Chapter 4

Macros and Subroutines

This chapter highlights sound programming habits particularly those pertaining to modularity and code readability. Modularity is the art of designing independent software modules or subprograms better known as subroutines.

SUBROUTINES

A subroutine is a set of instructions designed to perform a specific and usually repetitive task. The user’s program may execute the subroutine simply by “calling” its name. Upon termination of the subroutine’s chore, the CPU resumes execution of remaining jobs. The main advantages of subroutines in programming languages are:

- Staying away from re-inventing the wheel. In fact many useful and elaborate routines have been written by other programmers and are available to “cut-and-paste” free of charge. This speeds up software development and frees the programmer from the burden of re-designing a piece of code from scratch.
- The utilization of subroutines makes programs more compact since an instruction sequence does not have to be reproduced over and over if one intends to perform the same task repetitively. The logical thing to do is to design a subroutine and invoke it as needed in a program.
- Subroutines make programs more readable for they divide big tasks into smaller ones that the user can easily trace. For instance, someone interested in designing a digital thermostat would have to perform the following steps:
  ◦ Read the temperature from a sensor.
  ◦ Display it on a Liquid Crystal Display (LCD).
  ◦ Compare it against a set point for HVAC control.
The programmer may then design three autonomous subroutines namely: ReadTemp, DispTemp and Control. This way if a fault is detected in one of these software modules, it can be corrected independently of the other modules. Thus programmers can concentrate on debugging one subroutine at a time, reminiscent of how humans normally tackle life difficulties: “I will cross that bridge when I get to it”.

A **macro** is a set of instructions framed between the two assembler directives: `macro` and `endm`. It allows programmers to extend the instruction set to handle more complex operations such as 16-bit manipulations, BCD arithmetic, etc. When the assembler encounters a macro call, it inserts the instructions constituting the macro into the program. This is not the case for subroutines; instructions forming a subroutine have only one occurrence in the program and are executed upon demand.

Macros allow programmers to hide certain intricacies inside software blocks and hence they give programs a compact appearance and improved **readability**. In fact, tracing and debugging a long program is rather overwhelming whereas a concise and structured program attracts its reader and is easier to trace. In upcoming sections, macros and subroutines will be developed with the following goals in mind: instruction set extension, program compactness and improved readability.

### PRE-DECLARED VARIABLES

Using the same variable name in several subprograms has the benefit of reducing memory storage and also improving program readability. In order to eliminate side effects, variables must be properly protected in nested subroutines. The **software stack**, introduced in a later section, allows saving and retrieving these variables. In this textbook, a set of variables starting at address 0x040 (access bank) are pre-declared as shown:

```assembly
cblock 0x040
    RegA, RegB, RegC, RegD
    RegE:2, RegF:2
    RegG:2, RegH:2
    wE:2
endc
```

This **constant block** (cblock) declaration will be part of the file `<MyMacros.asm>`. This file essentially contains all the **macros** to be developed throughout this chapter. You need to include it in each assembly language program of this book via the directive:

```
#include <MyMacros.asm>
```

### BASIC MACROS

MPASM assembler macros have the following generic syntax:
Related Content

First Steps Towards a Wise Development Environment for Behavioral Models
[www.igi-global.com/article/first-steps-towards-a-wise-development-environment-for-behavioral-models/170517?camid=4v1a](www.igi-global.com/article/first-steps-towards-a-wise-development-environment-for-behavioral-models/170517?camid=4v1a)

Innovating Healthcare through Remote Monitoring: Effects and Business Model
[www.igi-global.com/article/innovating-healthcare-through-remote-monitoring/144814?camid=4v1a](www.igi-global.com/article/innovating-healthcare-through-remote-monitoring/144814?camid=4v1a)

Retrofitting Existing Web Applications with Effective Dynamic Protection Against SQL Injection Attacks
[www.igi-global.com/chapter/retrofitting-existing-web-applications-effective/65848?camid=4v1a](www.igi-global.com/chapter/retrofitting-existing-web-applications-effective/65848?camid=4v1a)

Software Security Engineering – Part I: Security Requirements and Risk Analysis
[www.igi-global.com/chapter/software-security-engineering-part/75749?camid=4v1a](www.igi-global.com/chapter/software-security-engineering-part/75749?camid=4v1a)