALU).

11. FSR Register

FSR (File Select Register) is used when specifying the address of the RAM file register by the indirect addressing method. The direct address method is the way of specifying the address of the register directly by the instruction code. In this case, the addressing bit can specify the address from 0 to 127 by the 7 bits. This specification range is for the one bank. To change the bank, it is necessary to combine with RP0 bits of the STATUS

register. Because FSR is the 8 bits, it is possible to be specified at once including the bank specification. In PIC16F84, the memory is not installed at the address of 80(50h) to 127(7Fh). It is convenient when using the FSR for the addressing when making the data area which was continued at the file register. The processing is simplified when increment the FSR when doing the writing or the reading continuously.

12. Address Multiplexer

It distinguishes the indirect addressing or the direct addressing.

13. EEDATA

This is the register to use when writing or reading the data to or from the EEPROM memory. Reading or writing to the EEPROM memory is generally a slow process, and the memory location should be checked after a read or a write to confirm that the operation has been completed successfully.

14. EEADR

This is the register which specifies the address of the EEPROM. Because it is composed of 8 bits, the address from 0 to 255 can be specified. In PIC16F84A, the EEPROM is only 64 bytes installed. When setting the data of the EEPROM by the source coding, it specifies 2100h as the memory address. When writing data to the EEPROM by the processing of a program, it is necessary to do the processing to set 55h and AAh to the EECON2 register in the order.

15. Timer

PIC16F84A microcontroller has only one timer (TMR0) which is 8 bits wide. It is in the time-out mode when the count becomes 256 and the T0IF bit of the INTCON register of SFR becomes "1". It is possible to make the interrupt occur when being in the time-out mode. To make the interrupt occur, the GIF bit and the T0IE bit of the INTCON register of SFR must be made "1". The timer interrupt is used frequently in microcontroller applications in order to obtain accurate time delays (e.g. in controlling the bit timing of a serial communication).

16. I/O Ports

The input-output architecture of the PIC16F84 microcontroller is very simple. There are 13 pins with individual direction control. The mode (the input or the output) of each pin can be set by the program. The 13 pins are divided into the two groups. They are the five pins as the A port and the eight pins as the B port. There is limitation on the timing of the control but each of the 13 pins can be controlled. The direction of each port pin can be controlled by setting a bit in special registers. Port A direction is called by a register called TRISA, and Port B direction is controlled by a register called TRISB.

17. Timing Generation

This circuit generates the clock pulse which fixes the operation speed. the oscillation operation is done by putting the capacitors the crystal(or ceramic) oscillator outside. When having the oscillation with the high stability, it uses the crystal. Generally, the circuit becomes simple when using the resonator which incorporated the ceramic and the capacitors into the one module. The clock pulse can be inputted from outside, too. The PIC16F84A execute the one instruction (the 1 cycle) by the four clock pulses using the pipeline architecture. But, in case of the JUMP to change the program address, the 2 cycles are necessary. In the execution time of usual instruction, it is the 200 nanoseconds because the pulse period of 20MHz is $1/(20\text{MHz}) = 50$ nanoseconds. The 5,000,000 instructions can be executed within the 1 second.

18. Initialization circuits

The PIC16F84A microcontroller has various initialization circuits as summarized in Table 2.3.



Table 2.3 Initialization circuits

POWON Timer	This is the timer to limit the operation until the voltage is stable in case of the turning on.
OSC Start Timer	This is the timer to limit the operation until the clock is stable in case of the turning on.
POWON Reset	This initializes the inner circuit of the PIC in case of the turning on.
Watchdog Timer	This is the timer to watch over the normal operation of the software of PIC. This timer must be regularly cleared by the software. When the timer does in the time-out, the PIC returns to the condition immediately after the turning on. This timer is used to recover the extraordinary operation when the software has the defect (the bug). Even if it is initialized, the bug doesn't pass away.

2.7.2 The PIC16F877 MICROCONTROLLER

PIC16F877 is another popular PIC microcontroller with more input-output pins, more program and data memories, analog-to-digital converter circuit, and USART based serial communication circuit. The architecture of this microcontroller will be discussed very briefly in this section.

The important features [13] specific to picking a microprocessor are 1) power, 2) memory size, 3) fast reprogrammability, 4) A/D channels, and 5) a 5 volt operating supply. Table 2.4 compares some of the popular microprocessors on the market. Though not an 8 bit processor, the Strong Thumb processor was included both because of its commercial popularity and low power consumption. At the time the Atmel AT90LS8535 offered the best performance even though newer designs like the Microchip PIC16F877 offered lower power consumption and greater functionality.

The PIC16F877 is a high-performance FLASH microcontroller that provides engineers with the highest design flexibility possible. In addition to 8192x14 words of FLASH program memory, 256 data memory bytes, and 368 bytes of user RAM, PIC16F877 also features an integrated 8-channel 10-bit Analogue-to-Digital converter. Peripherals include two 8-bit timers, one 16-bit timer, a Watchdog timer, Brown-Out-Reset (BOR), In-Circuit-Serial Programming™, RS-485 type UART for multi-drop data acquisition

applications, and I²C™ or SPI™ communications capability for peripheral expansion. Precision timing interfaces are accommodated through two CCP modules and two PWM modules.

Table 2.4: Comparison of some popular microcontrollers

	Atmel AVR AT90LS8535	Microchip PIC16F877 (preliminary)	MC68H(R)C 908JL3	Atmel AT91M404000 16/32 bit Strong Thumb
Flash Memory	4K	8Kx14	4K	external memory
Endurance	1k	1k	10k	N/A
MIPS/mA	1.25 (min)	1.66 (preliminary)	0.1 (typical)	0.6 MIPS/mA (1.35 mA static current)
A/D channels	8 (10 bit)	8 (10 bit)	12 (8 bit)	0
In-application programming (IAP)	No	Yes	Yes	Yes
Operating voltage	2.7-5.5V	2.0 – 5.5V	2.7-3.3V	2.7V-3.6V
I/O pins	35	40	23	100

Further information on the PIC family of microcontrollers can be obtained from the manufacturer's web site and data sheets [12].

2.8 SUMMARY

Today, microcontrollers are used almost in all electronic system applications either as stand-alone units, or as embedded controllers.

A microcontroller is a computer on a chip with its CPU, memory, and input-output circuitry. There are many types of microcontrollers, manufactured by various companies, with different processing powers.

This chapter has described the architecture and the basic features of the popular PIC16F84 microcontroller which is used in the following sections of this thesis.

3. MICROCONTROLLER SYSTEM DEVELOPMENT CYCLE

3.1 OVERVIEW

The high-level programming languages make the program development a relatively easy task. C, PASCAL and BASIC are some of the most commonly used high-level languages when programming microcontrollers. With high-level languages the development time is shortened and the possibility of making errors is reduced. As a result, the development cycle is much shorter when a high-level language is used. In this thesis the popular PIC BASIC language and the native PIC assembler languages are used during the software development of the multitasking algorithms.

3.2 BASIC ELEMENTS OF PIC BASIC LANGUAGE

A programming language [14] is understood as a set of commands and rules according to which we write the program and therefore we distinguish various programming languages such as BASIC, C, PASCAL etc. For the BASIC programming language the existing literature is pretty extensive and as a result of this, most of the attention in this chapter will be dedicated to the part concretely dealing with the programming of microcontrollers.

A program consists of a sequence of commands of language that our microcontroller executes one after another. The structure of a BASIC program is explained with more details in this chapter.

3.2.1 IDENTIFIERS

An identifier represents the name of some PIC BASIC element. Identifiers are used in PIC BASIC in order to assign program lines and the names of various symbols. An identifier itself could be any string of letters, numbers or even dashes with the limit that it is not allowed to begin an identifier with a number. In BASIC, identifiers do not distinguish small and capital letters, so that the strings TASTER and Taster are treated the same way. The maximum length for such strings is 32 characters. An example is given below.

3.2.2 LABELS

Labels represent textual signs for some programming lines or respectively some of its fragments on which the program can jump through some of the instructions used to change the program flow. It is obligatory to end the label with a colon. Contrary to many old BASIC versions, PIC BASIC does not allow numerical values as labels. i.e. all labels must start with a character and terminate with a colon.

```
symbol Taster = PORTA.0
symbol LED_0 = PORTB.0
```

```
B0 var byte
```

```
Main:           ' Label Main
    B0 = 0
    button Set,0,255,0,B0,1,LED_toggle
    goto Main
```

```
LED_toggle:     ' Label LED_toggle
    toggle LED_0
    goto Main
end
```

3.2.3 CONSTANTS

Constant declarations allow constant values to be assigned to identifiers. For example the constant minute has the value of 60 seconds, bearing the recollection to the number of seconds in a minute. Written at whatever program position, minute will be interpreted by compiler as if it had been written 60. There are two very important reasons for such habit in program writing. The first one is the programmers wish to be more manifest. Good visibility is achieved by giving to the variables and constants those names that could be associated with the very function they assume within the program. On the other hand, the bigger flexibility of the program is obtained as well. It is for an example so that if it becomes necessary in some future work to use the same code but with a change value of the

constant, it is enough make a change in the part for declaration by performing search and replace throughout the program.

```
minute con 60           ' No. of seconds in a minute
if seconds < minute then minute = minute + 1 ' If the number of seconds is different
                                           ' from 60, raise the variable minutes
```

Constants can equally be written in decimal, hexadecimal and binary form. Decimal constants are written without any prefix. Hexadecimal constants start with a Dollar sign \$ and binary constant start with percent sign %. To make the programming easier, single letters are converted into their ASCII counterparts. The sign constants must be placed into the inverted commas and they contain only one letter as a rule (in adverse case they are string constants).

```
56           ' 56 decimal
$0F          ' 15 hexadecimal
%10001100    ' 140 binary
"A"          ' ASCII value for decimal 65
"d"          ' ASCII value for decimal 100
```

3.2.4 VARIABLES

Variables serve for temporary storing of data and results of various arithmetic and logical operations. Variables are stored in the microcontrollers RAM locations, which means that the total numbers of the variables that can be used depend on the size of available RAM.

Variable definition is achieved with the formal word *var* at the beginning of the program. PIC BASIC supports variables like bit, byte and word. Variable type is selected with reference to the expected value that this same variable can assume in the course of the program run. Therefore the variable of the bit type can take value of 0 or 1, the variable of the byte values from 0 to 256 and finally, word from 0 to 65535.

```
Fleg var bit           ' Fleg is a variable of the type bit
B0 var byte           ' B0 is a variable of the type byte
W0 var word           ' W0 is a variable of the type word
B0 var W0.byte0       ' B0 is a first byte of the word W0
B1 var W0.byte1       ' B1 is a second byte of the word W0
```

3.2.5 SEQUENCES

Sequences of the variables are defined in a similar way as we have done with the variables.

The number of the elements of the sequence is given through value between "[]". Each element of the sequence is accessible by an index. Index starts with zero. When we come to define the number of the elements of the sequence one must always have in mind that the number of locations in RAM memory on which we intend to store variables is finite. Table 3.1 shows the maximum number of the elements of various types.

Table 3.1: The size of the sequence

The size of the sequence	
Element of the sequence	Maximal number of elements
BIT	256
BYTE	96*
WORD	48*

* Depends on microcontroller

Some examples of sequences are given below.

Sequence1 var byte[10] ' the sequence of 10 elements of the type byte

Sequence1 [0] represents the first element of the sequence and sequence1 [9] the last element of the sequence "sequence1".

Sequence2 var byte[8] ' the sequence of 8 elements of the type byte

Sequence2 [0] represents the first element of the sequence and sequence2 [7] the last element of the sequence "sequence2".

3.2.6 MODIFIERS

By means of a modifier it is possible to introduce a new name for a variable already defined. This direction is used relatively rarely but it ought to be mentioned for the sake of completeness. It is used in an identical way as a direction for the definition of the variables. Introduction of a new name is effectuated through the official word **var**.

```
ADCRestult var word
HigherByte var ADCresult.byte0      ' The new name for the higher byte of the
                                     ' word ADCresult
```

3.2.7 SYMBOLS

Symbols are granted the function exactly the same as direction for modifying variables, i.e. they serve for assigning the new names to the variables and constants. Symbols are introduced for the compatibility of the programs written for Basic Stamp and cannot be used for introducing variables.

```
symbol Taster = PORTA.0      ' Taster is a new name for RA0
symbol LED_0 = PORTB.0      ' LED_0 is a new name for RB0
```

3.2.8 DIRECTION INCLUDE

Direction **INCLUDE** serves for inserting files or file segments to a BASIC program. In this manner it is rendered possible to store some general definitions of variables or subroutines that are being executed as parts of several different programs. The effect achieved is the same as if at the location on which is placed the direction **INCLUDE** simultaneously copied the contents of whole file.

In the following example, file "modedefs.bas" is included in the main program.

Include "modedefs.bas"

' The transfer modes that use the
' commands SERIN and SEROUT

symbol SO = PORTA.3

symbol SI = PORTB.0

B0 var byte

Loop:

serin SI,T2400,B0

serout SO,T2400,[B0]

goto Loop

end

3.2.9 COMMENTS

In the course of program writing it is generally recommended to use comments even if it may be self-evident what the main purpose of the program is. Although it may well seem as a sheer waste of time, it may play later a crucial role (comments don't occupy an additional memory space in the memory of a microcontroller). Comments should give useful instructions about all that the program is doing. Comment as Set Pin0 to 1 simply explains the syntax of the language but fails to pinpoint the purpose of the act. Something of a sort Turn the Relay on may prove itself to be much more useful.

At the beginning of the program the aim of the program should be described together with the names of the authors and when it was written. Stipulating the information concerning revision and the exact date may be useful too. Even every concrete statement about connection to each pin can be crucial in an effort to memorize the very hardware for which this program was designed to operate.

symbol LED = PORTB.0	' LED diode is connected to RB0
Main:	' The beginning of the program
LED = 1	' Turn on LED
Pause 500	' Pause 500 mS
LED = 0	' Turn off LED
Pause 500	' Pause 500 mS
goto Main	' Jump to the beginning of program
end	' End of the program

3.2.10 PROGRAMMING LINE WITH MORE INSTRUCTIONS

Compactness and better visibility of a program can be achieved by logically grouping instructions by using ":". In that way the block of instructions can be placed all in a single line, while instructions remain mutually separated with ":".

B2 = B0

B0 = B1

B1 = B2

The three upper instructions can be written in a single row as:

B2 = B0 : B0 = B1 : B1 = B2

3.2.11 TRANSFER OF INSTRUCTION INTO ANOTHER LINE

In case that an instruction is very long and can not be fitted into a single line, it must be continued to the next line. The character "_" is used at the end of a line to mark the continuation of the line.

lookup KeyPress,["1","4","7","*", "2","5","8","0","3","6","9","#","N"]

3.2.12 DEFINE

Instructions of the PIC BASIC language can have some parameters from which depend the exact way the instructions are executed. Those parameters assume some predefined values

that appear in most of the cases. A frequency of an oscillator is a good example for that. If not otherwise stated the tact of the oscillator is taken by default as 4MHz. In case that the used oscillator is of a different frequency from 4MHz it is necessary using the DEFINE direction to specify that frequency and communicate it to all the programs that contain instructions depending on the clock of the microcontroller. One such instruction is for the serial transfer. In case that the instructions DEFINE is omitted and in gear is 8Mhz instead of 4Mhz oscillator, all the instructions that depend on the tact of microcontroller will be executed 2 times quicker. For instance, if the parameter of the speed of transfer amounts to 9600 bauds by using SERIN instruction, the data transfer would be effectuated at the speed 19200. In the same way the instruction pause 1000 the delay realized would be 0.5s instead of 1.0s. It is also possible similarly to upgrade the resolution of the instructions. What is next is the review of the usage for DEFINE direction in case of adjusting of parameters explained within each particular instruction as shown in Table 3.2.

Table 3.2: The use of a direction DEFINE

The use of a direction DEFINE		
parameter	description	instruction on which it acts
I2C_HOLD 1	pause I2C transfer while the tact is on a low level	I2COUT, I2COUT
I2C_INTERNAL 1	internal EEPROM in series 16Cxxx and 12Cxxx of the PIC microcontroller	I2COUT, I2COUT
I2C_SCLOUT 1	serial tact is a bipolar at the place of an open collector	I2CWRITE, I2CREAD
I2C_SLOW 1	for the tact > BMHz OSC with the devices of a standard velocity	I2CWRITE, I2CREAD
LCD_DREG PORTD	LCD data port	LCDOUT, LCDIN
LCD_DBIT 0	Initial bit of a data 0 or 4	LCDOUT, LCDIN

LCD_RSREG PORTD	RS (Register select) port	LCDOUT, LCDIN
LCD_RSBIT 4	RS (Register select) pin	LCDOUT, LCDIN
LCD_EREG PORTD	enable port	LCDOUT, LCDIN
LCD_EBIT 3	enable bit	LCDOUT, LCDIN
LCD_RWREG PORTD	read/write port	LCDOUT, LCDIN
LCD_RWBIT 2	read/write bit	LCDOUT, LCDIN
LCD_LINES 2	No of LCD lines	LCDOUT, LCDIN
LCD_INSTRUCTIONUS 2000	the time of delay of instruction in microseconds (us)	LCDOUT, LCDIN
LCD_DATAUS 50	the time of delay of data in microseconds	LCDOUT, LCDIN
OSC 4	tact of the oscillator in MHz: 3(3.58) 4 8 10 12 16 20 25 32 33 40	all instructions of the serial transfer and next pause
OSCCAL_1K 1	setting of OSCCAL for PIC12C671/CE673 microcontrollers	
OSCCAL_2K 1	the number of data bits	
SER2_BITS 8	the slowing of the tact of transfer	SHIFTOUT, SHIFTIN
SHIFT_PAUSEUS 50	instruction LFSR in 18Cxxx microcontrollers	LFSR
BUTTON_PAUSE 10		BUTTON
CHAR_PACING 1000		SEROUT, SERIN
HSER_BAUD 2400		HSEROUT, HSERIN
HSER_SPBRG 25		HSEROUT, HSERIN

HSER_RCSTA 90h		HSEROUT, HSERIN
HSRE_TXSTA 20h		HSEROUT, HSERIN
HSER_EVEN 1		HSEROUT, HSERIN
HSER_ODD 1		HSEROUT, HSERIN

3.2.13 DISABLE

Before entering the interrupt routine, it is necessary to switch off the interrupts in order to avoid any new interruption in the course of data processing. The interruptions are forbidden in a manner that the instruction "DISABLE" reset the bit GIE in the register INTCON.

```

    Disable    ' Forbid the interruptions
ISR:          ' Start of an interruption routine

    .
    .
    .          ' The end of the interruption routine
    Resume
    Enable

```

3.2.14 ENABLE

In the course of execution of the interruption routine, the interrupts must be forbidden by resetting the bit GIE in the INTCON register. When the interruption processing is finished, the interruptions must be allowed once again with the instruction "ENABLE".

```

    Disable
ISR:          ' Start of the interruption routine

    .
    .
    .          ' The end of the interruption routine
    Resume
    Enable    ' Allow interruptions

```


3.2.15 ON INTERRUPT

The "On interrupt" label indicates the start of the program segment where the interrupts are handled. In the following example, the interrupt service routine is called "ISR" and the program jumps to this label when an external or an internal interrupt occurs.

```
On interrupt ISR ' The interruption routine starts from the label ISR
Main:           ' Main program
    goto Main
    Disable
ISR:            ' Start of the interruption routine
    .
    .
    ' The end of the interruption routine
    Resume
    Enable
```

3.2.16 RESUME

This statement is used to resume the program execution at the end of an interrupt service routine. i.e. return from the interrupt routine to the main program.

```
    Disable
ISR:            ' Start of the interruption routine
    |
    |
    |
    ' End of the interruption routine
    Resume     ' Exit from the interruption routine
    Enable
```

3.3 PIC BASIC COMPILER

The compiler program runs on PC and its task is to translate the original BASIC code into the language of 0 and 1 understandable to the microcontroller. The process of translation of a BASIC program into a HEX code is shown on the Figure 3.1 below. The program written in PIC BASIC and registered as a file **Program.bas** is converted into an assembler code (**Program.asm**). So obtained assembler code is further translated into executive HEX code

which is written to the microcontroller memory by a programmer. (Programmer is a device used for transferring HEX files from PC to the microcontroller memory)

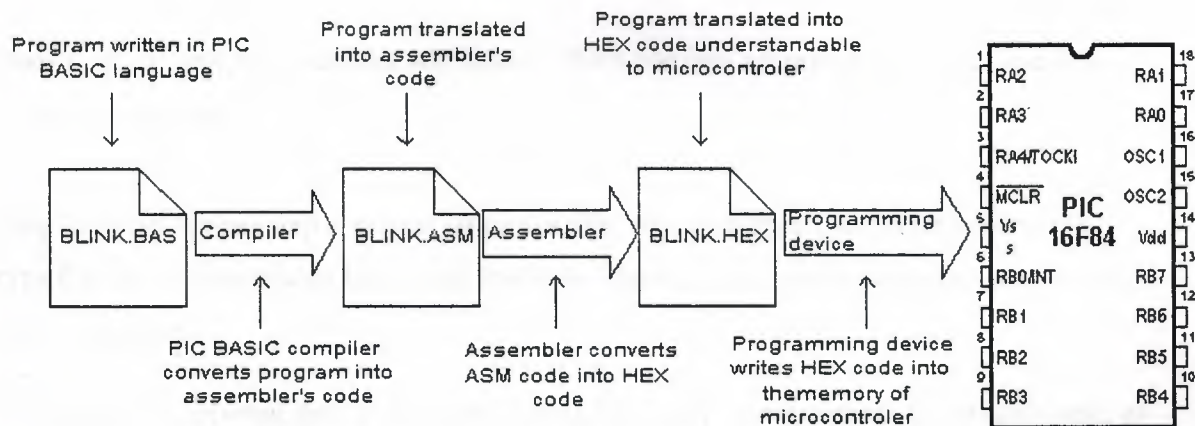


Figure 3.1 The PIC BASIC compiler.

3.4 WRITING AND COMPILATION OF A BASIC PROGRAM

The first step in the writing of a program code is to use a text editor (for example, the Windows Notepad text editor). Every written code must be saved on a single file with the ending .BAS exclusively as ASCII text. An example of one simple BASIC program - BLINK.BAS is given.

```
' Example of a program where the LED diode connected on
' PORT B pin 7 switches on and off every 0.5 seconds

Loop:
    High PORTB.7      ' Switch on LED on pin 7 of port B
    Pause 500         ' 0.5 sec pause

    Low PORTB.7       ' Switch off LED on pin 7 of port B
    Pause 500         ' 0.5 sec pause

    Goto loop         ' Go back to Loop

End                  ' End of program
```

When the original BASIC program is finished and saved as a single file with .BAS ending it is necessary to start PIC BASIC compiler. The compiling procedure takes place in two consecutive steps.

Step 1. In the first step compiler will convert BAS file into assembler code and save it as BLINK.ASM file.

Step 2. In the second step compiler automatically calls assembler, which converts ASM - type file into an executable HEX code ready for reading into the programming memory of a microcontroller.

The transition between first and second step is for a user - programmer an invisible one, as everything happens completely automatically and is thereby wrapped up as an indivisible process. In case of a syntax error of a program code, the compilation will not be successful and HEX file will not be created at all. Errors must then be corrected in original BAS file and repeat the whole compilation process. The best tactics is to write and test small parts of the program, than write one gigantic of 1000 lines or more and only then embark on error finding.

After the program is compiled, it can be simulated on a PC using a simulator program. The simulation process does not require any hardware and enables the programmer to simulate the program operation by running the program on a PC in single step mode. Simulation is very helpful as it enables the programmer to test the program and remove any possible errors before implements the program on the target hardware.

3.5 LOADING A PROGRAM INTO THE MICROCONTROLLER MEMORY

As a result of a successful compilation of a PIC BASIC program the following files will be created.

- BLINK.ASM - assembler file
- BLINK.LST - program listing

- BLINK.MAC - file with macros
- BLINK.HEX - executable file which is written into the programming memory

File with the HEX ending is in effect the program that is written into the programming memory of a microcontroller. The programming device with accessory software installed on the PC is used for this operation. Programming device is a contrivance in charge of writing physical contents of a HEX file into the internal memory of a microcontroller. The PC software reads HEX file and sends to the programming device the information about an exact location onto which a certain value is to be inscribed in the programming memory. PIC BASIC creates HEX file in a standard 8-bit Merged Intel HEX format accepted by the vast majority of the programming software. In the text below the contents of file BLINK.HEX is given (this file is in Intel format, where a checksum is used at the end of every line of code).

```
:100000002828A301A200FF30A207031CA307031C9A
:1000100023280330A100DF300F200328A101E83E90
:10002000A000A109FC30031C1828A00703181528FC
:10003000A0076400A10F152820181E28A01C222844
:1000400000002228080083130313831264000800B1
:1000500006148316061083120130A300F430022028
:1000600006108316061083120130A300F43002201C
:0600700028286300392876
:02400E00753DFE
:00000001FF
```

Besides reading of a program code into the programming memory, the programming device serves to set the configuration of a microcontroller. Here belongs the type of the oscillator, protection of the memory against reading, switching on of a watchdog timer etc. The connection between PC, programming device and the microcontroller is shown in Figure 3.2.

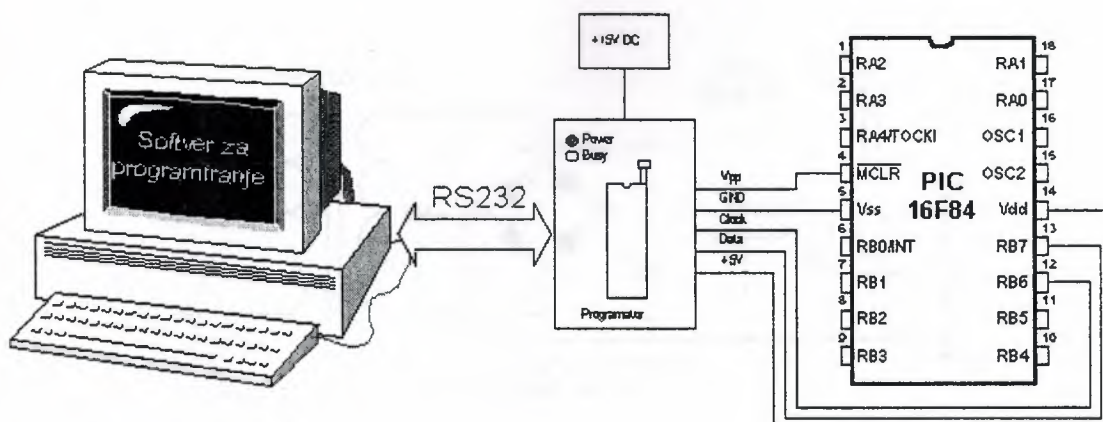


Figure 3.2: The connection between PC, programming device and the microcontroller.

The programming software is used exclusively for the communication with the programming device and is not suitable for any code writing. The one comprising text editor, software for programming microcontroller and possibly the simulator as an entity bears the name IDE i.e. Integrated Development Environment. One such environment is a Microchip's software package MPLAB [12]. MPLAB is a complete microcontroller development package, including an editor, a simulator, a librarian, and an assembler. The package is distributed free of charge and can easily be downloaded from the company's web-site. MPLAB supports all of the company's over 200 models of microcontrollers.

3.6 RUNNING YOUR PROGRAM

For correct operation of a microcontroller, i.e. correct running of a program it is necessary to assure the supply of the microcontroller, oscillator and the reset circuit. The supply of the microcontroller can be organized with a simple regulator (e.g. LM7805) as shown in the Figure 3.3 below. The circuit consists of a transformer which converts the 220V mains voltage to 9V. Then a bridge rectifier circuit is converts the signal into full-wave regulated signal. This signal is the applied to the LM7805 regulator which generates the required +5V voltage for the microcontroller.

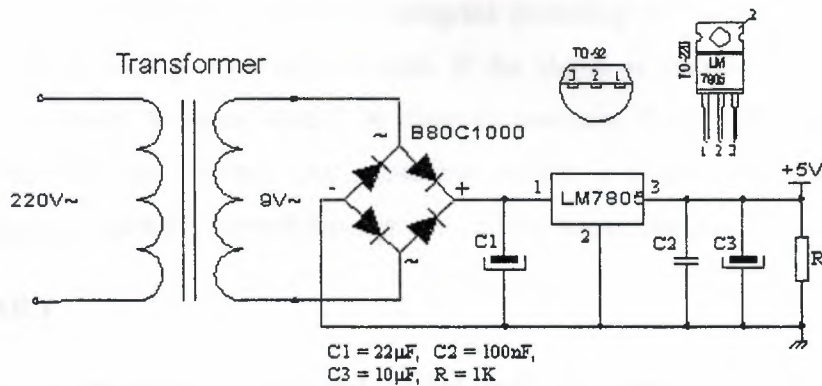


Figure 3.3: LM7805 regulator circuit.

The oscillator of the microcontroller can be a 4MHz crystal and two small 22pF capacitors or a ceramic resonator of the same frequency (ceramic resonator already contains the mentioned capacitors, but contrary to the oscillator has three termination instead of only two). The speed at which the microcontroller operates i.e. the speed at which the program runs depends heavily on this frequency of an oscillator. In the course of an application development the easiest to do is to use the internal reset circuit in a manner that MCLR pin is connected to +5V through a 10K resistor. In the sequence of text the scheme of a rectifier with circuit of LM7805 which gives the output of stable +5V, as well as the minimal configuration relevant for the operation of a PIC microcontroller.

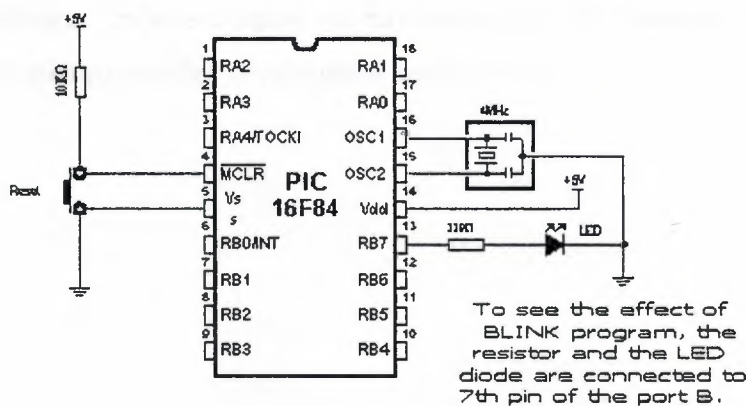


Figure 3.4: Switching on LED.

Note:-Minimal hardware configuration necessary for the operation of PIC microcontroller.

After the supply is brought to the circuit designed according to the Figure 3.4, the LED diode should be twinkling once each second. If the signal is completely missing (LED diode doesn't twinkle), a check should be done to ascertain if the +5V is present at the microcontroller VDD pin. Notice that a limiting resistor is used in series with the LED. This is necessary to limit the current through the LED to about 10mA.

3.7 SUMMARY

Programming is becoming easier and easier but also very memory hungry. As microcontrollers become more powerful and have larger amounts of memory on board, programming will become easier and simpler. For the moment the most efficient way is to use the assembly language in time-critical sections of the program and to use a high-level language in other sections of the program. Nowadays, most programmers use a high-level language such as Basic or C when programming the microcontrollers. These languages are easier to develop and also easier to maintain. This means that if for some reason the microcontroller or the program needs to be changed, the program can easily be converted to run on the new microcontroller.

Development of a microcontroller project requires several hardware and software development tools like: text editor, a high-level language compiler (e.g. PIC Basic), or a native PIC assembler, simulator software, chip programmer hardware, and chip programmer software. In this chapter we have discussed the features of the PIC Basic language and the microcontroller development environment.

4. MULTITASKING APPLICATION ON PIC MICROCONTROLLERS

4.1 OVERVIEW

The principles of multitasking has been described in detail in Chapter 1. In this Chapter the author has developed several single tasking and multitasking algorithms for the PIC microcontrollers. A simple hardware has been developed by the author and some of the multitasking algorithms have been tested using this hardware. In addition, a microcontroller simulator has been used to test and evaluate some of the multitasking algorithms. Both PIC Basic and the native assembler language of the PIC microcontrollers have been used in the examples.

4.2 PROGRAMMING LANGUAGES OF PIC MICROCONTROLLERS

Programming of a PIC microcontroller can be done in several languages and Assembler, C and Basic 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.

Programs in C language are easier to develop, easier to understand, but are slower in execution compared to assembler programs. Basic is the easiest one to learn and its instructions are simple, 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 load the program into the program memory of a microcontroller and then start the system. But before using a microcontroller we need to add a few more external components necessary for the operation of a microcontroller. First we must give life to a microcontroller by connecting it to a power supply (power needed for operation of all electronic instruments) and an oscillator whose role is similar to the role that heart plays in a human body. Based on its clocks microcontroller executes instructions

of a user program. As soon as it receives supply, the microcontroller will perform a small check up on itself, look up the beginning of the program and start executing it. Execution normally starts from address zero of the program memory.

4.3 SINGLE TASKS ON PIC MICROCONTROLLER USING PIC BASIC

In this section some single tasking applications are given which demonstrate the principles of program development and running on a PIC microcontroller. These examples are based on the popular PIC Basic programming language described in earlier chapters. A PIC16F84 microcontroller has been used in these examples since it is very easy to program and use this microcontroller.

4.3.1 LED DIODE EXAMPLE

One of the most frequently used components in electronics is the LED diode (LED stands for Light Emitting Diode). Some common LED diode features include: size, shape, color, working voltage (voltage across the diode) U_d and electric current I_d (current through the diode). LED diodes can have round, rectangular or triangular shapes, although manufacturers of these components can produce any needed shape by order. Size i.e. diameter of round LED diodes ranges from 3 to 12 mm, with 3 or 5 mm sizes most commonly used. Color of emitting light can be red, yellow, green, orange, blue, etc. Working voltage i.e. necessary for LED diode to emit light is 1.7V for red, 2.1V for green and 2.3 for orange color. This voltage can be higher depending on the manufacturer. Normal current I_d through a diode is 10 mA, while maximal current reaches 25 mA. High current consumption can present problem to devices with battery power supply, so in that case low current LED diode ($I_d \sim 1-2$ mA) should be used. For LED diodes to emit light with maximum capacity it is necessary to connect it properly (see Figure 4.1), or it might get damaged by excessive voltage.

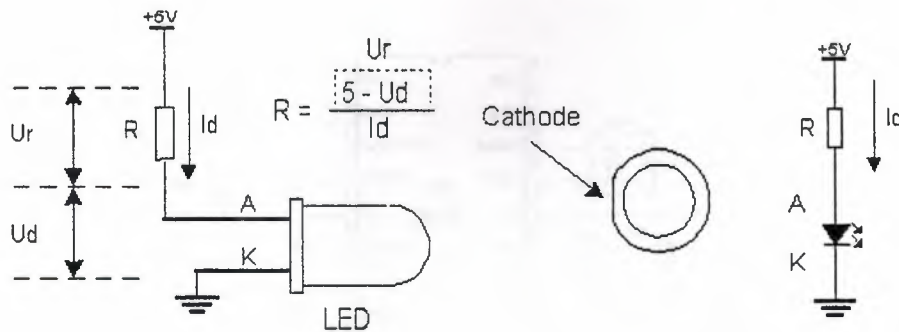


Figure 4.1: Diagram of LED diode.

The positive pole is connected to anode, while ground is connected to cathode. For matter of differentiating the two, cathode is marked by mark on casing and shorter pin. Diode will emit light only if current flows from anode to cathode; in the other case there will be no current. Resistor is added serial to LED diode, limiting the maximal current through diode and protecting it from damage. Resistor value can be calculated from the equation on the picture above, where U_r represents voltage on resistor. For +5V power supply and 10 mA current resistor used should have value of 330Ω .

LED diode can be connected to microcontroller in two different ways. One way is to have microcontroller "turning on" LED diode with logical one (i.e. +5 volts) and the other way is with logical zero (i.e. ground potential). The first way is not used very frequently because it requires the microcontroller to be diode current source. The second way works with higher current LED diodes as shown bellow in Figures 4.2, and 4.3.

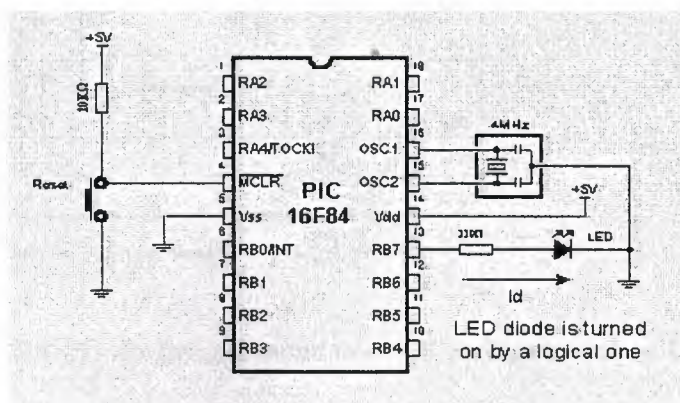


Figure 4.2: LED diode is turned on by a logical one.

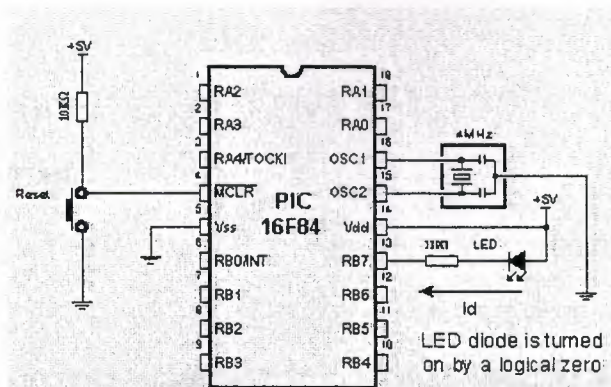


Figure 4.3: LED diode is turned on by a logical zero.

The following application uses instructions High, Low and Pause to turn on and off an LED diode every half a second. In this example 8 LEDs are connected to Port B of the microcontroller but only the LED connected to bit 7 of Port B is used (see Figure 4.4) in the program:

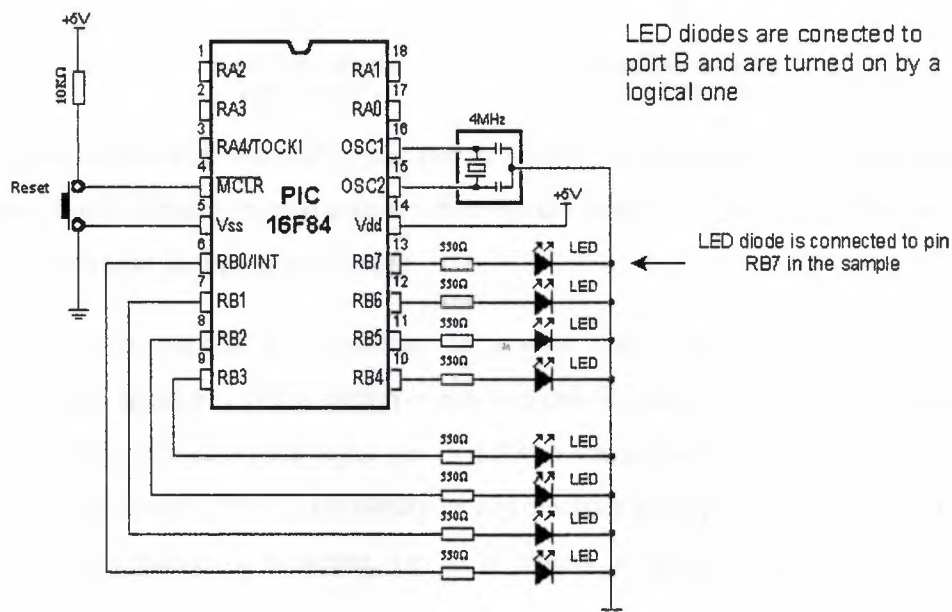


Figure 4.4: LED diodes are connected to portB and are turned on by a logical one.

```

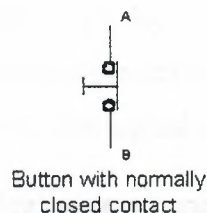
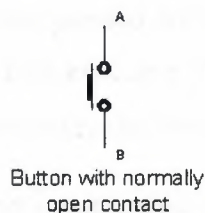
Loop:
  High PORTB.7
  Pause 500
  Low PORTB.7
  Pause 500
  Goto loop
End

```

The above example is a typical single tasking application where there is only one task and this task controls the flashing of the LED in an endless loop.

4.3.2 BUTTON EXAMPLE

Button is a mechanical component which connects or disconnects two points A and B over its contacts. By function, button contacts can be normally open or normally closed.



Pressing the button with normally open contact connects the points A and B, while pressing the button with normally closed contact disconnects A and B. Buttons can be connected to the microcontroller in one of two ways:

In the first case, button is connected in a way that logical one (+5V) remains on microcontroller input pin while button is not pressed. Resistor between a button and power voltage has role of holding the input pin in defined state when the button is not pressed (in this case a logical one). This is necessary as a protection from glitch on input pin that might cause misinterpretation of program, i.e. as if button is pressed when it is not as shown bellow in Figure 4.5.

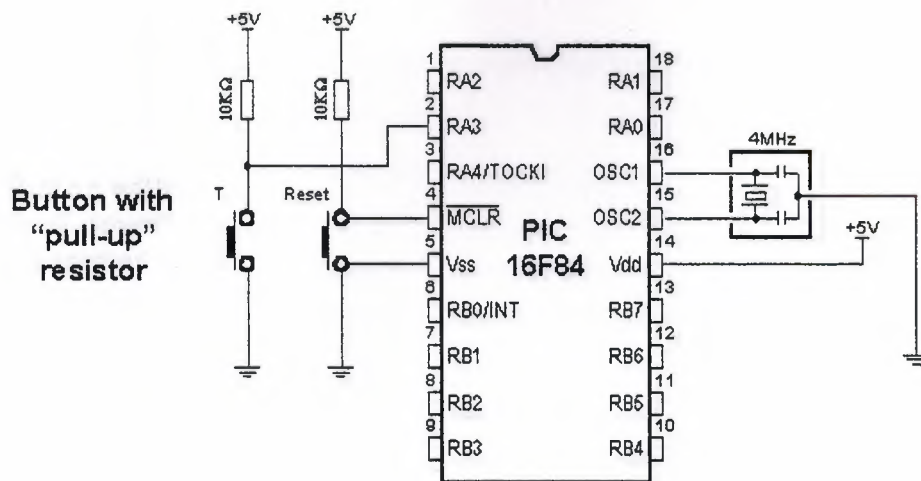


Figure 4.5: Button with "PULL-UP" resistor.

When the button is pressed, input pin is short circuited to the ground (0V) which indicates change on input pin. Voltage has dropped from 5V to 0V. This change is interpreted by program as if button was pressed and part of program code tied to a button (for example turn on LED diode) is then executed. This way of defining pin states is called defining with "pull-up" resistors, associating that the line is held up on the logical one level.

In the other case, button is connected in a way that logical zero remains on input pin. Now, resistor is between input pin and a logical zero, meaning that pressing the button brings logical one to input pin. Voltage goes up from 0V to +5V. Microcontroller program should recognize change on input pin and execute the specific part of program code. This way of defining pin states is called defining with "pull-down" resistors, associating that the line is held down on the logical zero level as shown bellow in Figure 4.6.

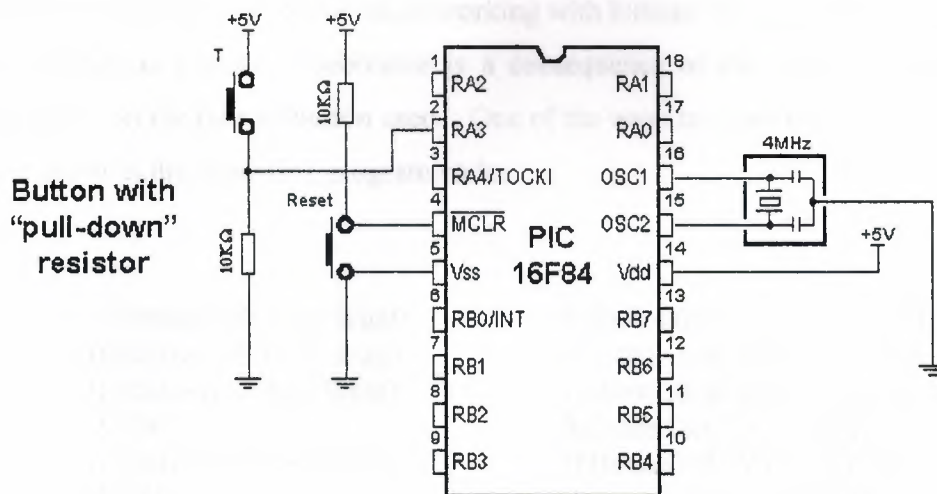


Figure 4.6: Button with “PULL-DOWN” resistor.

Common way to connect a button is with pull-up resistors, meaning that pressing the button changes pin state from logical one to logical zero. The Figure 4.7 displays four buttons connected to the microcontroller using the pull-up resistors.

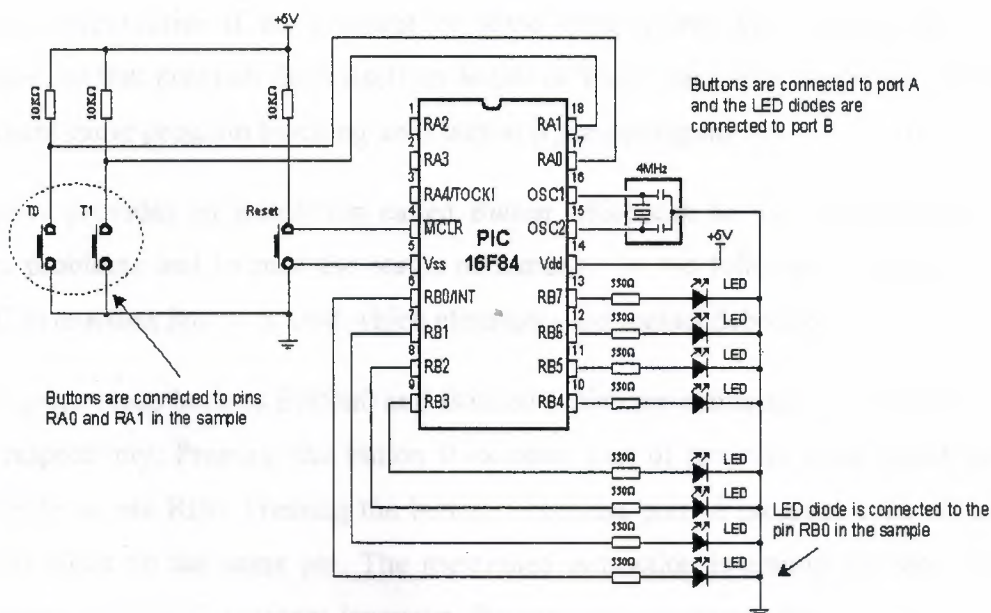


Figure 4.7: Four buttons connected to the microcontroller using the pull-up resistors.

One of the problems that may occur when working with buttons is the contact debounce the moment a button is pressed. Debounce is a consequence of the contact vibration and heavily depends on the type of button used. One of the ways to solve the contact debounce problem is given in the following program code:

	If Button0=0 Then Wait0	'If Button0=0, Jump to Wait0
	If Button1=0 Then Wait1	'If Button1=0, Jump to Wait1
Wait0:	If Button0=0 then Wait0	'If Button0=0' Wait until it is
	W=W+1	'Released and increase W
Wait1:	If Button0=0 then Wait0	'If Button1=0' Wait until it is
	W=W-1	'Released and decrease W

Pressing the Button0 causes the program to jump to address Wait0 where it remains in the loop until the button is released (this achieves that single button push is just once handled in program). When Button0 is released program continues executing instructions (in this case variable W is increased by one). Pressing Button1 causes the same effect, except that variable W is decreased by one.

Problems might arise if an interrupt or some other source slows down the program execution, so that program finds itself on Wait0 or Wait1 lines after the button is released. This might cause program blocking until button is pressed again.

PIC Basic provides an instruction called *Button* which can be used to debounce button contact problems and to read the status of buttons. In the following program code the BASIC instruction *Button* is used which eliminates the contact debounce.

The program reads buttons Button0 and Button1 which are connected to the pins RA0 and RA1, respectively. Pressing the button 0 executes part of program code which turns on LED diode on pin RB0. Pressing the button 1 executes part of program code which turns off LED diode on the same pin. The mentioned instruction is among the most complex instructions of BASIC program language. Besides few arguments that should be defined, instruction has an argument for setting the delay time between recognition of two different button pressures (the third argument). Its setting depends on the purpose of the button as well as mechanical properties of the button. Still, it came clear over time that maximal

value of last argument represents the best solution for most applications, because of great disproportion in human reaction and microcontroller speed.

B0 var byte	'Variable used by instruction BUTTON
Symbol Button0 = PORTA.0	'Button 0 is connected to pin RA0
Symbol Button1 = PORTA.1	'Button 1 is connected to pin RA1
Symbol LED = PORTB.0	'LED diode is connected to RB0
TRISA = \$FF	'ALL pins of port A are input
TRISB = \$00	'ALL pins of port B are Output
PORTB = \$00	'Turn off all LED diodes at start
Main:	
B0 = 0	'Initialize the variable B0
Button button0,0,255,0,B0,1,LedOn	'If Button 0 is pressed jump onto LedOn
B0 = 0	
Button button0,0,255,0,B0,1,LedOff	'If Button 1 is pressed jump onto LedOff
GOTO Main	'Jump back to the beginning of the program
LedOn:	
LED = 1	'Turn on LED diode
Goto Main	
LedOff:	
LED = 0	'Turn off LED diode
Goto Main	
End	'End of program

The above code is another example of single tasking on a microcontroller.

4.3.3 SEVEN-SEGMENT DISPLAY EXAMPLE

Seven-segment displays are frequently used in electronic circuits as indicators. In this section the controlling of a seven-segment display with a microcontroller is describes as another example of more complex single tasking.

Most common form of communication between the microcontroller system and a human being is, of course, the visual communication. The simplest form is the LED diode, while seven-segment digits represent more advanced form of visual communication. The name comes from the seven diodes (there is an eighth diode for a dot) arranged to form decimal

digits from 0 to 9. Appearance of a seven-segment digit is given on a Figure 4.8 as shown below.

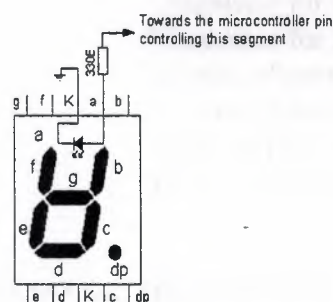


Figure 4.8: Seven-Segment digits.

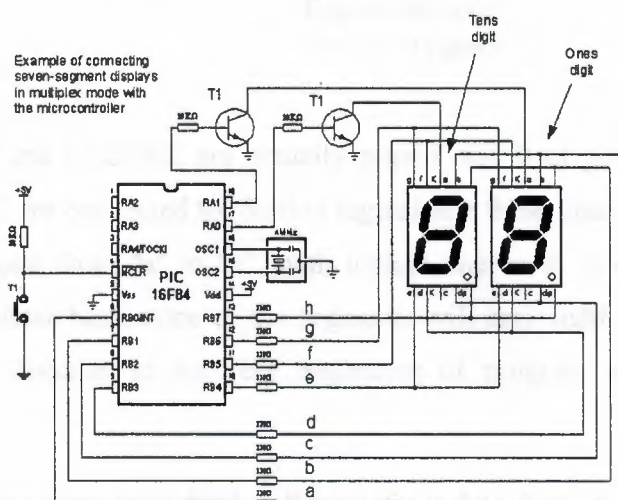


Figure 4.9: Connecting seven segment displays in multiplex mode with the microcontroller

One of the ways to connect a seven-segment display to the microcontroller is given in Figure 4.9 above. In this example, the system is connected to use seven-segment digits with common cathode. This means that segments emit light when logical one is brought to them, and that output of all segments must be a transistor connected to common cathode, as shown on the figure. If transistor is in conducting mode any segment with logical one will emit light, and if not no segment will emit light, regardless of its pin state.

If we use the above circuit diagram, one of the ways to realize the display in BASIC could be the following program code:

Digit var byte	'Value of number to be displayed
Maska var byte	'Mask of number to be displayed
i var byte	'temporary variable
LEDDis1 var PORTA.1	'Transistor for ones digit
LEDDis2 var PORTA.0	'Transistor for tens digit
TRISA=%00000000	'all pins of port A are output
TRISB=%00000000	'all pins of port B are output
LEDDis2=0	'Digit on PA1 (ones) is off
LEDDis1=1	'Digit on PA0 (ones) is on
Main:	
For i=0 to 9	
lookup Digit,[\$3F,\$06,\$5B,\$4F,\$66,\$6D,\$7D,\$07,\$7F,\$6f],Maska	
PORTB=Maska	'Send mask of a number to port B
Pause 500	'pause allowing to see the change
Next i	'Increase i by one
Goto Main	'Repeat the loop
End	'End of program

Variables LEDDis1 and LEDDis2 are actually pins 1 and 0 of port A, where bases of transistors T1 and T2 are connected to. Setting logical one these pins turn on the transistor, allowing every segment from "a" to "h", with logical one on it, to emit light. If there is logical zero on transistor base, none of the segments will emit light, regardless of the pin state. Tens digit is disabled at the very beginning of program, ahead of label Main (LEDDis2=0).

The purpose of the program is to display figures from 0 to 9 on the ones digit, with 0.5 seconds pause in between. In order to display any number, its mask must be sent to port B. For example, if we need to display "1", segments "b" and "c" must be set to 1 and the rest must be zero. If (according to the scheme above) segments b and c are connected to the first and the second pin of port B, values 0000 and 0110 should be set to port B. These values which are set to port are commonly called "masks". For example, Mask for number "1" is value 0000 0110 or \$06 (hexadecimal) as shown in Table 4.1. i.e. sending the hexadecimal number \$06 to Port B will show number "1" on the display.

Table 4.1: Contains corresponding mask values for numbers 0-9.

Digit	Seg. h	Seg. g	Seg. f	Seg. e	Seg. d	Seg. c	Seg. b	Seg. a	HEX
0	0	0	1	1	1	1	1	1	\$3F
1	0	0	0	0	0	1	1	0	\$06
2	0	1	0	1	1	0	1	1	\$5B
3	0	1	0	0	1	1	1	1	\$4F
4	0	1	1	0	0	1	1	0	\$66
5	0	1	1	0	1	1	0	1	\$6D
6	0	1	1	1	1	1	0	1	\$7D
7	0	0	0	0	0	1	1	1	\$07
8	0	1	1	1	1	1	1	1	\$7F
9	0	1	1	0	1	1	1	1	\$6F

The above program uses the instruction `Lookup` to apply an appropriate mask to numerical value. Instruction `Lookup` works very simply - it puts a character from a sequence, its position defined by numerical value `Digit`, to variable `Mask`. For example, `Mask` will take value `$5B` if `Digit` has value 2. In that manner, we can easily get mask for any decimal digit.

Continual display of `Mask` (`PORTB=Mask`) for appropriate value of variable `Digit`, with 0.5sec pause, will produce an effect of digits rotating from 0 to 9.

Problem with multiplexing occurs when displaying more than one digit is needed on two or more displays. It is necessary to put one mask on one digit quickly enough and activate it's transistor, then put the second mask and activate the second transistor (of course, if one of the transistors is in conducting mode, the other should not work because both digits will display the same value).

New program differs from the one above in converting 2-digits value to 2 masks, which are displayed in a way that human eye gets impression of simultaneous existence of both

figures (this is the reason for calling it "multiplexing" - only one display actually emits in any given moment).

Let's say we need to display number 35. First, the number should be separated into tens and ones (in this case, digits 3 and 5) and their masks sent to port B. This separation can be done with instruction Dig. For example, Digit1= W dig 0 will extract ones digit from variable W and store it into variable Digit1. If 0 is substituted with 1, tens digit will be extracted. Following the same logic, 2 extracts number of hundreds, 3 numbers of thousands, etc.

Digit var byte	'Value of number to be displayed
Mask Var byte	'Mask of number to be displayed
W var byte	'temporary variable
LIDDis1 var PORTA.1	'Transistor for ones digit
LIDDis2 var PORTA.0	'Transistor for tens digit
TRISA=%00000000	'all pins of port A are output
TRISB=%00000000	'all pins of port B are output
LIDDis1=0	'ones digit is off in the start
LIDDis2=0	'tens digit is off in the start
Main:	
W=35	
Digit=W dig 1	'Put tens to variable digit
Gosub bin2seg	'Call the conversion of binary number
	'To a code of appropriate 7seg digit
PORTB=Mask	'Set the mask of a digit to port B
LIDDis2=1	'Print the tens digit
Pause 1	'Hold it printed for 1 ms
LIDDis2=0	'Turn off the tens digit
Digit=W dig 0	'Put ones to variable digit
Gosub bin2seg	'Call the conversion of binary number
	'To a code of appropriate 7seg digit
PORTB=Digit	
LIDDis1=1	'Print the ones digit
Pause 1	'Hold it printed for 1 ms
LIDDis1=0	'Turn off the ones digit
Goto Main	'Again, for achieving the effect that
	'Both digits are on simultaneously
bin2seg:	
lookup Digit,[\$3F,\$06,\$5B,\$4F,\$66,\$6D,\$7D,\$07,\$7F,\$6f],Mask	
Return	
End	

This part of program code prints value 35 on two seven-segment displays. The rest of the program is very similar to the last example, except for having one transition caused by displaying one digit after another. This transition can be spotted when LEDDisp1 is being turned off and LEDDisp2 turned on with a new mask. Lookup table is still the same and may be called as a subroutine when needed.

The multiplexing problem is solved for now, but the program doesn't have a sole purpose to show values on displays. It is commonly just a subroutine for displaying certain information. However, this kind of solution for showing data on display will make essence of the program much more complicated. This newly encountered problem may be solved by moving part of the program for refreshing the digits (part of the program code for handling the masks and controlling the transistors) to interrupt routine. The following program shows how to use interrupts for refreshing the display. Main program increases the value of variable W from 0 to 99 and that value is printed on displays. After reaching the value of 99, counter begins again.

Digit var byte	'Value of number to be displayed
Mask Var byte	'Mask of number to be displayed
W var byte	'temporary variable
i var byte	'temporary variable
LIDDis1 var PORTA.1	'Transistor for ones digit
LIDDis2 var PORTA.0	'Transistor for tens digit
TRISA=%00000000	'all pins of port A are output
TRISB=%00000000	'all pins of port B are output
LIDDis1=0	'ones digit is off in the start
LIDDis2=0	'tens digit is off in the start
INTCON = %00100000	'Enable interrupt TMR0
OPTION_REG = %10000000	'Initialization of presale
ON Interrupt goto ISR	'Interrupt vector
INTCON = %10100000	'Enable interrupts
W=0	'Initialization of variable W
Main:	'Beginning of the program
For i=1 to 99	'Print values from 0 to 99
W=W+1	'Increase variable W
Gosub prepare	'Prepare value from W to be displayed
Pause 500	'Pause to see the digits
Next i	
Goto Main	'Print values from 0 to 99 again

Prepare:

Digit=W dig 1	'Value of ones is put to var. Digit
Gosub bin2seg	'Conversion digit to mask
Mask=Digit	'Mask 1 contains the mask of ones
Digit=W dig 0	
Gosub bin2seg	'Conversion digit to mask
Mask=Digit	'Mask 1 contains the mask of ones
Return	'return from subroutine

bin2seg:

lookup Digit,[\$3F,\$06,\$5B,\$4F,\$66,\$6D,\$7D,\$07,\$7F,\$6f],Mask	
Return	
Disable	'Disable interrupts while ISR is 'Executing

ISR:

PORTB=Mask	'Put a mask of tens digit to port B
LIDDis2=1	'Print the ones digit
Pause 1	'Hold it printed for 1 ms
LIDDis2=0	'Turn off the tens digit
PORTB=Mask	'Put a mask of ones digit to port B
LIDDis1=1	'Print the ones digit
Pause 1	'Hold it printed for 1 ms
LIDDis1=0	'Turn off the ones digit
INTCON.2 = 0	'Clear T0IF flag
Resume	'Return to program
Enable	'Interrupts are enabled again
End	'End of program

Interrupt initialized in this way will generate interrupt every time TMR0 timer changes state from 255 to 0. Every time interrupt takes place, interrupt routine will be executed so that human eye gets impression that both displays print values simultaneously. As can be seen from the program code, everything tied to displaying digits is moved to interrupt routine. However, part of the code for forming the masks to be displayed is in the special subroutine (Gosub Prepare) in order to make interrupt routine code as short as possible. Another reason for this kind of organization is also the need to create masks only when variable W is changed and not every time interrupt takes place.

In the course of main program, programmer does not have to take care of refreshing the display nor anything about displays whatsoever. It is only necessary to call subroutine "Preparation" every time value that will be displayed changes.

The above program code demonstrates another single tasking application on a microcontroller.

This is a more complex single tasking application showing how a light can be controlled. Building light control is a very simple device that is realized using the microcontroller technology. The principle is simple - pressing the button turns on the light in the building for a time period T . Upon that time, all lights turn off. Variable T is defined with potentiometer. It is possible to determine for how long the light will be on by reading the potentiometer as shown in Figure 4.10 below.



Symbol Button0 = PORTA.0	'Button 0 is connected to pin RA0
Symbol Time_in = PORTA.1	'Potentiometer for setting the
	'timer
Symbol Light = PORTA.3	'Output for light
Symbol LED = PORTB.7	'Output for control LED diode
B0 var byte	'Temporary variable used by instr. POT
B1 var byte	'Temporary variable used by instr. BUTTON
i var byte	'Variable in FOR....NEXT instruction
TRISA = %00000111	'Pins RA0, 1, 2 input
TRISB = %01111111	'Pin RB7 is output
Low Light	'Turn off the light
Low LED	'Turn off the control LED diode
Main:	'Beginning of the program
B0 = 0	
Pot Time_in,255,B0	'Time period for which the light is
	'on
B1 = 0	
Button button0,0,255,0,B1,1,Lite	'If Button=0
	'turn on the light
Pause 50	'50 ms pause
Goto Main	'Jump to beginning
Lite:	
High light	'Turn on the light
High LED	'Turn on the control LED
For i=0 to 60 + B0	'If B0 = 0 light is on for 1 min
Pause 1000	'If B0 = 255 light is on for 5 min
Next i	
Low light	'Turn off the light
Low LED	'Turn off the control LED diode
Goto Main	'Jump to beginning
End	'End of program

In the above circuit buttons are used to simulate the light switches at each floor of a building. These buttons are connected to the supply voltage using pull-up resistors. Normally pin RA0 of the microcontroller is at logic 1. Pressing any button lowers the RA0 voltage to logic 0 which is then sensed by the program. Port pin RA3 drives an opto-coupler device. Normally the output of the opto-coupler device is at logic 0 and the light is OFF. When pin RA3 is raised to logic 1, the opto-coupler output changes to logic 1 and the relay at the output is activated, turning the light ON. A small LED is connected to port pin RB1 to indicate when the relay is activated. Port pin RA1 is connected to a resistor-capacitor circuit. The resistor is in the form of a potentiometer. Normally the capacitor is

fully charged and the PIC Basic POT command is used to detect the capacitor discharge rate and hence the potentiometer setting, without the need of using an analog-to-digital converter.

4.4 MULTITASKING APPLICATION ON PIC MICROCONTROLLERS

In this section a number of algorithms have been developed for multitasking on a PIC microcontroller. In the first algorithm, the PIC Basic language is used and a *state machine* type multitasking algorithm is implemented on a microcontroller. A PIC microcontroller system hardware has been developed in order to test the operation of the algorithm. This hardware consists of the following:

- PIC16F84 microcontroller
- Analog-to-digital converter
- 7-segment decoder
- 2-digit 7-segment LED display
- Analog temperature sensor
- Relay switches to operate a fan and a heater
- Push-button switches

The circuit basically measures the ambient temperature using an analog temperature sensor integrated circuit. The temperature is then fed to a PIC16F84 type microcontroller which shows the temperature on the LED displays. If the temperature is above the required setting, then the fan is switched on. If on the other hand the temperature is lower than the requires setting then the heater is switched on.

4.4.1 STATE MACHINE MULTITASKING CIRCUIT BLOCK DIAGRAM

The block diagram of this multitasking circuit is shown in Figure 4.11. Firstly the ADC0804 converter receives the analogue signal from temperature sensor LM35. After

converting the signal to digital, it is sent to the PIC16F84A microcontroller. The microcontroller drives the 7segment displays and the relays.

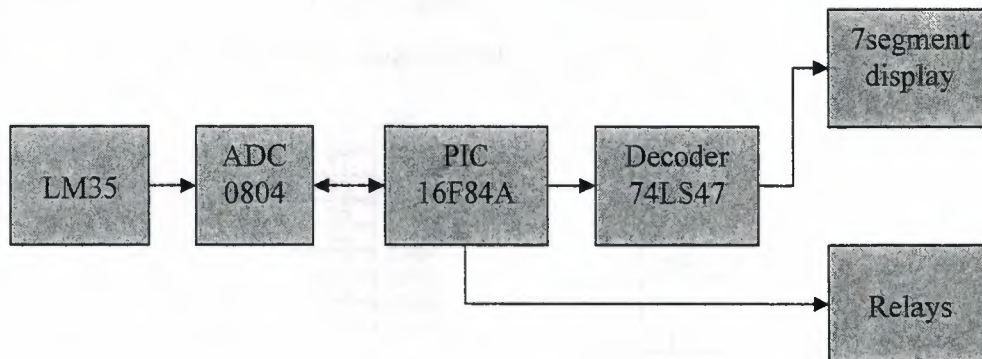


Figure 4.11: Block diagram of multitasking circuit.

4.4.2 ANALOGUE TEMPERATURE SENSOR (LM35)

The LM35 [22] is an analogue temperature sensor which converts the temperature to an analogue electrical signal. The LM35 is precision semiconductor temperature sensor gives an output of 10mV per degree Centigrade. The LM35 sensor pin configuration is shown in Figure 4.12.

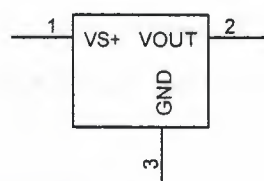


Figure 4.12: LM35 Pin configuration.

4.4.3 ANALOGUE TO DIGITAL CONVERTER (ADC0804)

ADC0804 [23] is an 8-bit parallel A/D converter. This is 20 pin devices which a conversion time of 100 microseconds. As shown in Figure 4.13, the pin description is listed bellow:

DB0-DB7	8 data output pins
RD	Read input

WR	Write input
INTR	Interrupt output
CLKR/CLKIN	Clock control inputs
Vin+	Positive analogue input

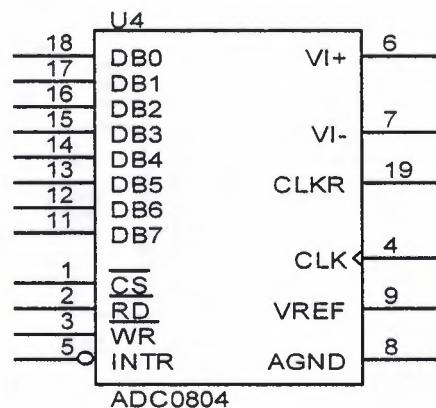


Figure 4.13: ADC0804 Pin configuration.

4.4.4 BCD TO 7SEGMENT DECODER (74LS47)

The circuits [24] accept 4-bit binary-coded-decimal (BCD) and, depending on the state of the auxiliary inputs, decodes this data to drive a 7-segment display indicator. The chip offers active LOW, high sink current outputs for driving indicators directly. Seven NAND gates and one driver are connected in pairs to make BCD data and its complement available to the seven decoding AND-OR-INVERT gates. In figure 4.14 the 74LS47 BCD to 7segment decoder is shown.

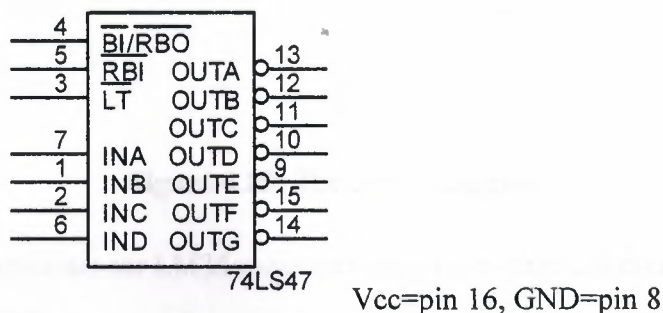


Figure 4.14: 74LS47 pin configuration.

4.4.5 THE CIRCUIT DIAGRAM

The complete circuit diagram is shown in Figure 4.15. The circuit consists of a number of integrated circuits, passive components, transistors and push-button switches.

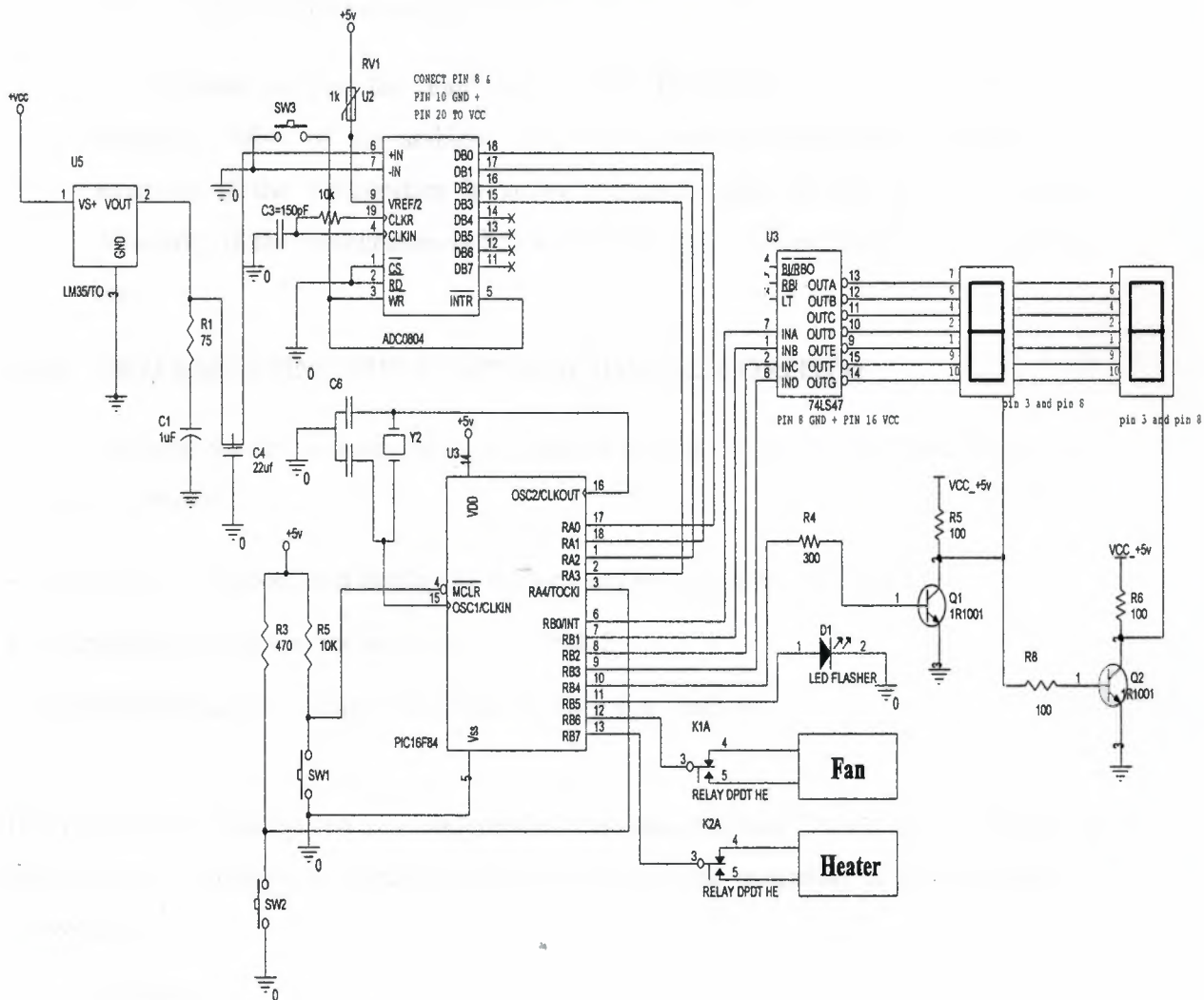


Figure 4.15: The circuit diagram

- The temperature sensor LM35 measures the temperature and converts it to analogue electrical signal.
- The converted signal is sent to A/D converter to convert the analogue signal to digital.

- The converted digital signal is sent to the PIC microcontroller as digital input.
- The extracted bits are sent to a 7-segment decoder to convert them so that they can be displayed in 7-segment displays. Port pin RB4 of the microcontroller determines which digit to turn on at any given time.
- The program controls the "Fan" relay or the "Electrical heater" relay based on the measured value of the ambient temperature and the temperature setting. For example, if the temperature is above required value the fan will be turned on. Similarly, if the temperature is below required value the electrical heater will turn on.

4.4.6 IMPLEMENTING THE STATE MACHINE ALGORITHM

In this example we try to keep the temperature between 15°C and 25°C and there are 3 activities, namely:

- Reading, converting, and displaying the ambient temperature (i.e. Task 1)
- Implementing the control algorithm (i.e. Task 2)
- Flashing the LED 10 times if the heater is OFF (i.e. Task 3)

If each activity is configured as an independent task then we have 3 tasks and the following state-machine multitasking algorithm can be derived (the operation of the algorithm is shown as a PDL):

BEGIN

Current-State = 1

DO FOREVER

IF Current-State = 1

Read temperature from analog sensor

Convert temperature to digital form

Display temperature on 7-segment display

Current-State = 2

ELSE IF Current-State = 2

IF temperature < 15°C

Turn ON heater

```

        Turn OFF fan
    ELSE IF temperature > 25°C
        Turn OFF heater
        Turn ON fan
    END IF
    Current-State = 3
    ELSE IF Current-State = 3
        DO 10 times
            Flash the LED
        ENDDO
    END IF
    Current-State = 1
ENDDO
END

```

Figure 4.16 shows the actual PIC Basic code developed by the author and used to test the above algorithm.

```

*****
'* Name : STATE-MACHINE.BAS *
'* Author : *
'* Date : 2/14/2005 *
'* Version : 1.0 *
'* Notes : This is an example State-Machine based multitasking code. *
'* The code consists of 3 tasks called Task1, Task2 and Task3. *
'* *
*****

TRISA = %00011111 ' All pins of port A are input
TRISB = $00 ' All Pins of Port B are output

Left Var byte ' Data for the left display
Right var Byte ' Data for the right display
i var byte
ADC_In var byte ' Store the ADC data
Digit Var byte ' Scale the temperature data
Current_State Var byte ' State variable

Current_State = 1 ' Set initial state to 1

```


Main:

```
SELECT CASE Current_State
CASE 1
    Gosub Task1
    Current_State = 2
CASE 2
    Gosub Task2
    Current_State = 3
CASE 3
    Gosub Task3
    Current_State = 1
END SELECT
Goto Main
```

Task1:

```
ADC_In = PORTA          ' read the temp data
Digit = ADC_In & %00001111 ' store and mask The Data From The ADC
Digit = Digit * 2        ' scale the data
Right = Digit dig 0      ' Extract the ones Digit
PORTB = Right & %00001111 ' Send the nibble to 74ls47
high PORTB.4             ' Activate The Right-Hand 7-segment Display
pause 1
Left = Digit dig 1       ' extract the tens digit
PORTB = Left & %00001111 ' send the nibble to 74ls47
LOW PORTB.4              ' Activate the Left-Hand 7-segment display
pause 1
return
```

Task2:

```
if Digit < 15 then      ' if the temp is less than 15
    low PORTB.7         ' switch off Fan
    high PORTB.6        ' switch on Heater
endif

if Digit > 25 then      ' if the temp. is greater than 25
    low PORTB.6         ' switch off Heater
    HIGH PORTB.7        ' Switch on Fan
endif
return
```

Task3:

```
if PORTA.4 = 0 then
    for i = 1 to 10
        toggle PORTB.5
        pause 10
    next i
```

```
endif  
return
```

```
End
```

Figure 4.16: The Basic program

The state-machine approach is simple but it has the main disadvantage that each task is allowed to execute in its entirety which may take too long. This can in general be prevented by splitting each function into a number of smaller states and executing only one state each time round the infinite loop.

4.4.7 IMPLEMENTING THE TIME SLICED MULTITASKING ALGORITHM

In this section an algorithm is developed to implement the time sliced multitasking on a PIC microcontroller. The timer (TMR0) of the microcontroller is set-up to generate interrupts at regular time intervals and the tasks are scheduled whenever an interrupt occurs. The algorithm is described below as a PDL for a 3 task system:

BEGIN

```
Set-up timer interrupts  
Counter = 1
```

DO FOREVER

```
IF Timer-Expired = True  
    IF Counter = 1  
        DO Task 1  
    ELSE IF Counter = 2  
        DO Task 2  
    ELSE IF Counter = 3  
        DO Task 3  
    Counter = 1
```

ENDIF

ENDIF

```
Timer-Expired = False
```

ENDDO

INTERRUPT-SERVICE-ROUTINE:

```
Increment Counter  
Re-initialise timer interrupts  
RETURN from interrupt
```

END

The actual code is shown in Figure 4.17. In this code the timer TMR0 is set to overflow every time it reaches a count of 255 and a pre-scaler value of 256 is used by setting PSA0, PSA1, and PSA2 to logic 1. With a 4MHz clock rate, the internal clock rate is $4\text{MHz} / 4 = 1\text{MHz}$ and thus the timer clock period is 1 microsecond. In this example, timer interrupts will be generated when the timer overflows, which is at time intervals of $256 \times 256 = 65536$ microseconds, or approximately at every 65ms. Variable *Counter* is incremented by 1 inside the interrupt service routine and thus, a new task will be scheduled at every 65ms.

The Tasks are same as before and are not shown in the following code.

```
*****
'* Name : TIME-SLICING.BAS *
'* Author : *
'* Date : 3/10/2005 *
'* Version : 1.0 *
'* Notes : This is an example Time Sliced multitasking code. *
'* The code consists of 3 tasks called Task1, Task2 and Task3. *
'* and each task receives 65ms of processor time. *
'* *
*****

Counter          Var Byte
Timer_Expired     Var Byte

' Set timer TMR0 to interrupt at every 65ms
'

OPTION_REG = $17          ' Set PSA0=PSA1=PSA2=1 (prescaler 256)
INTCON = $A0              ' Enable TMR0 interrupts
ON INTERRUPT GOTO TIMER_INT

    Timer_Expired = 0

MAIN:
    IF Timer_Expired = 1
        SELECT CASE Counter
            CASE 1
                Gosub Task1
            CASE 2
                Gosub Task2
            CASE 3
```



```

        Gosub Task3
        Counter = 1
    END SELECT
ENDIF
Timer_Expired = 0
Goto Main

```

' Timer interrupt Service Routine

```

Disable
TIMER_INT:
    Counter = Counter + 1
    Timer_Expired = 1
    INTCON.2 = 0
    Resume
    Enable
END

```

'Disable interrupts
 'Point to next task
 'Set Timer_Expired flag
 'Re-enable Timer interrupt
 'Resume normal program
 'Re-enable interrupts in general

Figure 4.17: Time sliced multitasking code.

4.4.8 IMPLEMENTING THE COOPERATIVE MULTITASKING ALGORITHM

In Cooperative multitasking each task releases the CPU whenever it thinks it is the right time to do so. For example, if an I/O condition is not satisfied the task may decide to release the CPU for the other tasks in the system. Also, a task does not necessarily run to its full length before it releases the CPU. The important point in Cooperative multitasking is that when a task gets back the CPU time, it continues from the point it left just before it released the CPU. As a result of this, all tasks seem to run as if they are executed continuously by different processors.

Because a task has to continue from the point it left before it released the CPU, it is necessary to save the task return addresses in memory so that the scheduling algorithm knows how to resume the tasks from the correct addresses.

The Cooperative multitasking algorithm for a 2 task PIC application is given below as a PDL:

BEGIN

Get Task 1 starting address
Save Task 1 starting address in Context memory
Load Task 2 starting address into W register
Call Exchange
Reserved location for subroutine Return point

Exchange:

Swap W register and Context memory
Load W into Program Counter
Jump to Program Counter address

Task 1:

.....
Return with address of next instruction in W
.....
.....
Goto Task 1

Task 2:

.....
Return with address of next instruction in W
.....
.....
Goto Task 2

END

Task 1 and Task 2 can release the CPU whenever they want by issuing a return instruction with the address of the next instruction (return point within the task) within the task loaded into the W register. The routine starting with the label *Exchange* is a tiny scheduler which swaps the address in the W register with the address in Context memory and then loads W into the program counter and jumps to this address. The effect of this is a simple context switching where the CPU is shared between the two tasks as the tasks release and then grab the CPU.

Figure 4.18 shows the actual PIC assembly language code which implements the above Cooperative multitasking algorithm for 2 tasks.

```

*****
;* Name   : COOPERATIVE.ASM                                     *
;* Version : 1.0                                                *
;* Notes  : This is an example Cooperative multitasking code.   *
;*                                                *
*****

context      equ    0x0C                                     ; Address of Context memory

switch_task  MACRO                                           ; Return with address in W
    retlw $+1
ENDM

SWAP         MACRO reg                                       ; Swap W register and reg
    xorwf reg,f
    xorwf reg,w
    xorwf reg,f
ENDM

    movlw Task1                                           ;Get Task 1 address
    movwf context                                       ;Save in Context memory
    movlw Task2                                           ;Get Task 2 address
    ;
    ; Scheduling. Swap the contents of W register and Context memory
    ;
    call $+2
    goto $-1
    swap                                           ;Swap W register and Context memory
    movwf pcl                                           ;Jump to program counter address

Task1        .....
            .....
            switch_task                                   ;Release CPU
            .....
            goto Task1

Task2        .....
            .....
            switch_task                                   ;Release CPU
            .....
            goto Task2

END

```

Figure 4.18 Cooperative multitasking with 2 tasks.

At the beginning of the program the Context memory is assigned address 0x0C of the general purpose memory, which is the first available memory location on PIC16F84 microcontrollers. The *switch_task* macro is then defined which returns from a subroutine with the address in the W register. The SWAP macro swaps registers W and the general purpose register specified by its parameter *reg*. The two tasks are labelled Task1 and Task2 respectively and as shown in the code the tasks release the CPU by calling the macro *switch_task*.

An example use of the above code is given below where 2 tasks are used and Task 1 increments a variable called *counter1* starting from 1 and displays the result on Port B. Similarly, Task 2 increments another variable called *counter2* starting from 100 and again displays the result on Port B. As a result, the following data should be displayed on Port B: 0, 100, 1, 101, 2, 102,.....

```

*****
;* Name   : COOPERATIVE_TEST.ASM                      *
;* Version : 1.0                                         *
;* Notes  : This is an example Cooperative multitasking code. *
;*          Two tasks are used. Task 1 increments a variable starting *
;*          1 and outputs to Port B. Task 2 increments another variable *
;*          starting from 100 and again outputs to Port B. *
;*          *                                           *
*****

LIST P=16F84
#include "p16f84.inc"

context      equ    0x0C                                ; Address of Context memory
counter1     equ    0x0D                                ; Counter1 address
counter2     equ    0x0E                                ; Counter2 address

switch_task  MACRO                                         ; Return with address in W
    retlw $+1
END

SWAP         MACRO reg                                     ; Swap W register and reg
    xorwf reg,f
    xorwf reg,w
    xorwf reg,f
ENDM

```

```

        ORG 0                                ;Start of Main code

Main    bsf    STATUS,5                      ;Move to Bank 1
        clrf   TRISB                        ;Port B is output port
        bcf    STATUS,5                      ;Move back to Bank 0

        movlw  1                            ;Counter1 = 1
        movwf  counter1                     ;W = 100
        movlw  d'100'                       ;Counter2 = 100
        movwf  counter2

        movlw  Task1                        ;Get Task 1 address
        movwf  context                       ;Save in Context memory
        movlw  Task2                        ;Get Task 2 address
        ;
        ; Scheduling. Swap the contents of W register and Context memory
        ;
        call  $+2
        goto  $-1
        swap context                         ;Swap W register and Context memory
        movwf  PCL                          ;Jump to program counter address
;
;***** START OF TASKS *****
;
Task1    incf   counter1,f                   ;increment Counter1
        movf   counter1,w                   ;W = Counter1
        movwf  PORTB                        ;Send to Port
        switch_task                         ;Release CPU
        goto   Task1

Task2    incf   counter2,f                   ;increment Counter2
        movf   counter2,w                   ;W = Counter2
        movwf  PORTB                        ;Send to Port
        switch_task                         ;Release CPU
        goto   Task2

        END

```

The above code was simulated on a PC using the MPLAB simulation package and the results were correct.

A screen shot of the simulation is shown in Figure 4.19.

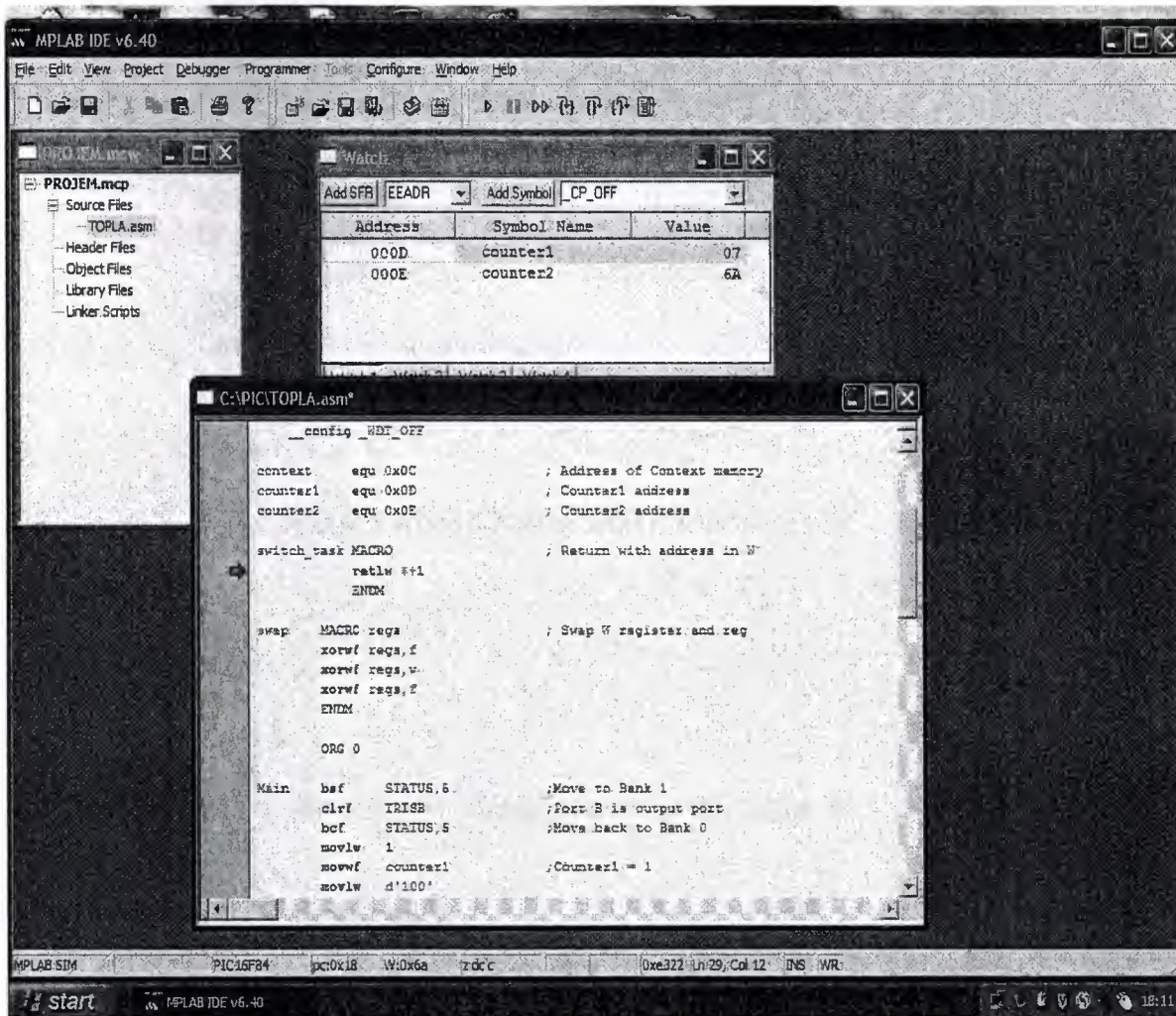


Figure 4.19: Screen shot of the simulation.

4.4.8.1 GENERALISING TO MANY TASKS

The Cooperative multitasking algorithm given above is only for 2 tasks but this algorithm can be generalised to any number of tasks (provided there is enough memory space on the microcontroller) as described by the PDL in Figure 4.20 below:

BEGIN

DO for all tasks

 Get Task *i* starting address

 Save Task *i* starting address in Context memory

ENDDO

Set *pointer* to top of Context memory
 Call Exchange
 Reserved location for subroutine Return point
 Exchange:
 Swap W register and Context memory pointed to by *pointer*
 Increment *pointer*
 IF *pointer* points to last task in Context memory
 Set *pointer* to top of Context memory
 ENDIF
 Load W into Program Counter
 Jump to Program Counter address

Task 1:

.....
 Return with address of next instruction in W

 Goto Task 1

Task 2:

.....
 Return with address of next instruction in W

 Goto Task 2

Task 3:

.....
 Return with address of next instruction in W

 Goto Task 3

Task n:

.....
 Return with address of next instruction in W

 Goto Task n

END

Figure 4.20: Generalised Cooperative multitasking algorithm.

The PIC assembly code to implement the algorithm for n Cooperative multitasking tasks is given in Figure 4.21 below. Several MACROs are used to set-up the multitasking environment. *switch_task* macro saves the return address of each task in the W register so that we can return to each task. *swap* macro is used to swap the contents of the W register and any other file register. *context* macro is called at the beginning of the main program and this macro creates the memory locations where the return addresses of the tasks are to be stored. This macro receives the number of tasks and the first available free memory location and the creates statements of the following format (assuming there are 3 tasks with the memory start address 0x20):

```
context1 equ 0x20
context2 equ 0x21
context3 equ 0x22
```

init macro is called at the beginning of the main program and this macro loads the task starting addresses to the *context* memory locations so that the scheduler can start the tasks. For example, if there are 3 tasks, the *init* macro creates the following statement:

```
movlw Task1           ;get address of Task1
movwf context1        ;save in memory address context1
movlw Task2           ;get address of Task2
movwf context2        ;save in memory address context2
movlw Task3           ;get address of Task3
movwf context3        ;save in memory address context3
```

ntasks is the numbers of tasks in the system and this variable must be set-up to the correct value at the beginning of the program.

```

;*****
;* Name   : COOPERATIVE_TEST.ASM
;* Version : 1.0
;* Notes  : This is an example Cooperative multitasking code for n tasks
;*
;*****
LIST P=16F84
#include "p16f84.inc"

task_count    equ 0x0C                                ;task_count
;
; MACROS
;
switch_task    MACRO                                ; Return with address in W
retlw $+1
ENDM

swap           MACRO reg
xorwf reg,f
xorwf reg,w
xorwf reg,f
ENDM

context        MACRO ntasks,k
t = 1
m = k
while t <= ntasks
context#v(t) equ 0x0#v(m)
t = t + 1
m = m + 1
endw
ENDM

init           MACRO ntasks                                ; Swap W register and reg
t = 1
while t <= ntasks
movlw Task#v(t)
movwf context#v(t)
t = t + 1
endw
ENDM

;***** START OF MAIN CODE *****
ORG 0

ntasks equ n                                ;Number of tasks

```



```

Main .....;Start of MAIN code
.....
.....

context ntasks, 0x20;create context addresses
init ntasks;load task addresses
movlw ntasks+1
movwf task_count
;
; task scheduler
;
movlw context1
movwf FSR
movf INDF,w
call $+5
swap INDF
incf FSR,f
cont swap INDF
decf task_count,f
btfsc STATUS,2
goto rst
movwf PCL
rst movlw ntasks+1
movwf task_count
movlw context1
movwf FSR
movf INDF,w
goto cont
;
;***** START OF TASKS *****
;
Task1 .....;Release CPU
switch_task
goto Task1

Task2 .....;Release CPU
switch_task
goto Task2

Task3 .....;Release CPU
switch_task
goto Task3

```

```

Task4      .....
           switch_task          ;Release CPU
           goto Task4

Taskn      .....
           switch_task          ;Release CPU
           goto Taskn

           END

```

Figure 4.21 Multitasking Cooperative n tasks.

An example program with 5 tasks is given in Figure 4.22 below. In this program each task uses an independent counter and the value of the count is sent to Port B for testing the program. The program was run on the MPLAB simulator in single-stepping mode and correct results were obtained.

```

*****
;* Name   : COOPERATIVE_TEST.ASM                      *
;* Version : 1.0                                         *
;* Notes  : This is an example Cooperative multitasking code. *
;*          Five tasks are used with each one having its counter. *
;*          The value of the count is displayed on Port B for testing *
;*          *                                           *
*****
LIST P=16F84
#include "p16f84.inc"

task_count    equ    0x0C
counter1      equ    0x0D
counter2      equ    0x0E
counter3      equ    0x0F
counter4      equ    0x10
counter5      equ    0x11

switch_task    MACRO                                ; Return with address in W
               retlw $+1
               ENDM

swap           MACRO reg
               xorwf reg,f
               xorwf reg,w

```

```

        xorwf reg,f
        ENDM

context  MACRO ntasks,k
        t = 1
        m = k
        while t <= ntasks
        context#v(t) equ 0x0#v(m)
        t = t + 1
        m = m + 1
        endw
        ENDM

init  MACRO ntasks                                ; Swap W register and reg
        t = 1
        while t <= ntasks
        movlw Task#v(t)
        movwf context#v(t)
        t = t + 1
        endw
        ENDM

ORG 0

ntasks equ 5                                ;5 tasks in the system

Main  bsf    STATUS,5                        ;Move to Bank 1
      clrf   TRISB                          ;Port B is output port
      bcf    STATUS,5                        ;Move back to Bank 0
      movlw  1
      movwf  counter1                       ;Counter1 = 1
      movlw  0x0a
      movwf  counter2                       ;Counter2 = 10
      movlw  0x14
      movwf  counter3                       ;Counter3 = 20
      movlw  0x1e
      movwf  counter4                       ;Counter4 = 30
      movlw  0x28
      movwf  counter5                       ;Counter5 = 40

      context ntasks,0x20
      init ntasks
      movlw  ntasks+1
      movwf  task_count
      ;

```



```

; task scheduler
;
movlw context1
movwf FSR
movf INDF,w
call $+5
swap INDF
incf FSR,f
cont swap INDF
decf task_count,f
btfsc STATUS,2
goto rst
movwf PCL
rst movlw ntasks+1
movwf task_count
movlw context1
movwf FSR
movf INDF,w
goto cont
;
;***** START OF TASKS *****
;
Task1    incf  counter1,f           ;increment Counter1
         movf  counter1,w         ;W = Counter1
         movwf PORTB             ;Send to Port
         switch_task             ;Release CPU
         goto Task1

Task2    incf  counter2,f           ;increment Counter2
         Movf  counter2,w         ;W = Counter2
         Movwf PORTB             ;Send to Port
         switch_task             ;Release CPU
         goto Task2

Task3    incf  counter3,f           ;increment Counter1
         movf  counter3,w         ;W = Counter1
         movwf PORTB             ;Send to Port
         switch_task             ;Release CPU
         goto Task3

Task4    incf  counter4,f           ;increment Counter2
         Movf  counter4,w         ;W = Counter2
         Movwf PORTB             ;Send to Port
         switch_task             ;Release CPU

```

```

goto Task4
Task5    incf    counter5,f           ;increment Counter2
        Movf    counter5,w           ;W = Counter2
        Movwf   PORTB               ;Send to Port
        switch_task                 ;Release CPU
        goto Task5
END

```

Figure 4.22 Multitasking Cooperative 5 tasks.

4.5 SUMMARY

This chapter has described the implementation of various multitasking algorithms on the PIC microcontroller. In addition, various single tasking applications have been developed and tested by the author using a PIC microcontroller based hardware, developed by the author. This hardware consisted of a temperature sensor, analog-to-digital converter, 7-segment decoder and display, and relays. The Cooperative multitasking algorithm was tested using the MPLAB package, which is freely available from the Microchip web-site.

CONCLUSION

Traditionally microcontrollers have been used in single tasking mode where a single microcontroller is used for a given application. As the applications became more complex the need for multitasking has become a necessity in microcontroller applications. Multitasking on microcontrollers is becoming an important part of a majority of the electronic devices today. Also, as a result of their compact size, low cost, low power consumption and ease in implementation, multitasking on microcontrollers is currently becoming an important concept.

This thesis has investigated the principles of multitasking and several multitasking algorithms have been developed for low-cost popular microcontrollers. These algorithms have been tested using suitable microcontroller simulation packages, such as the MPLAB from Microchip Inc. It is shown in the thesis that multitasking algorithms can easily be implemented on microcontrollers which have very limited resources, such as limited memory and limited I/O capabilities.

Chapter 1 of the thesis has described the general principles of multitasking. Microcontrollers, and their architecture has been outlined in Chapter 2 in detail. In this Chapter, the popular PIC family of microcontrollers have been used as the example microcontroller. The reason for choosing PIC was because currently this is one of the most popular microcontrollers used in industry, commerce, and by the hobby market. The basic microcontroller development cycle and the microcontroller development tools, including the microcontroller language tools are summarised in Chapter 3. Finally, Chapter 4 describes the development of several microcontroller based multitasking algorithms. These algorithms have been developed and tested by the author using commercially available simulator packages.

Although the multitasking algorithms developed by the author should be sufficient for most industrial and commercial applications, they can be extended to include more complex multitasking algorithms, such as the preemptive multitasking.

REFERENCES

- [1] "Multitasking Introduction",
<http://www.bknd.com/cc5x/multitasking.shtml>
- [2] CS 110 Handout #33 spring 2001 May 23, 2001 I/O Announcements
http://www.stanford.edu/class/cs110/handouts/33_IO.pdf
- [3] "Operating Systems – Multitasking"
http://hjem.get2net.dk/rune_moeller_barnkob/multitasking.html
- [4] John Iovine, "PIC Microcontroller Project Book", 2000
<http://www.amazon.com/exec/obidos/ASIN/0071354794/ref%3Dnosim/robotbooks-20/002-1409841-1684012>
- [5] D. Ibrahim, "*Microcontroller programming in C for the 8051*", Butterworth-Heinemann, 2000, London.
- [6] D. Ibrahim, "*Z80 Programming*", Bilesim Yayincilik, 2003, Ankara, Turkey
- [7] John Crisp, "*Introduction to Microprocessors and Microcontrollers*", Second Edition, Elsevier, 1998-2004, London.
- [8] Nebojsa Matic, "*The PIC Microcontroller*", Third Edition, 2003
<http://www.microelectronika.com>.
- [9] William H. Payne, "Embedded Controller Forth for the 8051 Family", San Diego, CA, Third Edition, 1990.
- [10] William H. Payne, "Embedded Controllers Data book", 1992.
- [11] Hon-Won Huang, "PIC Microcontroller: An introduction to Software & Hardware", Thomson Delman Learning, 2004.
- [12] "Microchip Data Sheets and Application Notes"
<http://www.microchip.com>

- [13] Colin K McCordv," *The PIC16F877 Microcontroller*", 2002.
<http://www.mccord.plus.com/FYP/4.htm>
- [14] Nebojsa Matic," *BASIC for PIC Microcontroller*", 2003-2004
<http://www.microelectronika.com>.
- [15] P. Ssasov, "Microcontroller Technology: the 68HC11",
Prentice-Hall Publishing, 1993.
- [16] S. Mackenzie, "The 8051 Microcontroller",
Prentice-Hall, 1998.
- [17] A. Johnson-Laird, "The Programmers CP/M Handbook",
McGraw-Hill Publishing, 1985.
- [18] S. Heath, "Embedded Systems Design",
Newnes, Butterworth-Heinemann, 1999, Oxford
- [19] "Salvo User Manual"
www.pumpkin.com
- [20] J.R. Tocci, "Microprocessors and Microcomputers",
Prentice-Hall Publishing, 1995.
- [21] D. Ibrahim, "PIC Basic: Programming and Projects",
Newnes, Butterworth-Heinemann, 2003.
- [22] Temperature Sensor "LM35"
<http://www.hardware.dibe.unige.it/DataSheets/LM35.pdf>
- [23] Analogue to Digital converter "ADC0804"
<http://www.iguanalabs.com/adc0804.pdf>
- [24] BCD to 7segment decoder "74LS47"
<http://www.ee.washington.edu/stores/DataSheets/74ls/74ls48.pdf>