# CHAPTER 4

## Assembly Language Programming

Introduction

An example writting program

Control directives

- 4.1 define
- 4.2 include
- 4.3 constant
- 4.4 variable
- 4.5 set
- 4.6 equ
- 4.7 org
- 4.8 end

Conditional instructions

- 4.9 if
- 4.10 else
- 4.11 endif

- 4.12 while
- 4.13 endw
- 4.14 ifdef
- 4.15 ifndef

Data directives

- 4.16 cblock
- 4.17 endc
- 4.18 db
- 4.19 de
- 4.20 dt

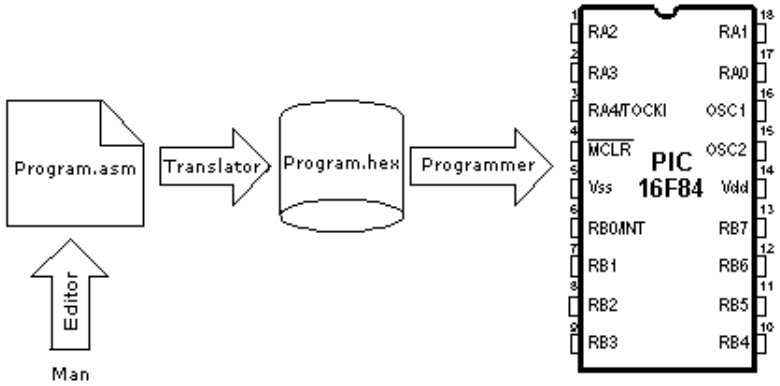
Configurating a directive

- 4.21 _CONFIG
- 4.22 Processor

Assembler arithmetic operators
Files created as a result of program translation
Macros

# Introduction

The ability to communicate is of great importance in any field. However, it is only possible if both communication partners know the same language, i.e follow the same rules during communication. Using these principles as a starting point, we can also define communication that occurs between microcontrollers and man . Language that microcontroller and man use to communicate is called "assembly language". The title itself has no deeper meaning, and is analogue to names of other languages , ex. English or French. More precisely, "assembly language" is just a passing solution. Programs written in assembly language must be translated into a "language of zeros and ones" in order for a microcontroller to understand it. "Assembly language" and "assembler" are two different notions. The first represents a set of rules used in writing a program for a microcontroller, and the other is a program on the personal computer which translates assembly language into a language of zeros and ones. A program that is translated into "zeros" and "ones" is also called "machine language".




**The process of communication between a man and a microcontoller**

Physically, "**Program**" represents a file on the computer disc (or in the memory if it is read in a microcontroller), and is written according to the rules of assembler or some other language for microcontroller programming. Man can understand assembler language as it consists of alphabet signs and words. When writing a program, certain rules must be followed in order to reach a desired effect. A **Translator** interprets each instruction written in assembly language as a series of zeros and ones which have a meaning for the internal logic of the microcontroller.
Lets take for instance the instruction "RETURN" that a microcontroller uses to return from a sub-program.
When the assembler translates it, we get a 14-bit series of zeros and ones which the microcontroller knows how to interpret.

**Example:** RETURN 00 0000 0000 1000

Similar to the above instance, each assembler instruction is interpreted as corresponding to a series of zeros and ones.
The place where this translation of assembly language is found, is called an "execution" file. We will often meet the name "HEX" file. This name comes from a hexadecimal representation of that file, as well as from the suffix "hex" in the title, ex. "test.hex".
Once it is generated, the execution file is read in a microcontroller through a programmer.

An **Assembly Language** program is written in a program for text processing (editor) and is capable of producing an ASCII file on the computer disc or in specialized surroundings such as MPLAB - to be explained in the next chapter.

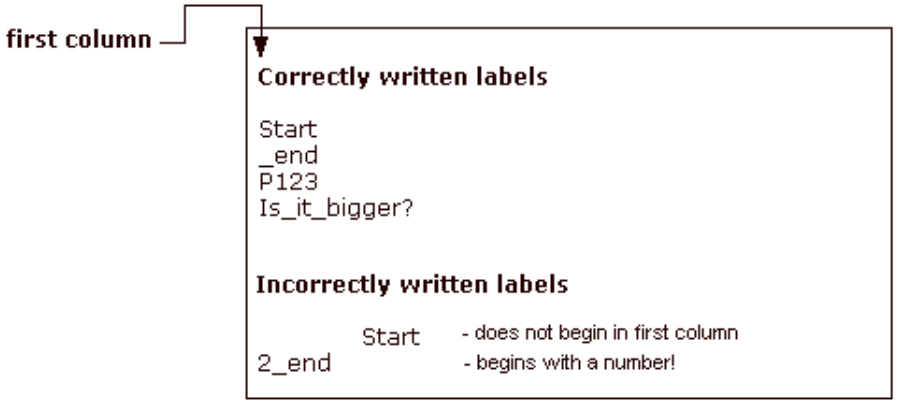
## Assembly language

Basic elements of assembly language are:

- Labels
- Instructions
- Operands
- Directives
- Comments

## Labels

A **Label** is a textual designation (generally an easy-to-read word) for a line in a program, or section of a program where the micro can jump to - or even the beginning of set of lines of a program. It can also be used to execute program branching (such as Goto .....) and the program can even have a condition that must be met for the Goto instruction to be executed. It is important for a label to start with a letter of the alphabet or with an underline "_". The length of the label can be up to 32 characters. It is also important that a label starts in the first clumn.




## Instructions

Instructions are already defined by the use of a specific microcontroller, so it only remains for us to follow the instructions for their use in assembly language. The way we write an instruction is also called instruction "syntax". In the following example, we can recognize a mistake in writing because instructions movlp and gotto do not exist for the PIC16F84 microcontroller.

**Correctly written instructions**

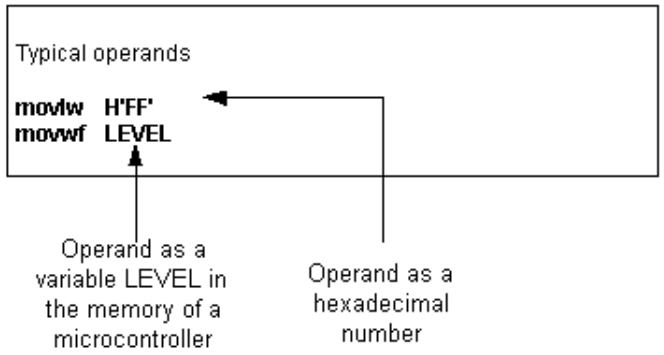
```
movlw     H'01FF'
goto      Start
```

**Incorrectly written instructions**

```
movlp     H'01FF'
gotto     Start
```

## Operands

Operands are the instruction elements for the instruction is being executed. They are usually **registers** or **variables** or **constants**.




## Comments

**Comment** is a series of words that a programmer writes to make the program more clear and legible. It is placed after an instruction, and must start with a semicolon ";".

## Directives

A **directive** is similar to an instruction, but unlike an instruction it is independent on the microcontroller model, and represents a characteristic of the assembly language itself. Directives are usually given purposeful meanings via variables or registers. For example, LEVEL can be a designation for a variable in RAM memory at address 0Dh. In this way, the variable at that address can be accessed via LEVEL designation. This is far easier for a programmer to understand than for him to try to remember address 0Dh contains information about LEVEL.

**Some frequently used directives:**

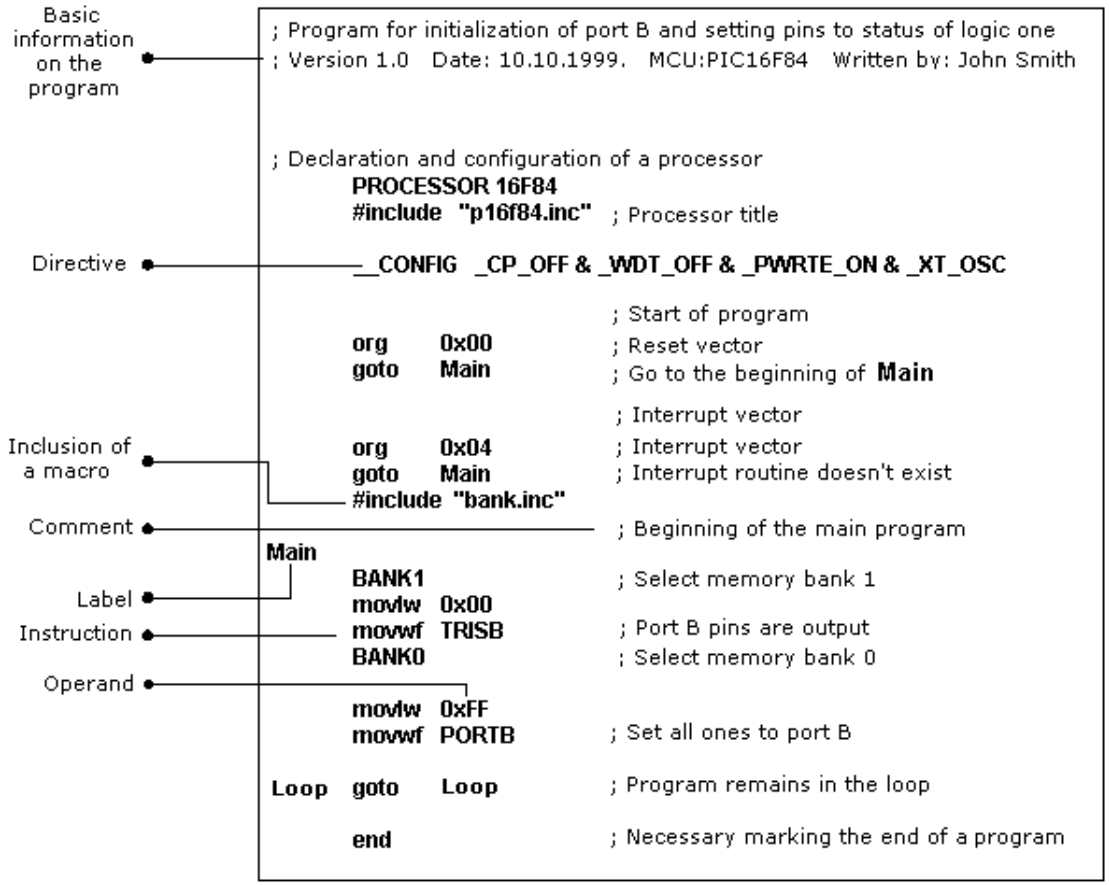
```
PROCESSOR 16F84
#include    "p16f84.inc"

__CONFIG  _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC
```

## An example of a writting program

The following example illustrates a simple program written in assembly language respecting the basic rules.

When writing a program, beside mandatory rules, there are also some rules that are not written down but need to be followed. One of them is to write the name of the program at the beginning, what the program does, its version, date when it was written, type of microcontroller it was written for, and the programmer’s name.




Since this data isn't important for the assembly translator, it is written as **comments**. It should be noted that a comment always begins with a semicolon and it can be placed in a new row or it can follow an instruction.
After the opening comment has been written, the **directive** must be included. This is shown in the example above.

In order to function properly, we must define several microcontroller parameters such as: - type of oscillator,
- whether watchdog timer is turned on, and
- whether internal reset circuit is enabled.
All this is defined by the following directive:

_CONFIG _CP_OFF&_WDT_OFF&PWRTE_ON&XT_OSC

When all the needed elements have been defined, we can start writing a program.
First, it is necessary to determine an address from which the microcontroller starts, following a power supply start-up. This is (org 0x00).
The address from which the program starts if an interrupt occurs is (org 0x04).
Since this is a simple program, it will be enough to direct the microcontroller to the beginning of a program with a "**goto Main**" instruction.

The instructions found in the **Main** select memory bank1 (BANK1) in order to access TRISB register, so that port B can be declared as an output (movlw 0x00, movwf TRISB).

The next step is to select memory bank 0 and place status of logic one on port B (movlw 0xFF, movwf PORTB), and thus the main program is finished.
We need to make another loop where the micro will be held so it doesn't "wander" if an error occurs. For that purpose, one infinite loop is made where the micro is retained while power is connected. The necessary "end" at the end of each program informs the assembly translator that no more instructions are in the program.

# Control directives

## 4.1 #DEFINE Exchanges one part of text for another

**Syntax:**
#define<text> [<another text>]

**Description:**
Each time <text> appears in the program , it will be exchanged for <another text >.

**Example:**

#define turned_on 1
#define turned_off 0

**Similar directives:** #UNDEFINE, IFDEF,IFNDEF

## 4.2 INCLUDE Include an additional file in a program

**Syntax:**
#include <file_name>
#include "file_name"

**Description:**
An application of this directive has the effect as though the entire file was copied to a place where the "include" directive was found. If the file name is in the square brackets, we are dealing with a system file, and if it is inside quotation marks, we are dealing with a user file. The directive "include" contributes to a better layout of the main program.

**Example:**
#include <regs.h>
#include "subprog.asm"

## 4.3 CONSTANT Gives a constant numeric value to the textual designation

**Syntax:**
Constant <name>=<value>

**Description:**
Each time that <name> appears in program, it will be replaced with <value>.

**Example:**
Constant MAXIMUM=100
Constant Length=30

**Similar directives:** SET, VARIABLE

## 4.4 VARIABLE Gives a variable numeric value to textual designation

**Syntax:**
Variable<name>=<value>

**Description:**
By using this directive, textual designation changes with particular value.
It differs from CONSTANT directive in that after applying the directive, the value of textual designation can be changed.

**Example:**
variable level=20

variable time=13

**Similar directives:** SET, CONSTANT

## 4.5 SET Defining assembler variable

**Syntax:**
<name_variable>set<value>

**Description:**
To the variable <name_variable> is added expression <value>. SET directive is similar to EQU, but with SET directive name of the variable can be redefined following a definition.

**Example:**
level set 0
length set 12
level set 45

**Similar directives:** EQU, VARIABLE

## 4.6 EQU Defining assembler constant

**Syntax:**
<name_constant> equ <value>

**Description:**
To the name of a constant <name_constant> is added value <value>

**Example:**
five equ 5
six equ 6
seven equ 7

**Similar instructions:** SET

## 4.7 ORG Defines an address from which the program is stored in microcontroller memory

**Syntax:**
<label>org<value>

**Description:**
This is the most frequently used directive. With the help of this directive we define where some part of a program will be start in the program memory.

**Example:**
```
Start org   0×00
      movlw 0xFF
      movwf PORTB
```

The first two instructions following the first ’org’ directive are stored from address 00, and the other two from address 10.

## 4.8 END End of program

**Syntax:**
end

**Description:**
At the end of each program it is necessary to place ’end’ directive so that assembly translator would know that there are no more instructions in the program.

**Example:**
```
.
.
movlw 0xFF
movwf PORTB
end
```

# Conditional instructions

## 4.9 IF Conditional program branching

**Syntax:**
if<conditional_term>

**Description:**
If condition in <conditional_term> was met, part of the program which follows IF directive would be executed. And if it wasn't, then the part following ELSE or ENDIF directive would be executed.

**Example:**
if level=100
goto FILL
else
goto DISCHARGE
endif

**Similar directives:** #ELSE, ENDIF

## 4.10 ELSE The alternative to 'IF' program block with conditional terms

**Syntax:**
Else

**Description:**
Used with IF directive as an alternative if conditional term is incorrect.

**Example:**
If time< 50
goto SPEED UP
else goto SLOW DOWN
endif

**Similar instructions:** ENDIF, IF

## 4.11 ENDIF End of conditional program section

**Syntax:**
endif

**Description:**
Directive is written at the end of a conditional block to inform the assembly translator that it is the end of the conditional block

**Example:**
If level=100
goto LOADS
else
goto UNLOADS
endif

**Similar directives:** ELSE, IF

## 4.12 WHILE Execution of program section as long as condition is met

**Syntax:**
while<condition>
.
endw

**Description:**
Program lines between WHILE and ENDW would be executed as long as condition was met. If a condition stopped being valid, program would continue executing instructions following ENDW line. Number of instructions between WHILE and ENDW can be 100 at the most, and number of executions 256.

**Example:**
While i<10
i=i+1
endw

## 4.13 ENDW End of conditional part of the program

**Syntax:**
endw

**Description:**
Instruction is written at the end of the conditional WHILE block, so that assembly translator would know that it is the end of the conditional block

**Example:**
while i<10
i=i+1

endw

**Similar directives:** WHILE

## 4.14 IFDEF Execution of a part of the program if symbol was defined

**Syntax:**
ifdef<designation>

**Description:**
If designation <designation> was previously defined (most commonly by #DEFINE instruction), instructions which follow would be executed until ELSE or ENDIF directives are not would be reached.

**Example:**
#define test
.
ifdef test ;how the test was defined
.....; instructions from these lines would execute
endif

**Similar directives:** #DEFINE, ELSE, ENDIF, IFNDEF, #UNDEFINE

## 4.15 IFNDEF Execution of a part of the program if symbol was defined

**Syntax:**
ifndef<designation>

**Description:**
If designation <designation> was not previously defined, or if its definition was erased with directive #UNDEFINE, instructions which follow would be executed until ELSE or ENDIF directives would be reached.

**Example:**
#define test
.....
#undefine test
.....
ifndef test ;how the test was undefined
..... .; instructions from these lines would execute
endif

**Similar directives:** #DEFINE, ELSE, ENDIF, IFDEF, #UNDEFINE

# Data Directives

## 4.16 CBLOCK Defining a block for the named constants

**Syntax:**
Cblock [<term>]
<label>[:<increment>], <label>[:<increment>].....
endc

**Description:**
Directive is used to give values to named constants. Each following term receives a value greater by one than its precursor. If <increment> parameter is also given, then value given in <increment> parameter is added to the following constant.
Value of <term> parameter is the starting value. If it is not given, it is considered to be zero.

**Example:**
```
Cblock 0x02
First, second, third  ;first=0x02, second=0x03, third=0x04
endc

cblock 0x02
first : 4, second : 2, third  ;first=0x06, second=0x08, third=0x09
endc
```

**Similar directives:** ENDC

## 4.17 ENDC End of constant block definition

**Syntax:**
endc

**Description:**
Directive was used at the end of a definition of a block of constants so assembly translator could know that there are no more constants.

**Similar directives:** CBLOCK

## 4.18 DB Defining one byte data

**Syntax:**
[<label>]db <term> [, <term>,.....,<term>]

**Description:**
Directive reserves a byte in program memory. When there are more terms which need to be assigned a byte each, they will be assigned one after another.

**Example:**
db ’t’, 0×0f, ’e’, ’s’, 0×12

**Similar instructions:** DE, DT

## 4.19 DE Defining the EEPROM memory byte

**Syntax:**
[<term>] de <term> [, <term>,....., <term>]

**Description:**
Directive is used for defining EEPROM memory byte. Even though it was first intended only for EEPROM memory, it could be used for any other location in any memory.

**Example:**
org H’2100’
de "Version 1.0" , 0

**Similar instructions:** DB, DT

## 4.20 DT Defining the data table

**Syntax:**
[<label>] dt <term> [, <term>,....., <term>]

**Description:**
Directive generates RETLW series of instructions, one instruction per each term.

**Example:**
dt "Message", 0
dt first, second, third

**Similar directives:** DB, DE

## Configurational directives

### 4.21 _CONFIG Setting the configurational bits

**Syntax:**
_ _config<term> or_ _config<address>,<term>

**Description:**
Oscillator, watchdog timer application and internal reset circuit are defined. Before using this directive, the processor must be defined using PROCESSOR directive.

**Example:**
_CONFIG _CP_OFF&_WDT_OFF&_PWRTE_ON&_XT_OSC

**Similar directives:** _IDLOCS, PROCESSOR

### 4.22 PROCESSOR Defining microcontroller model

**Syntax:**
Processor <microcontroller_type>

**Description:**
Instruction sets the type of microcontroller where programming is done.

**Example:**
processor 16F84

## Assembler arithmetic operators

Operator Description Example

| Operator | Description | Example |
|---|---|---|
| **$** | Current status of program counter | goto $ +3 |
| **(** | Left bracket | 1 + ( d * 4 ) |
| **)** | Right bracket | ( Length + 1 ) * 256 |
| **!** | NE (logic complement) | if ! ( a - b ) |
| - | Complement | flags = -flags |
| - | Negation (second complement) | -1 * Length |
| **high** | Returns higher byte | movlw high CTR_Table |
| **low** | Returns lower byte | movlw low CTR_Table |
| * | Multiplying | a = b * c |
| **/** | Subdividing | a = b / c |
| **%** | Subdividing by module | entry_len = tot_len % 16 |
| + | Addition | tot_len = entry_len * 8 + 1 |
| - | Subtraction | entry_len = ( tot - 1 ) / 8 |
| << | Moving to the left | val = flags << 1 |
| >> | Moving to the right | val = flags >> 1 |
| >= | Higher than, or equal | if entry_idx >= num_entries |
| > | Higher than | if entry_idx > num_entries |
| < | Lesser than | if entry_idx < num_entries |
| <= | Lesser than, or equal | if entry_idx <= num_entries |
| == | Equal | if entry_idx == num_entries |
| **!=** | Not equal | if entry_idx != num_entries |
| **&** | Operation AND on bits | flags = flags & ERROR_BIT |
| **^** | Exclusive OR on bits | flags = flags ^ ERROR_BIT |
| **\|** | Logic OR on bits | flags = flags \| ERROR_BIT |
| **&&** | Logic AND | if (len == 512) && (b == c) |
| **\|\|** | Logic OR | if (len == 512) \|\| (b == c) |
| = | Equal | entry_index = 0 |
| += | Add and assign | entry_index += 1 |
| -= | Subtract and assign | entry_index -= 1 |
| *= | Multiply and assign | entry_index *= entry_length |
| **/=** | Divide and assign | entry_total /= entry_length |
| **%=** | Divide at module and assign | entry_index %= 8 |
| <<= | Move to the left and assign | flags <<= 3 |
| >>= | Move to the right and assign | flags >>= 3 |
| **&=** | Logic AND and assign | flags &= ERROR_FLAG |
| \|= | Logic OR on bits and assign | flags \|= ERROR_FLAG |
| ^= | Exclusive OR on bits and assig | flags^= ERROR_FLAG |
| ++ | Increment by one | i ++ |
| - - | Decrease by one | i -- |

# Files created as a result of program translation

As a result of the process of translating a program written in assembler language we get files like:

- Executing file (Program_Name.HEX)
- Program errors file (Program_Name.ERR)
- List file (Program_Name.LST)

The first file contains translated program which was read in microcontroller by programming. Its contents can not give any information to programmer, so it will not be considered any further.
The second file contains possible errors that were made in the process of writing, and which were noticed by assembly translator during translation process. Errors can be discovered in a "list" file as well. This file is more suitable though when program is big and viewing the 'list' file takes longer.
The third file is the most useful to programmer. Much information is contained in it, like information about positioning instructions and variables in memory, or error signalization.

Example of 'list' file for the program in this chapter follows. At the top of each page is stated information about the file name, date when it was translated, and page number. First column contains an address in program memory where a instruction from that row is placed. Second column contains a value of any variable defined by one of the directives : SET, EQU, VARIABLE, CONSTANT or CBLOCK. Third column is reserved for the form of a translated instruction which PIC is executing. The fourth column contains assembler instructions and programmer's comments. Possible errors will appear between rows following a line in which the error occured.

```
LOC  OBJECT CODE     LINE SOURCE TEXT
  VALUE

                     00001   ;Program for initialization of port B and setting its pins
                     00002   ;to the state of logic one
                     00003   ;Version: 1.0  Date: 10.05.2000.      MCU: PIC16F84  Written
                     00004   ;by:  Petar Petrovic
                     00005
                     00006   ;Declaration and configuration of the processor
                     00007   PROCESSOR 16F84
                     00008   #include "p16f84.inc"  ;Processor title
                     00001   LIST
                     00002   ;P16F84.INC  Standard Header File, Version 2.00  Microchip
                             ;Technology, Inc.
                     00136   LIST
                     00009
2007 3FF1            00010   __CONFIG    _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC
                     00011
000C                 00012   CONSTANT BASE = 0x0c
                     00013
                     00014   ;Start of a program
0000                 00015   org  0x00       ;Reset vector
0000 2805            00016   goto  Main      ;Go to the beginning of the main program
                     00017
                     00018   ;Interrupt vector
0004                 00019   org      0x04   ;Interrupt vector
0004    2805         00020   goto     Main   ;Interrupt routine does not exist
                     00021
                     00022 ;Beginning of the main program
                     00023  #include "Bank.inc"     ; File with macros
                     00001 ;*****************************************************************
                     00002 ;         Makros BANK0 and BANK1
                     00003 ;*****************************************************************
                     00004
0000    0010         00005   W_Temp           set     BASE+4
0000    0011         00006   Stat_Temp       set     BASE+5
0000    0012         00007   Option_Temp     set     BASE+6
                     00008
                     00009
                     00010   BANK0   macro
                     00011   bcf     STATUS,RP0      ; Select memory bank 0
                     00012   endm
                     00013
                     00014   BANK1   macro
                     00015   bsf     STATUS,RP0      ; Select memory bank 1
                     00016   endm
                     00017
0005                 00024   Main
                     00025   BANK1                   ; Select memory bank 1
0005    1683         M       bsf     STATUS,RP0      ; Select memory bank 1
0006    3000         00026   movlw    0x00
Message[302]: Register in operand not in bank 0. Ensure that bank bits are
correct.
0007    0086         00027   movwf   TRISB           ;Port B pins are output
                     00028
                     00029   BANK0                   ;Select memory bank 0
0008    1283         M       bcf     STATUS,RP0      ;Select memory bank 0
0009    30FF         00030   movlw   0xFF
000A    0086         00031   movwf   PORTB           ;Set all ones to port B
                     00032
000B    280B         00033   Loop    goto    Loop    ; Program stays in the loop
                     00034
                     00035   END             ;Necessary marking the end of a program

MEMORY USAGE MAP ('X' = Used,  '-' = Unused)

0000 : X---XXXXXXXX---- ---------------- ---------------- ----------------
2000 : -------X-------- ---------------- ---------------- ----------------

All other memory blocks unused.

Program Memory Words Used:    9
Program Memory Words Free:    1015

Errors:          0
Warnings:        0 reported,      0 suppressed
Messages:        1 reported,      0 suppressed
```

At the end of the "list" file there is a table of symbols used in a program. Useful element of ’list’ file is a graph of memory utilization. At the very end, there is an error statistic as well as the amount of remaining program memory.

## Macros

Macros are a very useful element in assembly language. They could briefly be described as "user defined group of instructions which will enter assembler program where macro was called". It is possible to write a program even without using macros. But with their use written program is much more readable, especially if more programmers are working on the same program together. Macros have the same purpose as functions of higher program languages.

**How to write them:**

<label> macro [<argument1>,<argument2>,.....<argumentN>]
.....
.....
endm

From the way they were written, we could be seen that macros can accept arguments, too which is also very useful in programming. Whenever argument appears in the body of a macro, it will be replaced with the <argumentN> value.

**Example:**

```
NA_PORTB        macro ARG1
                BANK0           ;Select memory bank 0
                movlw ARG1      ;Value from ARG1 argument
                                ;is stored in working register
                movwf PORTB     ;value from ARG1
                                ; argument placed on port B
                endm            ;macro ended
```

The above example shows a macro whose purpose is to place on port B the ARG1 argument that was defined while macro was called. Its use in the program would be limited to writing one line: ON_PORTB 0xFF , and thus we would place value 0xFF on PORTB. In order to use a macro in the program, it is necessary to include macro file in the main program with instruction include "macro_name.inc". Contents of a macro is automatically copied onto a place where this instruction was written. This can be best seen in a previous list file where file with macros "bank.inc" was copied below the line #include"bank.inc"