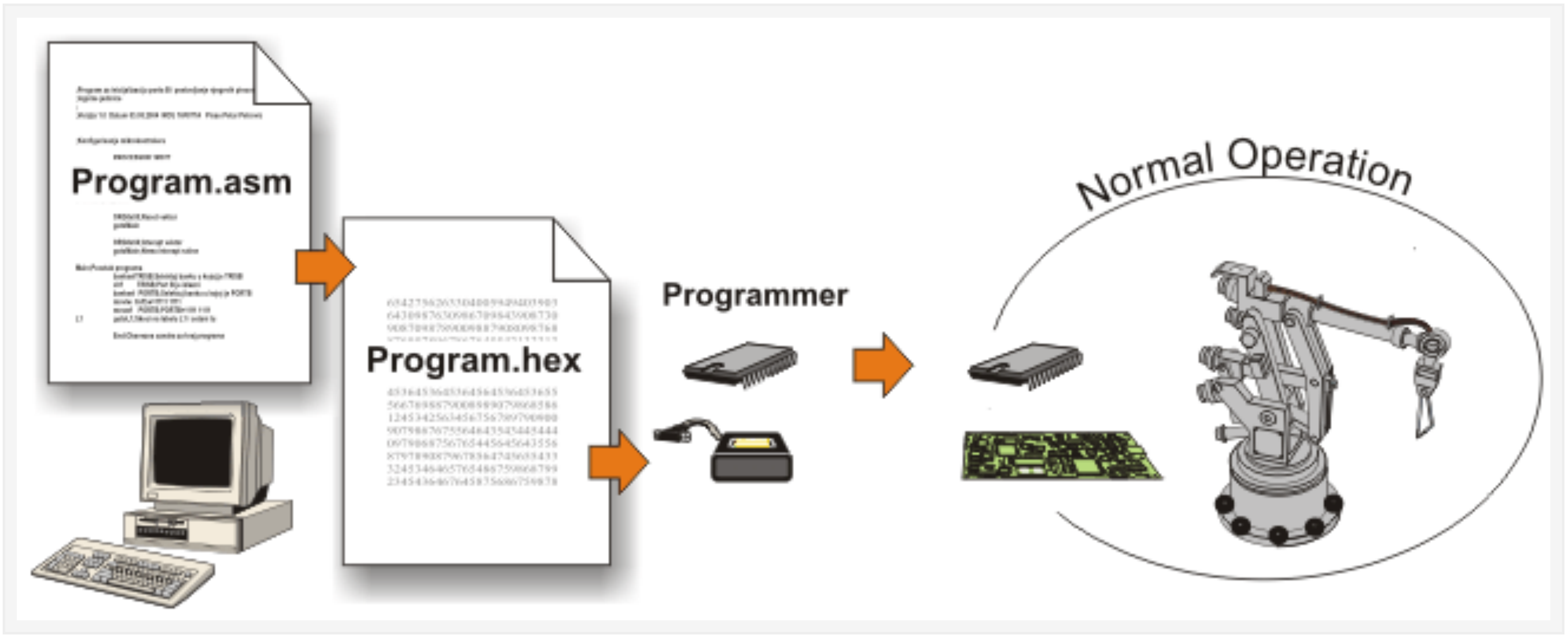
APPENDIX A. PROGRAMMING A MICROCONTROLLER

MIKROELEKTRONIKA

Microcontrollers and humans communicate through the medium of the programming language called Assembly language. The word Assembler itself does not have any deeper meaning, it corresponds to the names of other languages such as English or French. More precisely, assembly language is only a passing solution. In order that the microcontroller can understand a program written in assembly language, it must be compiled into a language of zeros and ones. Assembly language and Assembler do not have the same meaning. The first one refers to the set of rules used for writing program for the microcontroller, while the later refers to a program on a personal computer used to translate assembly language statements into the language of zeros and ones. A compiled program is also called Machine Code. A “Program” is a data file stored on a computer hard disc (or in memory of the microcontroller, if loaded) and written according to the rules of assembly or some other programming language. Assembly language is understandable for humans because it consists of meaningful words and symbols of the alphabet. Let us take, for example the command “RETURN” which is, as its name indicates, used to return the microcontroller from a subroutine. In machine code, the same command is represented by a 14-bit array of zeros and ones understandable by the microcontroller. All assembly language commands are similarly compiled into the corresponding array of zeros and ones. A data file used for storing compiled program is called an “executive file”, i.e. “HEX data file”. The name comes from the hexadecimal presentation of a data file and has a suffix of “hex” as well, for example “probe.hex”. After has been generated, the data file is loaded into the microcontroller using a programmer. Assembly language programs may be written in any program for text processing (editor) able to create ASCII data files on a hard disc or in a specialized work environment such as MPLAB described later.



ELEMENTS OF ASSEMBLY LANGUAGE

A program written in assembly language consists of several elements being differently interpreted while compiling the program into an executable data file. The use of these elements requires strict rules and it is necessary to pay special attention to them during program writing in order to avoid errors.

ASSEMBLY LANGUAGE SYNTAX

As mentioned, it is necessary to observe some specific rules in order to enable the process of compiling into executive HEX code to run without errors. Compulsory rules explaining how sequences of expressions are put together to form the statements that make up an assembly language program are called syntax. There are only several of them:

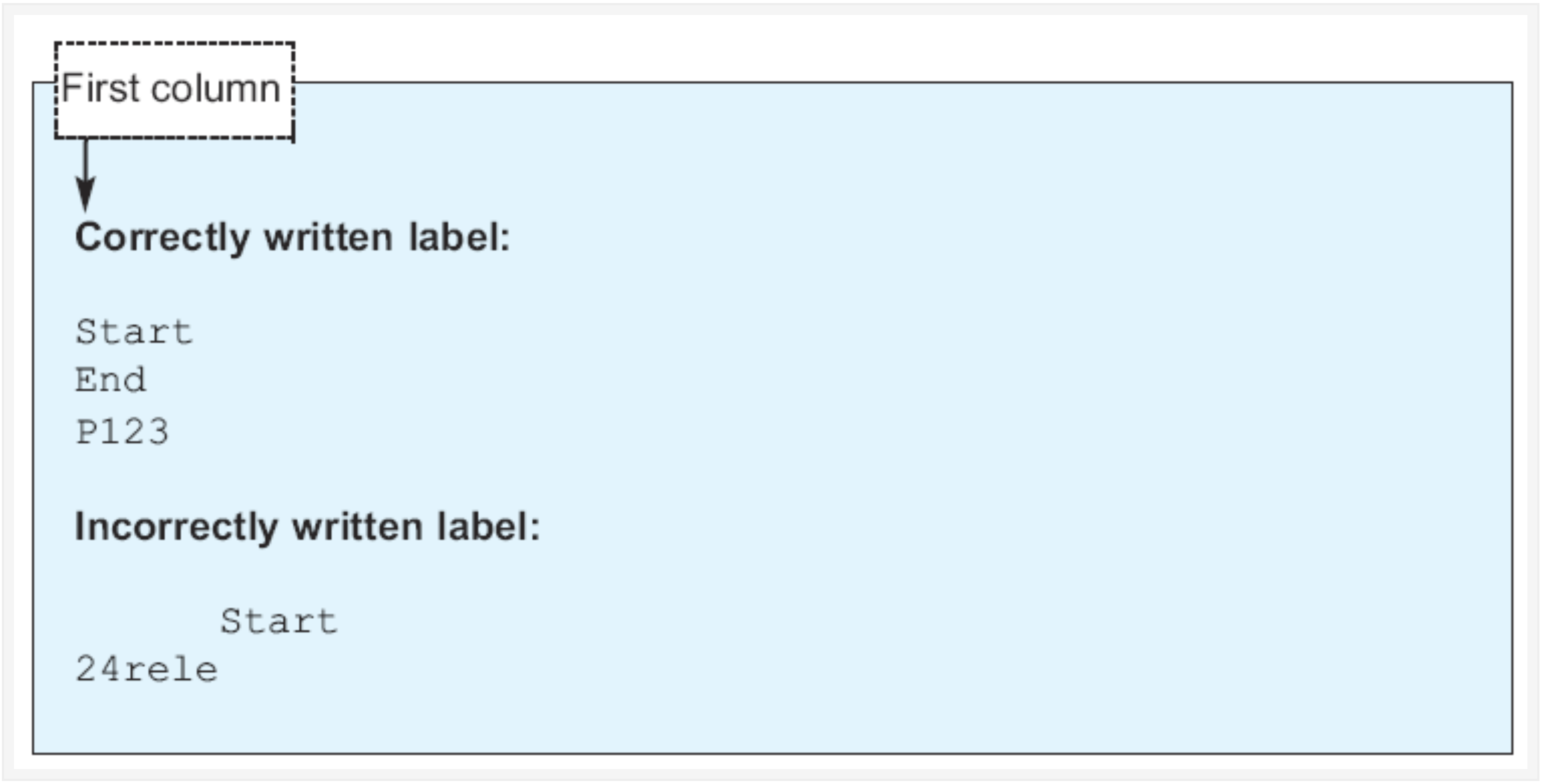
- Every program line may consist of a maximum of 255 characters;
- Every program line that is to be compiled must start with a symbol, label, mnemonics or directive;
- Text following the mark ";" in a program line represents a comment which is ignored by the assembler (not compiled); and
- All the elements of one program line (labels, instructions etc.) must be separated by at least one space character. For the sake of better clearness, a push-button TAB is commonly used instead of it, so that it is easy to delimit columns with labels, directives etc. in a program.

LABELS

A label represents a textual version of some address in ROM or RAM memory. Each label has to start in the first column with a letter of alphabet or "_" and may consist of maximum of 32 characters. Besides, it is easily used:

- It is sufficient to enter the name of a label instead of a 16-bit address in instruction which calls some subroutine or a jump. The label with the same name should also be written at the beginning of a program line in which a subroutine starts or where a jump should be executed. As a general rule, labels have easily recognizable names.

During program compiling, the assembler will automatically replace the labels by the corresponding addresses.



Correctly written label:

```
Start
End
P123
```

Incorrectly written label:

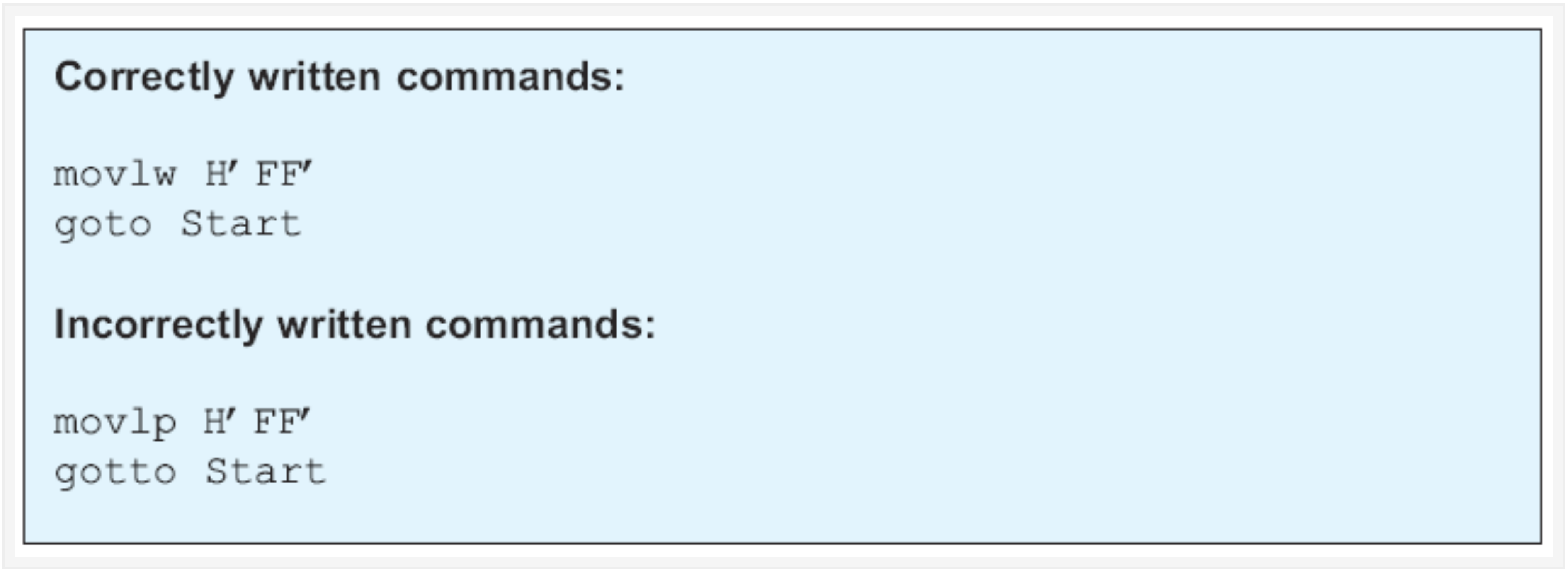
```
      Start
24rele
```

COMMENTS

Acomment is often an explanatory text written by the programmer in order to make a program clearer and easier to understand. It is not necessary to comment every line. When three or four lines of code work together to accomplish some higher level task, it is better to have a single higher level comment for the group of lines. Therefore, it is added if needed and has to start with ";". Comments added to assembly source code are not compiled into machine code.

INSTRUCTIONS

Instructions are defined for each microcontroller family by the manufacturer. Therefore, it is up to the user to follow the rules of their usage. The way of writing instructions is also called instruction syntax. The instructions "`movlp`" and "`gotto`", in the following example, are recognized by the PIC16F887 microcontroller as an error since they are not correctly written.



Correctly written commands:

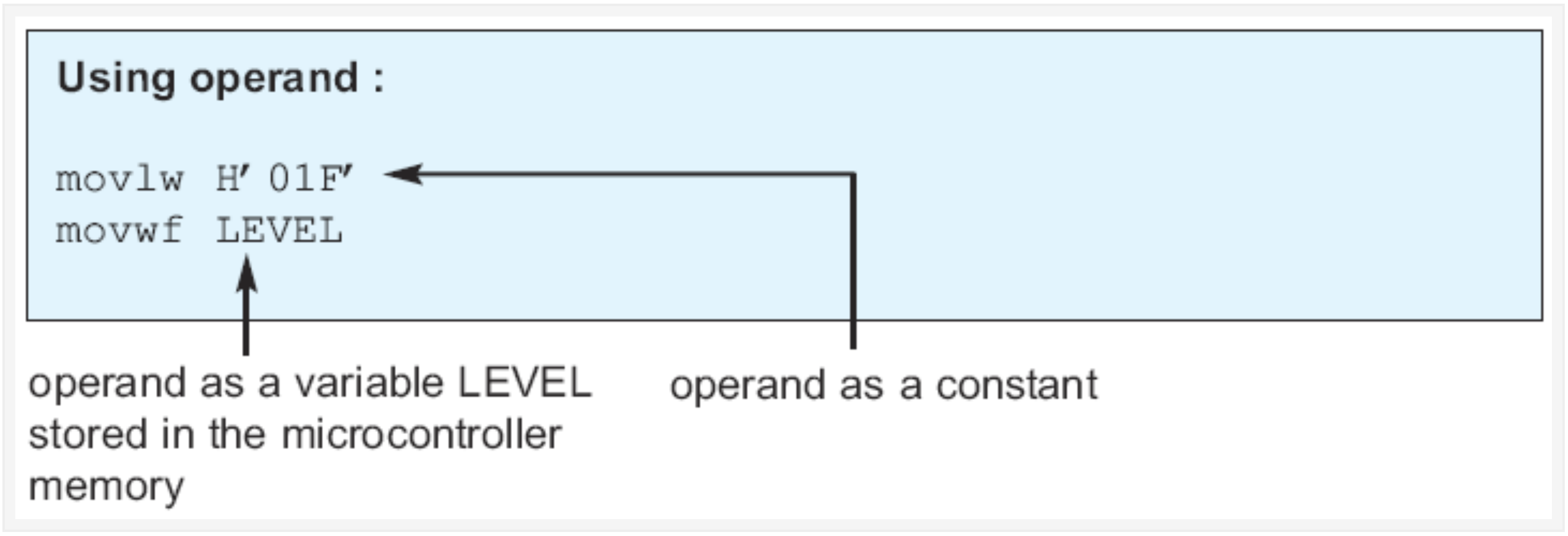
```
movlw H' FF'
goto  Start
```

Incorrectly written commands:

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

OPERANDS

An operand is a value (an argument) upon which the instruction, named by mnemonic, operates. The operands may be a register, a variable, a literal constant, a label or a memory address.



Using operand :

```
movlw  H' 01F'
movwf  LEVEL
```

operand as a variable LEVEL stored in the microcontroller memory
operand as a constant

DIRECTIVES

Unlike instructions being written to on-chip program memory after compilation, directives are commands of assembly language itself and do not directly affect the operation of the microcontroller. Some of them must be used in every program while others are only used to facilitate or enhance the operation. Directives are written to the column reserved for instructions. The rule which must be observed allows only one directive per program line.

This section covers only a few of the most commonly used directives. It would certainly take up too much space and time to describe all the directives recognized by the MPLAB program. Anyway, a complete list containing all directives which the MPLAB assembler can understand is provided in *Help*.

PROCESSOR Directive

This directive must be written at the beginning of each program. It defines the type of the microcontroller which the program is written for. For example:

```
Processor 16f887
```

EQU directive

This directive is used to replace a numeric value by a symbol. In this way, some a specific location in memory is assigned a name. For example:

```
MAXIMUM EQU H’25’
```

This means that a memory location at address 25 (hex.) is assigned the name “MAXIMUM”. Every appearance of the label “MAXIMUM” in the program will be interpreted by the assembler as the address 25 (MAXIMUM = H’25’). Symbols may be defined this way only once in a program. That this directive is mostly used at the beginning of the program.

ORG directive

This directive specifies a location in program memory where the program following directive is to be placed. For example:

```
            ORG      0x100
START       ...      ...
            ...
            ORG      0x1000
TABLE       ...
            ...
```

This program starts at location 0x100. The table containing data is to be stored at location 1024 (1000h).

END directive

Each program must be ended by using this directive. Once a program encounters this directive, the assembler immediately stops compiling. For example:

```
...
END ;End of program
```

\$INCLUDE directive

The name of this directive fully indicates its purpose. During compiling, it enables the assembler to use data contained in another file on a computer hard disc. For example:

```
...
#include <p16f887.inc>
```

CBLOCK and ENDC directives

All variables (their names and addresses) that will be used in a program must be defined at the beginning of the program. Because of this it is not necessary to specify the address of each specified variable later in the program. Instead, it is enough to specify the address of the first one by using directive CBLOCK and list all others afterwards. The compiler automatically assigns these variables the corresponding addresses as per the order they are listed. Lastly, the directive ENDC indicates the end of the list of variables.

```
CBLOCK    0x20
          START    ; address 0x20
          RELE     ; address 0x21
          STOP     ; address 0x22
          LEFT     ; address 0x23
          RIGHT    ; address 0x24
ENDC
          ...
```

IF, ENDIF and ELSE directives

These directives are used to create so called conditional blocks in a program. Each of these blocks starts with the directive IF and ends with the directive ENDIF or ELSE. A statement or a symbol (in parentheses) following the directive IF represents a condition which determines which part of the program is to be compiled:

- If the statement is correct or the value of a symbol is equal to one, program compiles all instructions written before directive ELSE or ENDIF; and
- If the statement is not correct or the value of a symbol is equal to zero, only instructions written after directives ELSE or ENDIF are to be compiled.

Example 1:

```
IF    (VERSION>3)
       CALL    Table_2
       CALL
ENDIF
       ...
```

If the program is released after the version 3 (statement is right) then subroutines “Table 2” and “Extension” are executed. If the statement in parentheses is wrong (VERSION<3), two instructions calling subroutines are ignored and will not be compiled therefore.

Example 2:

If the value of symbol “Model” is equal to one then first two instructions after directive IF are compiled as well as instructions after directive ENDIF (all instructions between ELSE and ENDIF are ignored). Otherwise, if Model=0 then instructions between IF and ELSE are ignored, whereas instructions after directive ELSE are compiled.

```
IF    (Model)
      MOVFW      BUFFER
      MOVWF      MAXIMUM
ELSE
      MOVFW      BUFFER1
      MOVWF      MAXIMUM
ENDIF
      ...
```

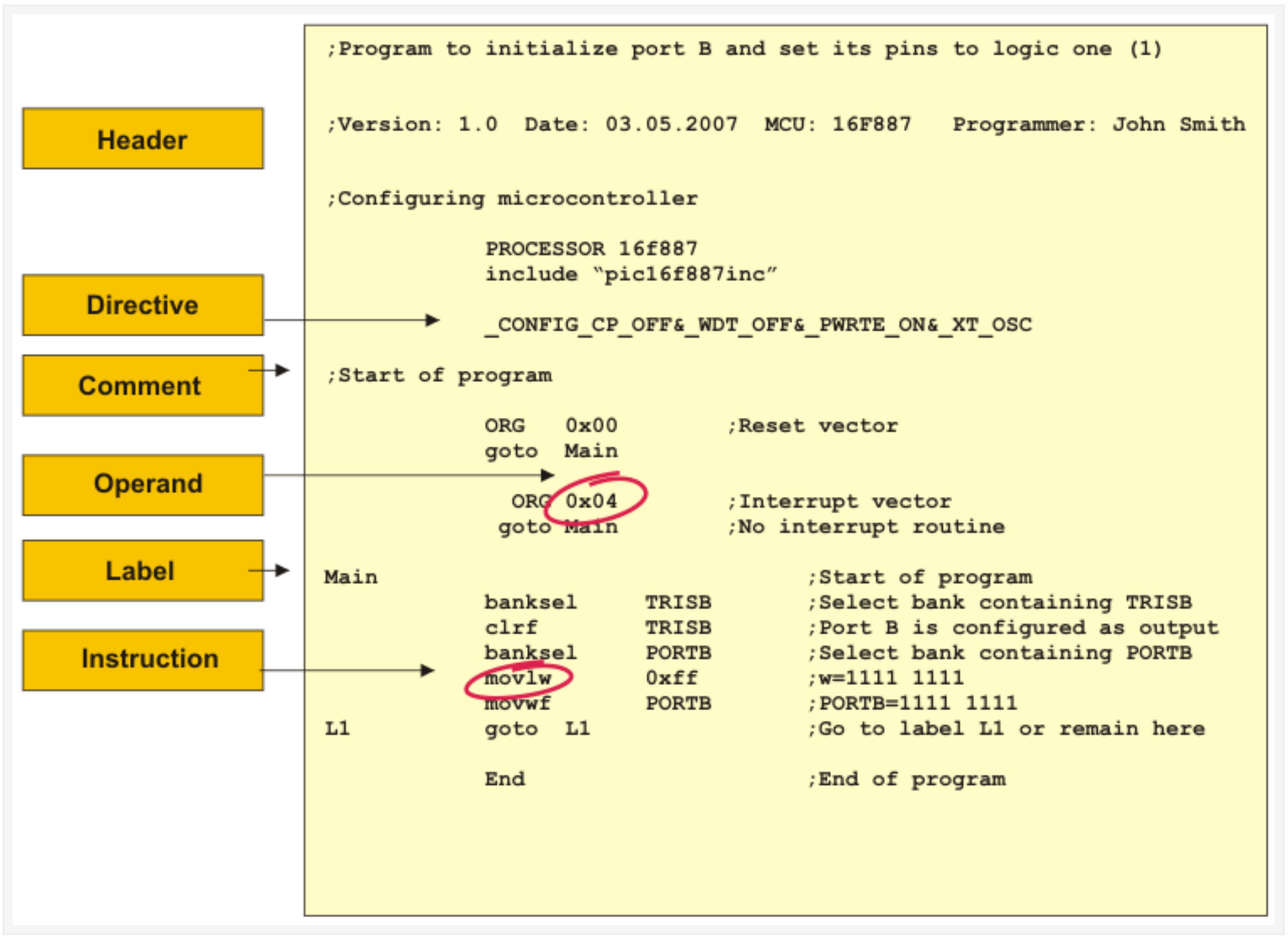
BANKSEL directive

In order to access an SFR register it is necessary to select the appropriate bank in RAM memory by using bits RP0 and RP1 of the STATUS register. This directive is used in this case. Simply, since “inc” data file contains the list of all registers along with their addresses, the assembler knows which bank corresponds to which register. After encountering this directive, assembler selects the bits RP0 and RP1 for the specified register on its own. For example:

```
          ...
BANKSEL   TRISB
          CLRF TRISB
          MOVLW B'01001101'
BANKSEL   PORTB
          MOVWF PORTB
          ...
```

EXAMPLE OF HOW TO WRITE A PROGRAM

The following example illustrates what a simple program written in assembly language looks like.



```
;Program to initialize port B and set its pins to logic one (1)

;Version: 1.0  Date: 03.05.2007  MCU: 16F887   Programmer: John Smith

;Configuring microcontroller

            PROCESSOR 16f887
            include "pic16f887inc"

            _CONFIG_CP_OFF&_WDT_OFF&_PWRTE_ON&_XT_OSC

;Start of program

            ORG   0x00          ;Reset vector
            goto  Main

              ORG 0x04          ;Interrupt vector
             goto Main          ;No interrupt routine

Main                                  ;Start of program
            banksel     TRISB         ;Select bank containing TRISB
            clrf        TRISB         ;Port B is configured as output
            banksel     PORTB         ;Select bank containing PORTB
            movlw       0xff          ;w=1111 1111
            movwf       PORTB         ;PORTB=1111 1111
L1          goto  L1                  ;Go to label L1 or remain here

            End                       ;End of program
```

Apart from the regular rules of assembly language, there are also some unwritten rules which should be observed during program writing. One of them is to write in a few words at the beginning of a program what the program's name is, what it is used for, version, release date, type of the microcontroller it is written for and the name of the programmer. Since this information is not of importance for the assembler, it is written as a comment which always starts with semicolon ';' and can be written in a new line or immediately after a command.

After writing this general comment, it is time to select the microcontroller by using directive PROCESSOR. This directive is followed by another one used to include all the definitions of the PIC16F887 microcontroller's internal registers in the program. These definitions are nothing but the ability to address port B and other registers as PORTB instead of 06h, which makes the program clearer and more legible.

In order that the microcontroller will operate properly, a several parameters such as the type of oscillator, state of the watch-dog and internal reset circuit must be defined. It is done by utilizing the following directive:

```
_CONFIG _CP_OFF&_WDT_OFF&PWRTE_ON&XT_OSC
```

When all necessary elements are defined, the process of program writing can start. First and foremost, it is necessary to specify the address from which the microcontroller starts when the power goes on (`org 0x00`) as well as the address from which the program proceeds with execution if an interrupt occurs (`org 0x04`). Since this program is very simple, it is enough to use command “`goto Main`” in order to direct the microcontroller to the beginning of the program. The next command selects memory bank 1 in order to enable access to the TRISB register to configure port B as output (`banksel TRISB`). The main program ends by selecting memory bank 0 and setting all port B pins to logic one (1)(`movlw 0xFF`, `movwf PORTB`).

It is necessary to create a loop to keep program from “getting lost” in case an error occurs. For this purpose, there is an endless loop executed all the time while the microcontroller is switched on.

“`end`” is required at the end of every program to inform the assembler that there are no more commands to be compiled.

DATA FILES RESULTING FROM PROGRAM COMPILING

The result of compiling a program written in assembly language are data files. The most important and most commonly used data files are:

- Executive data file (Program_Name.HEX);
- Error data file (Program_Name.ERR); and
- List data file (Program_Name.LST).

The first file contains compiled program which is loaded into the microcontroller. Its contents give no information of importance to the programmer so it will not be discussed here. The second file contains errors made in writing process and detected by the compiler during compiling process. Errors can be detected in list data file, which takes more time, so the error data file is more suitable for long programs.

The third file is the most useful for the programmer. It contains lots of information on commands and variables locations in on-chip memory as well as error signalization. There is a symbol table at the end of each data file list containing all the symbols used in a program. Other useful elements of list data file are memory usage maps and error statistics provided at the very end of the file list.

MACROS AND SUBROUTINES

The same sequence of computing instructions is usually used repeatedly within a program. Assembly language is very demanding. The programmer is required to take care of the last little detail when writing a program, because only one wrong command or label name may cause the program to not work properly or it may not work at all. Therefore, it is less tedious and less error-prone to use a sequence of instructions as a single program statement which works properly for sure. To implement this idea, macros and subroutines are used.

MACROS

A macro contains programmer-defined symbols that stand for a sequence of text lines. It is defined by using directive **macro** which names macro and arguments if needed. Macro must be defined prior it is used. Once a macro has been defined, its name may be used in the program.When the assembler encounters macro’s name, it replaces it by the appropriate sequence of instructions and processes them just as though they have appeared in the program. Many different macro-instructions are available for various purposes, eliminating some of the repetitiveness of the programming, as well as simplifying the writing, reading and understanding of the program. The simplest use of macros may be giving a name to an instruction sequence being repeated. Let us take, for example, global interrupt enable procedure, SFRs' bank selection.

```
macro_name macro arg1, arg2...
           ...
           sequence of instructions
           ...
           endm
```

The following example shows four macros. The first two macros select banks, the third one enables interrupt, whereas the fourth one disables interrupt.

```
bank0 macro                ; Macro bank0
           bcf STATUS, RP0 ; Reset RP0 bit
           bcf STATUS, RP1 ; Reset RP1 bit
           endm            ; End of macro
bank1 macro                ; Macro bank1
           bsf STATUS, RP0 ; Set RP0 bit
           bcf STATUS, RP1 ; Reset RP1 bit
           endm            ; End of macro
enableint macro            ; Global interrupt enable
           bsf INTCON,7    ; Set bit
           endm            ; End of macro
disableint macro           ; Global interrupt disable
           bcf INTCON,7    ; Reset bit
           endm            ; End of macro
```

Macros defined in this way are saved in a particular data file with extension INC which stands for INCLUDE data file. As seen, these four macros do not have arguments. However, macros may include arguments if needed.

The following example shows macros with arguments. Pin is configured as input if the corresponding bit of the TRIS register is set to logic one (bank1). Otherwise, it is configured as output.

```
input macro arg1,arg2      ;Macro Input
       bank1               ;Bank containing TRIS registers
       bsf arg1,arg2       ;Set the specified bit (1=Input)
       bank0               ;Macro for bank 0 selection
       endm                ;End of macro

output macro arg1,arg2     ;Macro Output
       bank1               ;Bank containing TRIS registers
       bcf arg1,arg2       ;Clear the specified bit (0=Output)
       bank0               ;Macro for bank 0 selection
       endm                ;End of macro
```

Macro with arguments may be called in the following way:

```
...
output TRISB,7 ;Pin RB7 is configured as output
...
```

When calling this macro, the first specified argument TRISB is replaced by the first argument **arg1** in macro definition. Similarly, number 7 is replaced by the argument **arg2**, and the following code is generated:

```
...
bsf STATUS, RP0   ;Set RP0 bit = BANK1
bcf STATUS, RP1   ;Reset RP0 bit = BANK1

bcf TRISB,7       ;Configure RB7 as output

bcf STATUS,RP0    ;Clear RP0 bit = BANK0
bcf STATUS,RP1    ;Clear RP1 bit = BANK0
...
```

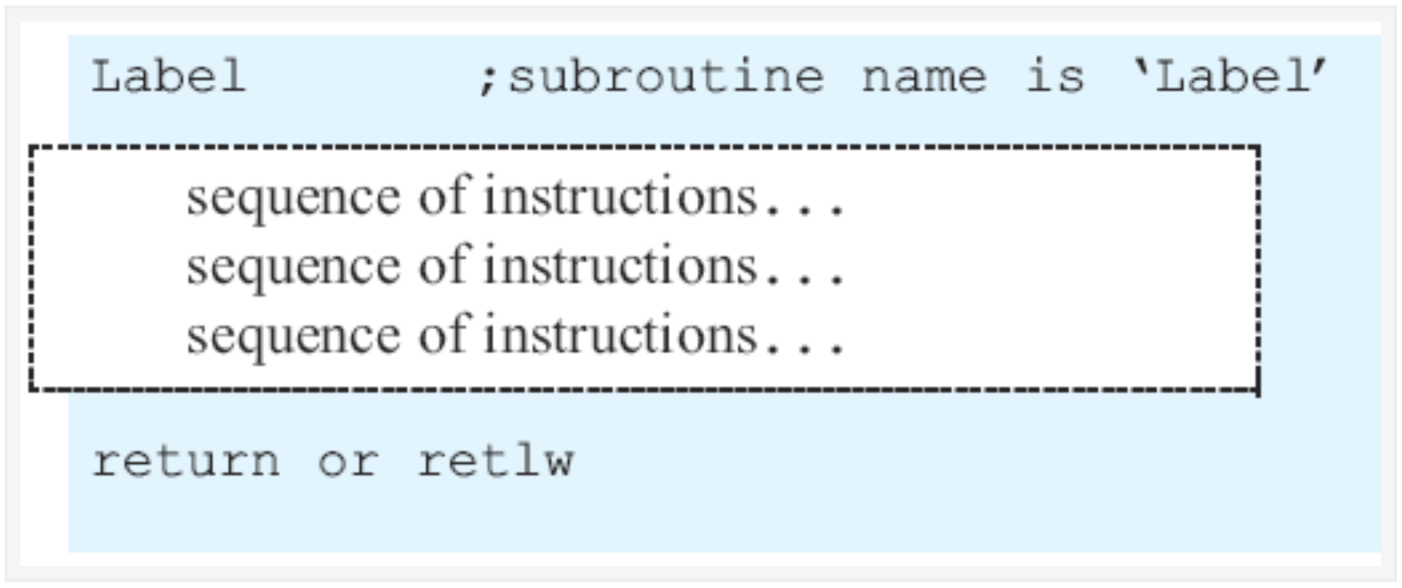
It is clear at first sight that the program becomes more legible and flexible by using macros. The main disadvantage of macro is that it occupies a lot of memory space because every macro name in a program is replaced by its predefined code. Owing to the fact that programs often use macro, everything is more complicated if it is long.

```
callc macro label  ;Macro callc
local Exit         ;Define local Label within macro
      bnc   Exit   ;If C=0 jump to Exit
      call  label  ;If C=1 call subroutine at address Label(out of macro)
Exit               ;Local Label within macro
      endm         ;End of macro
```

In the event that a macro has labels, they must be defined as local ones by using directive local. The given example contains macro which calls a subroutine (`call label` in this case) if the *Carry* bit of the STATUS register is set. Otherwise, the first following instruction is executed.

SUBROUTINES

Asubroutine contains a sequence of instructions, begins with a label (subroutine_name) and ends with command *return* or *retlw*. The main difference comparing to macro is that subroutine is not replaced by its code in the program, but program jumps to subroutine to execute it. It happens every time the assembler encounters command *call Subroutine_name* in the program. On the command *return*, it leaves a subroutine and continues execution from where it left off the main program. Subroutine may be defined both prior to or upon the call.



```
Label          ;subroutine name is 'Label'
    sequence of instructions...
    sequence of instructions...
    sequence of instructions...
return or retlw
```

As seen, concerning macros, the input and output arguments are of great importance. Concerning subroutines, it is not possible to define arguments within the subroutine itself. However, variables predefined in the main program may be used as subroutine arguments.

A logical sequence of events is as follows: defining variables, calling subroutine which uses them and at the end reading variables changed upon the execution of subroutine.

The program in the following example performs addition of two 2-byte variables ARG1 and ARG2 and moves result to the variable RES. When 2-byte variables are used, it is necessary to define higher and lower byte for each of them. The program itself is very simple. It first adds lower bytes of variables ARG1 and ARG2 and higher afterwards. If the sum of addition of two lower bytes is greater than 255 (maximal byte value) the remainder is added to the RESH variable.

```
; Program to add two 16-bit numbers
; Version: 1.0 Date: April 25, 2007 MCU:PIC16F887

      PROCESSOR 16f887 ; Defining processor
      #include "p16f887.inc" ; Microchip INC database
      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

      Cblock  0x20      ; Beginning of RAM
      ARG1H             ; Argument 1 higher byte
      ARG1L             ; Argument 1 lower byte
      ARG2H             ; Argument 2 higher byte
      ARG2L             ; Argument 2 lower byte
      RESH              ; Result higher byte
      RESL              ; Result lower byte
      endc              ; End of variables
      ORG     0x00      ; Reset vector
      goto    Start
```

```
Start                        ; Write values to variables
        movlw   0x01         ; ARG1=0x0104
        movwf   ARG1H
        movlw   0x04
        movwf   ARG1L
        movlw   0x07         ; ARG2=0x0705
        movwf   ARG2H
        movlw   0x05
        movwf   ARG2L
Main                         ; Main program
        call    Add16        ; Call subroutine Add16
Loop    goto    Loop         ; Remain here
Add16                        ; Subroutine to add two 16-bit numbers
        clrf    RESH         ; RESH=0
        movf    ARG1L,w      ; w=ARG1L
        addwf   ARG2L,w      ; w=w+ARG2L
        movwf   RESL         ; RESL=w
        btfsc   STATUS,C     ; Is the result greater than 255?
        incf    RESH,f       ; If greater, increment RESH by one

        movf    ARG1H,w      ; w=ARG1H
        addwf   ARG2H,w      ; w=w+ARG2
        addwf   RESH,f       ; RESH=w
        return               ; Return from subroutine
        end                  ; End of program
```

In Short

The main difference between macros and subroutines is that macro is after compiling replaced by its code (enables the programmer to type less). It may also have arguments while subroutine uses less memory, but does not have arguments.

MPLAB

MPLAB is a Windows program package which enables easy program writing as well as easy program development. It is best to describe it as development environment for a standard program language designed for PC programming. MPLAB technically simplifies some operations consisting of a lot of parameters, which, until the IDE environment* appeared, were executed from the command line. However, tastes are different and there are some programmers who prefer standard editors and command line compilers. Every program written in MPLAB is clear, but there are also help documentation- just in case.

INSTALLING MPLAB

MPLAB consists of several parts:

- The program which sorts data files of the same project into one group (**Project Manager**);
- program for text generating and processing (**Text Editor**); and
- simulator used to simulate the operation of a program loaded into the microcontroller.

Besides, there are also built in programmers such as PICStart Plus and ICD *(In Circuit Debugger)* that can be used to program software into PIC microcontroller device. Since not being the subject of this book, they are mentioned as options only.

In order to start MPLAB, your PC should contain:

- PC compatible computer belonging to class 486 or better;
- Any Windows operating system;

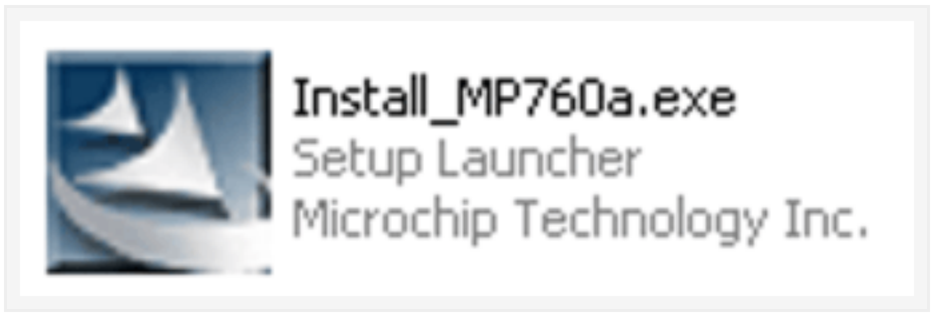
- VGA graphic card;
- 8MB memory (32MB recommended);
- 200MB available hard disc; and
- A mouse.

MPLAB installation comes first. Data files from MPLAB CD should be copied to a hard disc. The process of installation is similar to almost all other Windows program installations. First of all a welcome window appears, then options to select and at last installation itself. A message notifying that the program is successfully installed and ready for use appears. Are you ready?

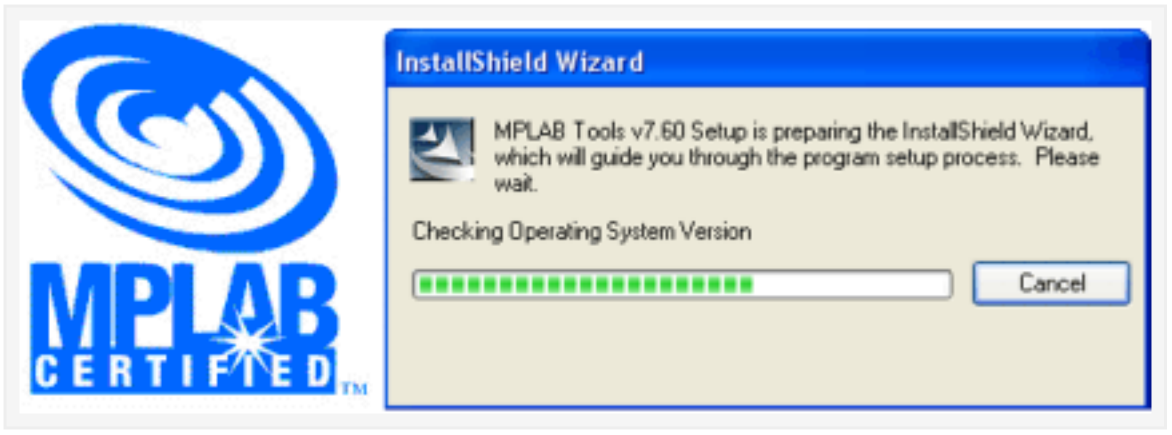
Steps to follow prior the installation:

1. Start Microsoft Windows;
2. Insert the CD into CD ROM;
3. Click START and select option RUN;
4. Click BROWSE and select CD ROM drive; and
5. Find folder MPLAB on CD ROM.

Everything is ready now to start installation. The following pictures describe the installation steps.



Click on this icon to start up the process…

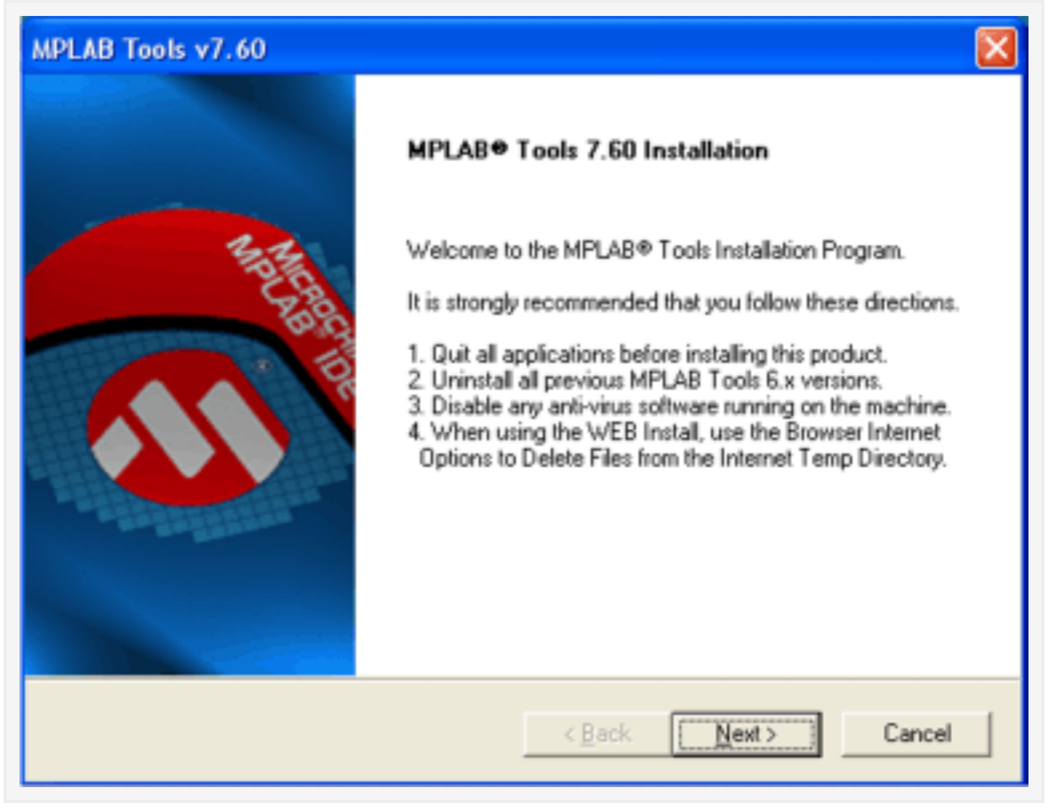


MPLAB Tools v7.60 Setup is preparing the InstallShield Wizard, which will guide you through the program setup process. Please wait.

Checking Operating System Version

Something is going on… The picture coming up indicates that the process of installation has just started! Next window contains the word “Welcome”. Need explanation?

Actually, the program reminds you to close all active programs in order to not interfere with the installation process. Next- of course!



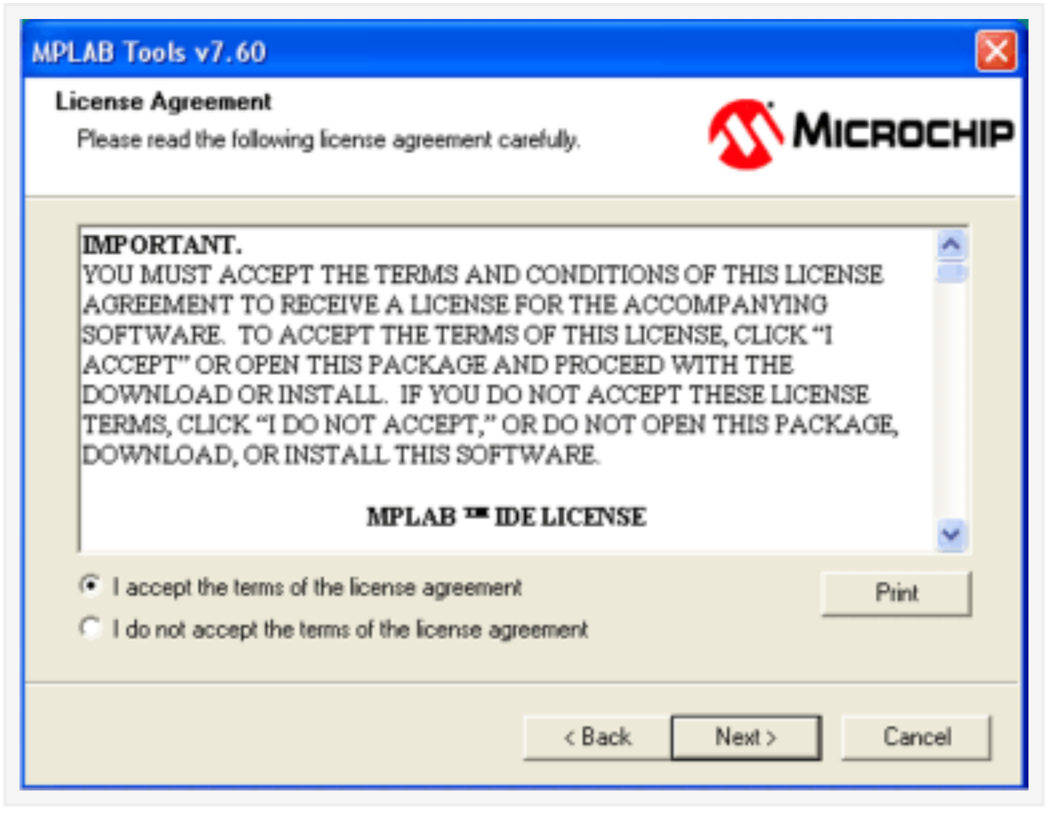
MPLAB® Tools 7.60 Installation

Welcome to the MPLAB® Tools Installation Program.

It is strongly recommended that you follow these directions.

1. Quit all applications before installing this product.
2. Uninstall all previous MPLAB Tools 6.x versions.
3. Disable any anti-virus software running on the machine.
4. When using the WEB Install, use the Browser Internet Options to Delete Files from the Internet Temp Directory.

Prior to continue, you have to accept the MPLAB software license conditions. Select the option "I accept" and click NEXT.



License Agreement

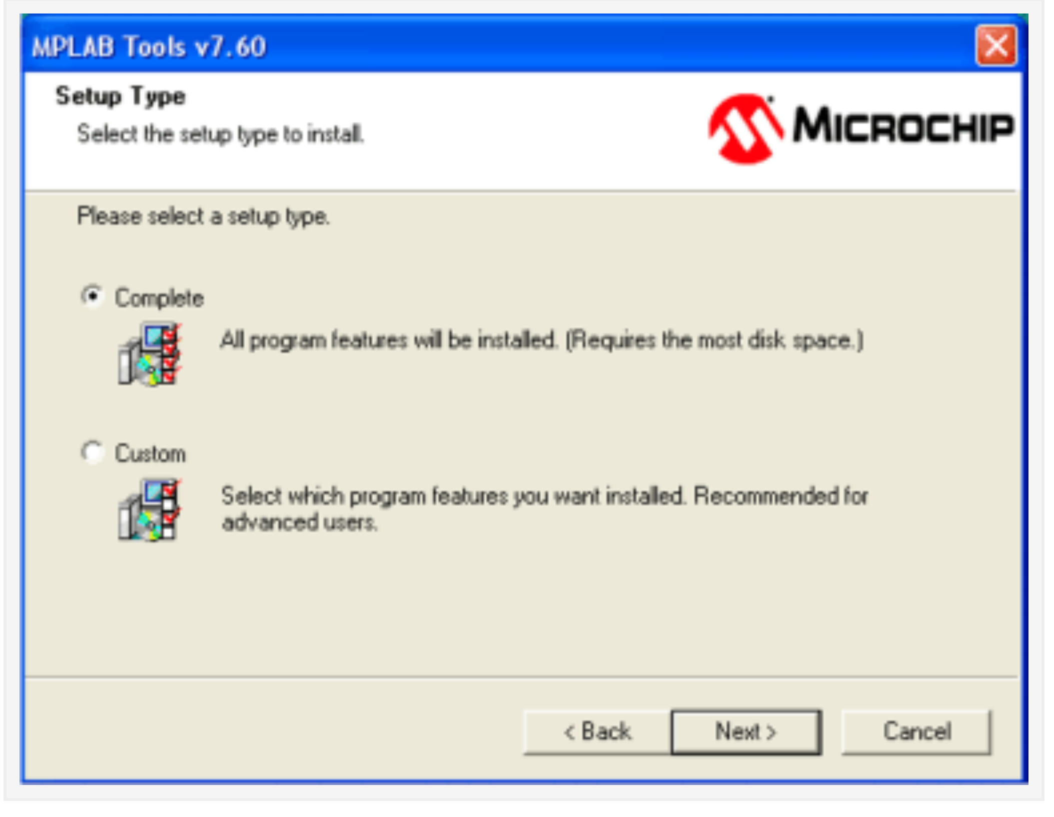
Please read the following license agreement carefully.

IMPORTANT.
YOU MUST ACCEPT THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT TO RECEIVE A LICENSE FOR THE ACCOMPANYING SOFTWARE. TO ACCEPT THE TERMS OF THIS LICENSE, CLICK "I ACCEPT" OR OPEN THIS PACKAGE AND PROCEED WITH THE DOWNLOAD OR INSTALL. IF YOU DO NOT ACCEPT THESE LICENSE TERMS, CLICK "I DO NOT ACCEPT," OR DO NOT OPEN THIS PACKAGE, DOWNLOAD, OR INSTALL THIS SOFTWARE.

MPLAB ™ IDE LICENSE

[✓ I accept the terms of the license agreement]
I do not accept the terms of the license agreement

Do you want to install the entire software? Yes. Next…



Setup Type

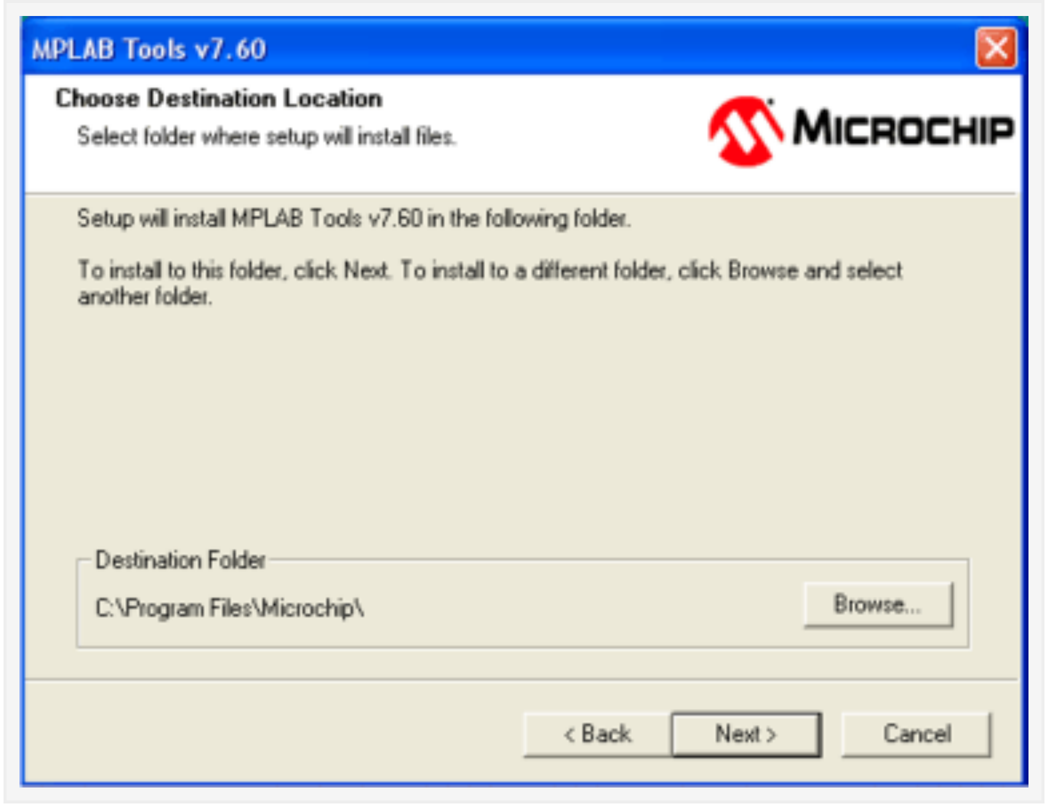
Select the setup type to install.

Please select a setup type.

[✓ Complete] All program features will be installed. (Requires the most disk space.)

Custom Select which program features you want installed. Recommended for advanced users.

Similar to other programs, MPLAB should be also installed into a folder. It may be any folder on any hard disc. If it is not necessary to make changes, select the specified address and click Next.


MPLAB Tools v7.60

Choose Destination Location

Select folder where setup will install files.

Setup will install MPLAB Tools v7.60 in the following folder.

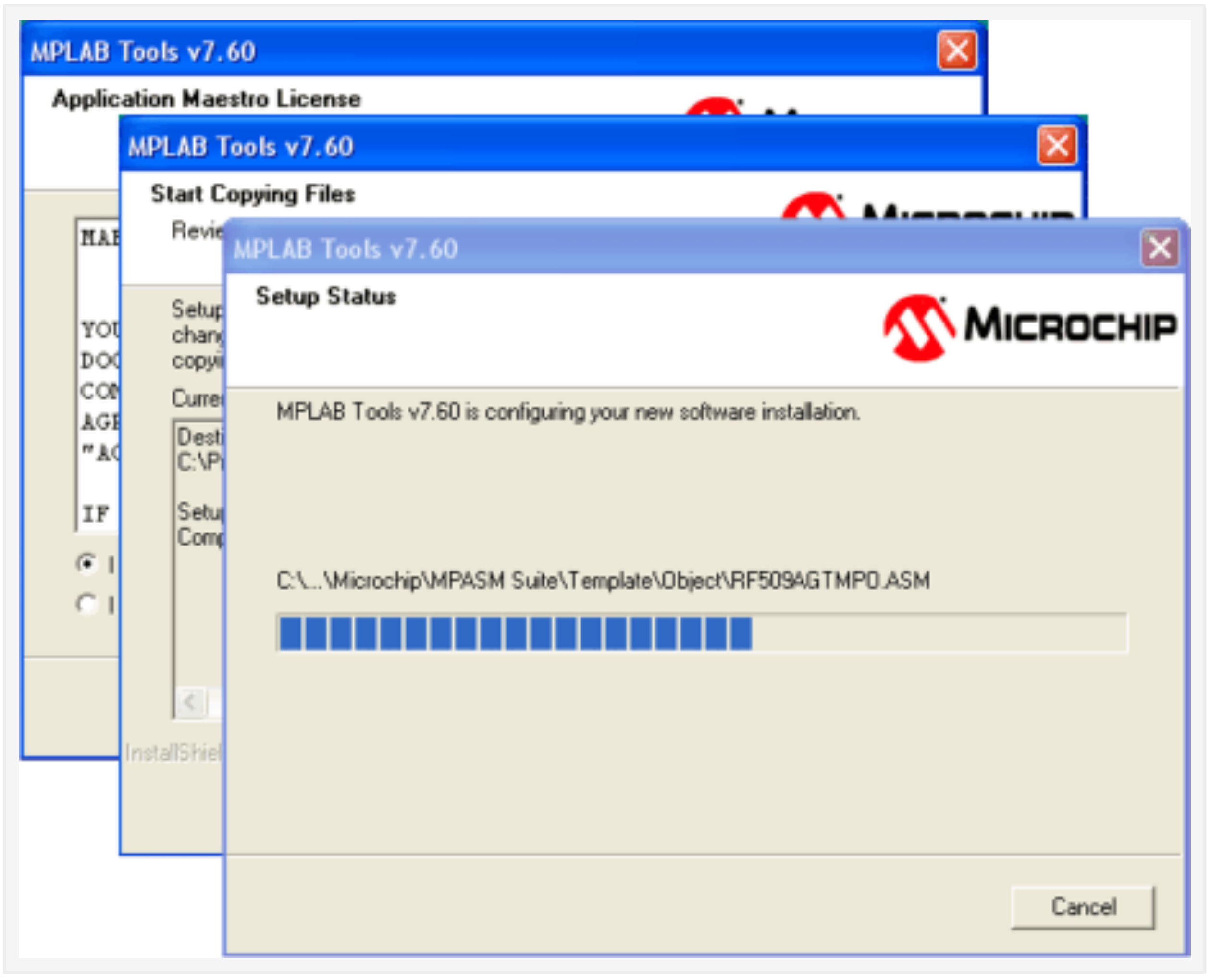
To install to this folder, click Next. To install to a different folder, click Browse and select another folder.

Destination Folder

C:\Program Files\Microchip\

Another license, another acceptance of options specified by the computer… Next, Next…

Be patient!


MPLAB Tools v7.60

Application Maestro License

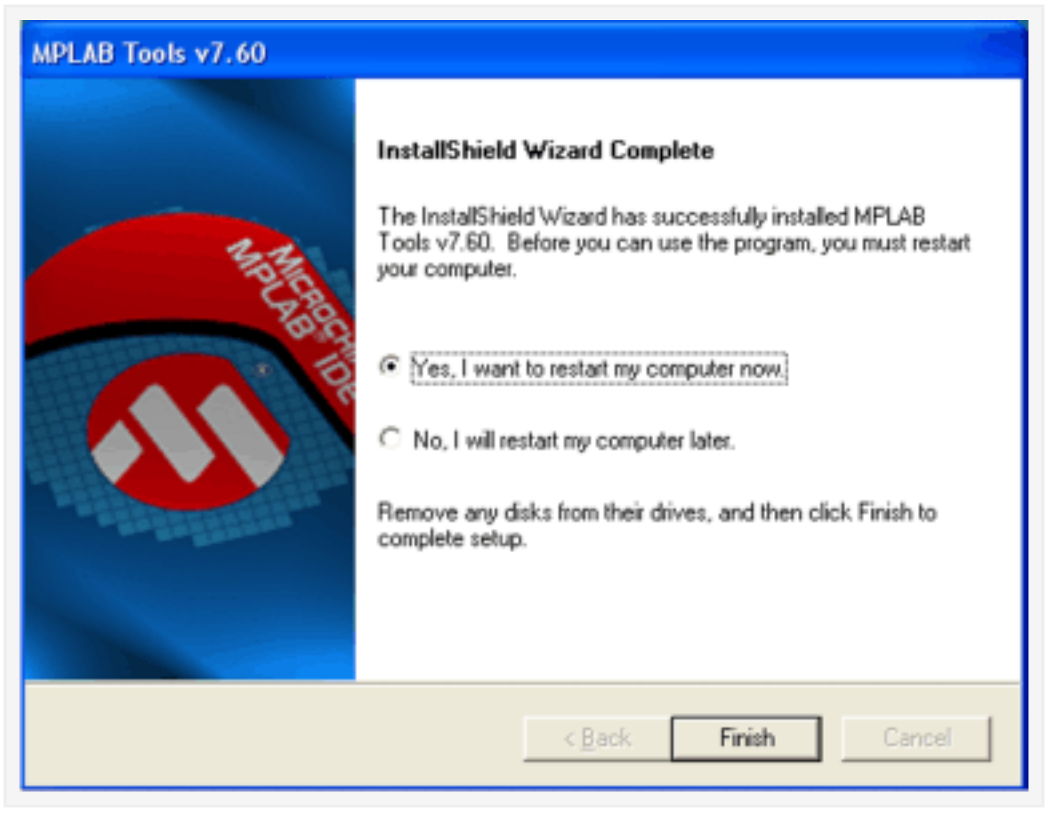
Start Copying Files

Setup Status

MPLAB Tools v7.60 is configuring your new software installation.

C:\...\Microchip\MPASM Suite\Template\Object\RF509AGTMP0.ASM

Finally! This is what you have been waiting for. Click Finish. The computer will be restarted along with the program saved on hard disc. Everything is OK!


MPLAB Tools v7.60

InstallShield Wizard Complete

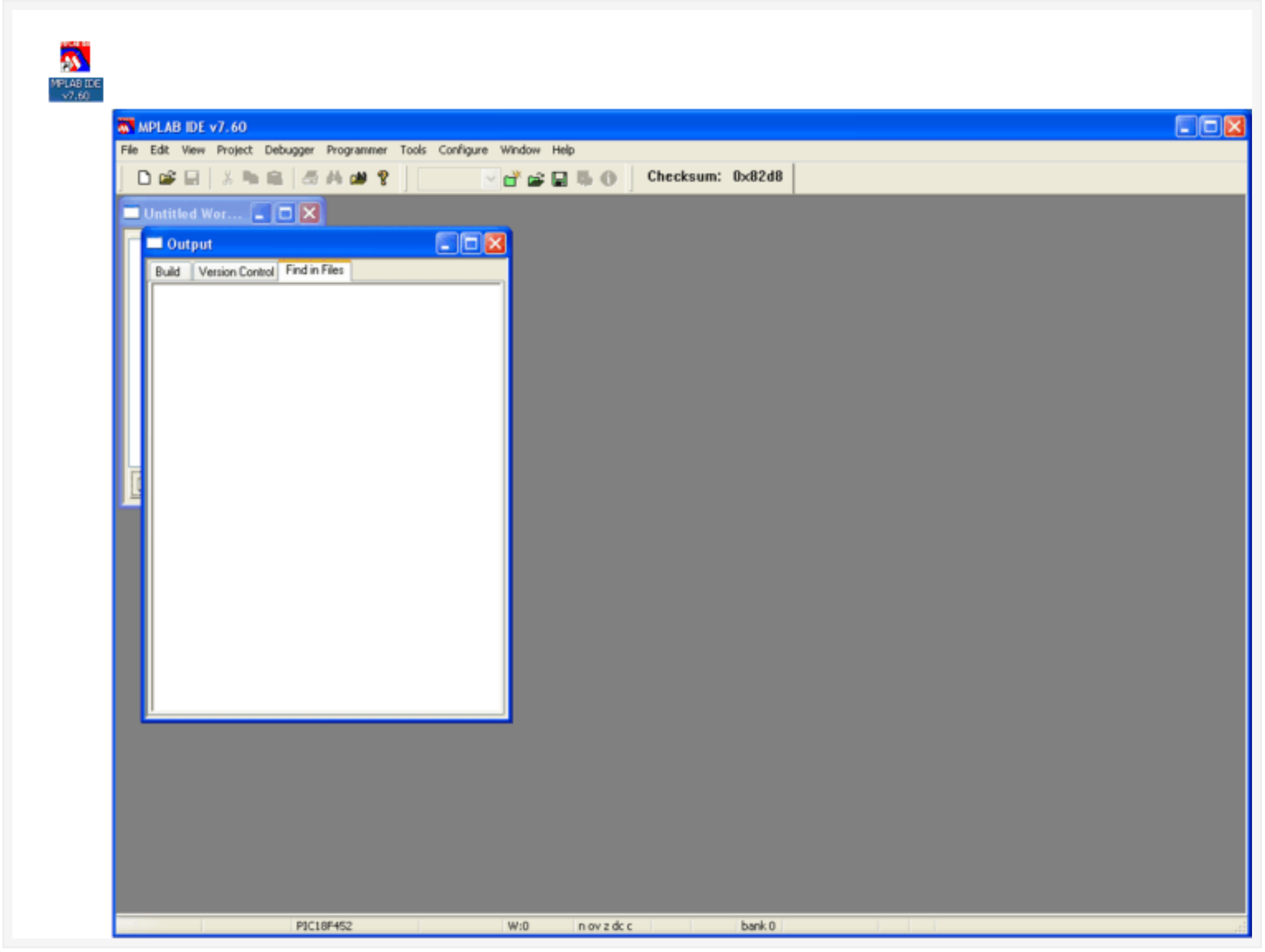
The InstallShield Wizard has successfully installed MPLAB Tools v7.60. Before you can use the program, you must restart your computer.

[✓ Yes, I want to restart my computer now.]

No, I will restart my computer later.

Remove any disks from their drives, and then click Finish to complete setup.

Click the MPLAB desktop icon in order to start the program and learn about it.


MPLAB IDE v7.60
MPLAB IDE v7.60
File Edit View Project Debugger Programmer Tools Configure Window Help
Checksum: 0x82d8
Untitled Wor...
Output
Build Version Control Find in Files
PIC18F452 W:0 n ov z dc c bank 0

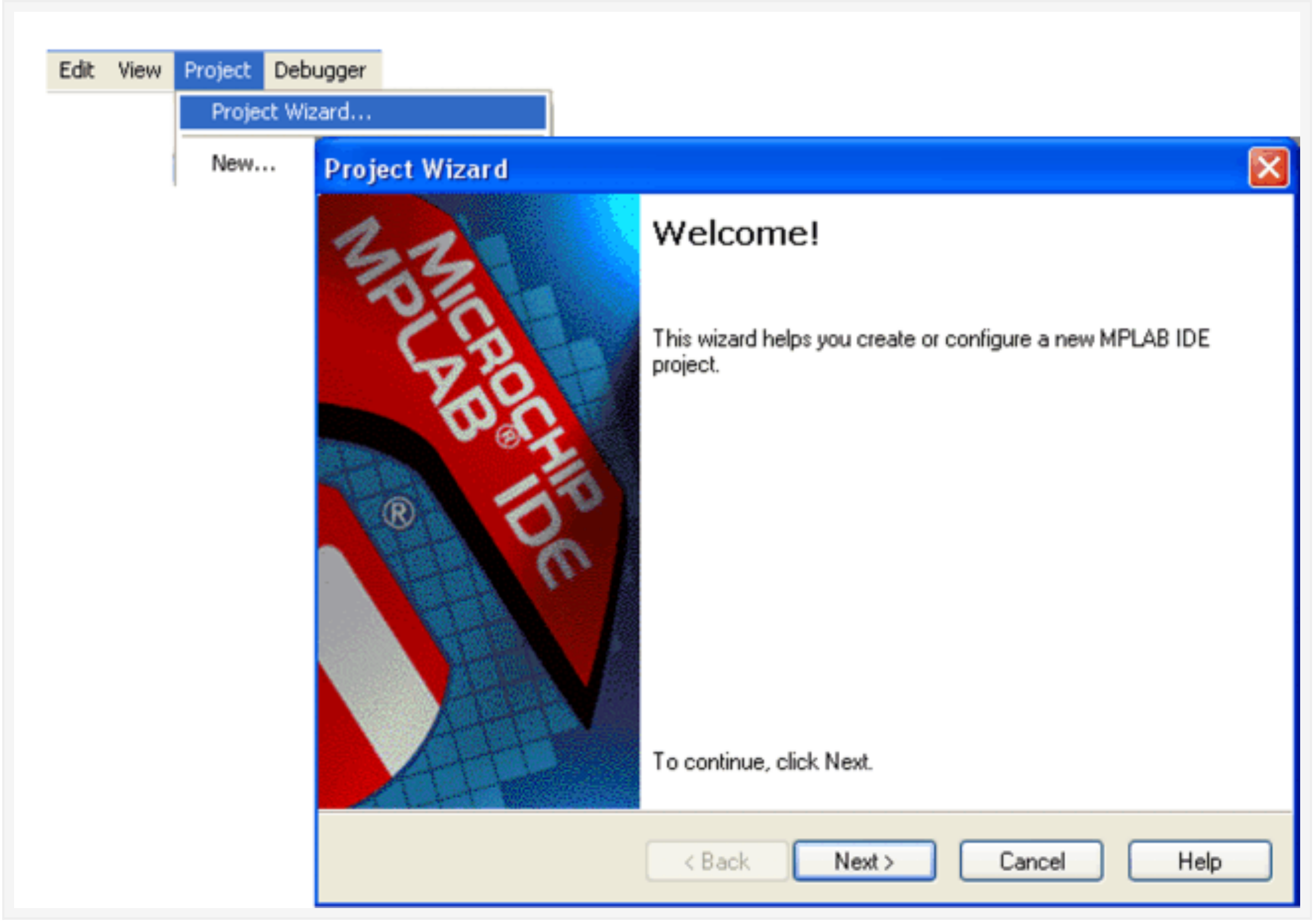
As seen, MPLAB is similar to most Windows programs. Apart from the working area, there are menus (contains options: File, Edit etc.), toolbars (contains different icons) and a status bar at the bottom of the window. Similar to Windows, there is a rule to have shortcuts for the most commonly used program options created in order to easily access them and speed up operation therefore. These shortcuts are actually icons below the menu bar. In other words, all options contained in the toolbar are contained in the menu too.

PROJECT-MAKING

Follow these steps to prepare program for loading into the microcontroller:

1. Make a project;
2. Write a program; and
3. Compile it.

In order to make a project, it is necessary to click the option “PROJECT” and then “PROJECT WIZARD”. A welcome window appears.



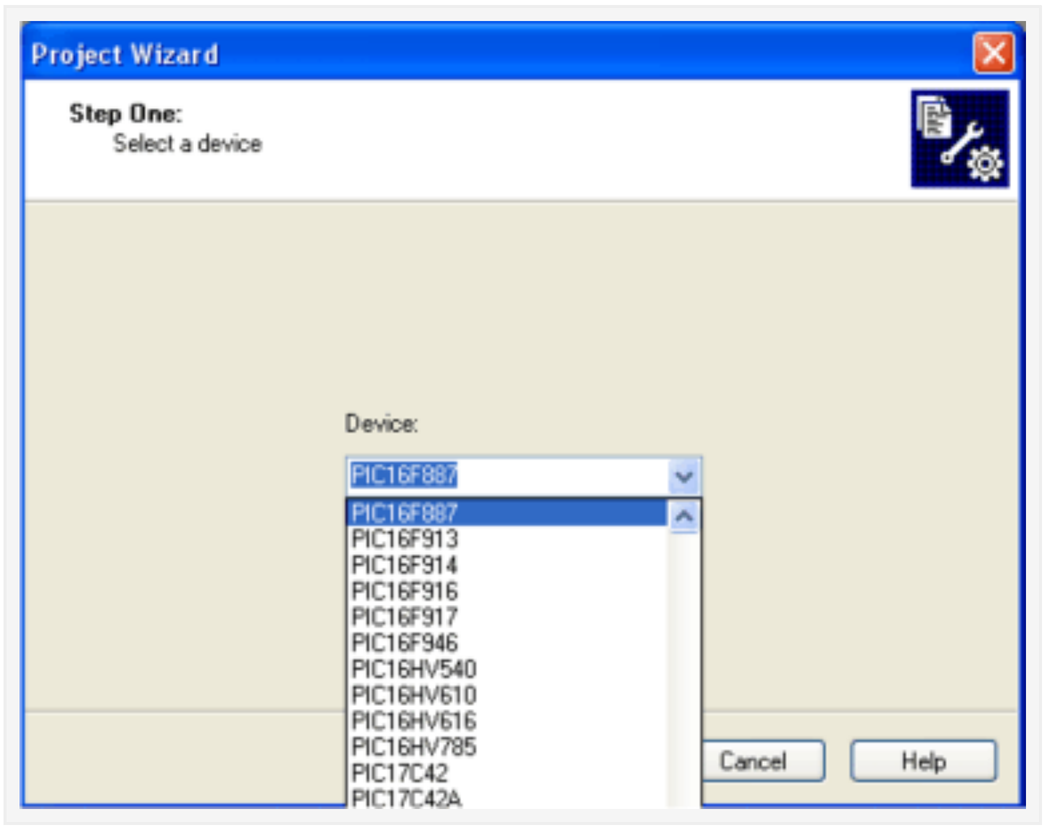
Welcome!

This wizard helps you create or configure a new MPLAB IDE project.

To continue, click Next.

Keep on project-making by clicking NEXT. Then select the microcontroller you will be using.

In our case, it is PIC16F887 microcontroller.

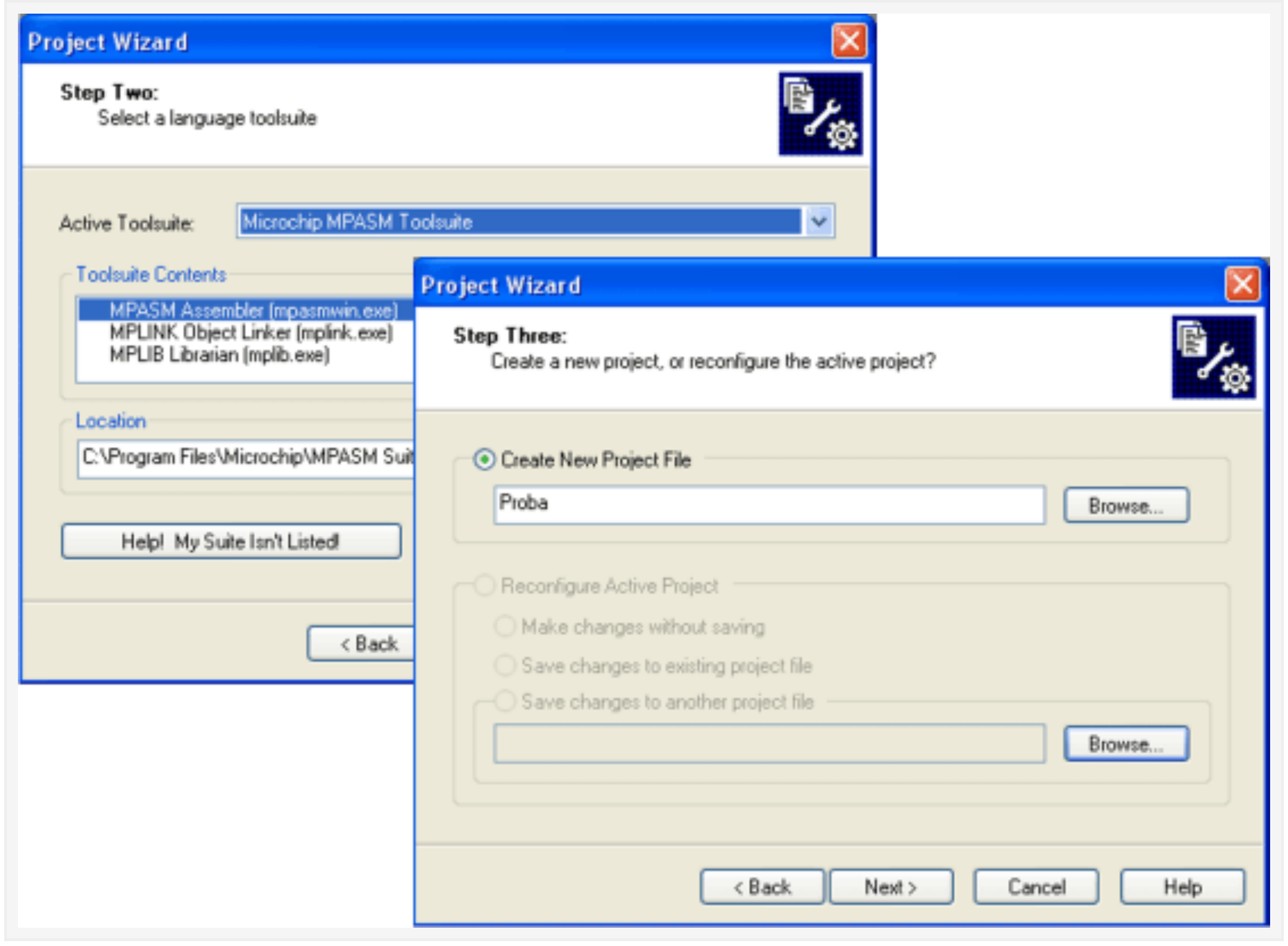


Step One:
Select a device

Device: PIC16F887

PIC16F887
PIC16F913
PIC16F914
PIC16F916
PIC16F917
PIC16F946
PIC16HV540
PIC16HV610
PIC16HV616
PIC16HV785
PIC17C42
PIC17C42A

At the end, the project is assigned a name which usually indicates the purpose and the content of the program being written. The project should be moved to the desired folder. It is best that the folder associates with PIC microcontrollers (See figure).


Project Wizard

Step Two:
Select a language toolsuite

Active Toolsuite: Microchip MPASM Toolsuite

Toolsuite Contents
MPASM Assembler (mpasmwin.exe)
MPLINK Object Linker (mplink.exe)
MPLIB Librarian (mplib.exe)

Location
C:\Program Files\Microchip\MPASM Sui

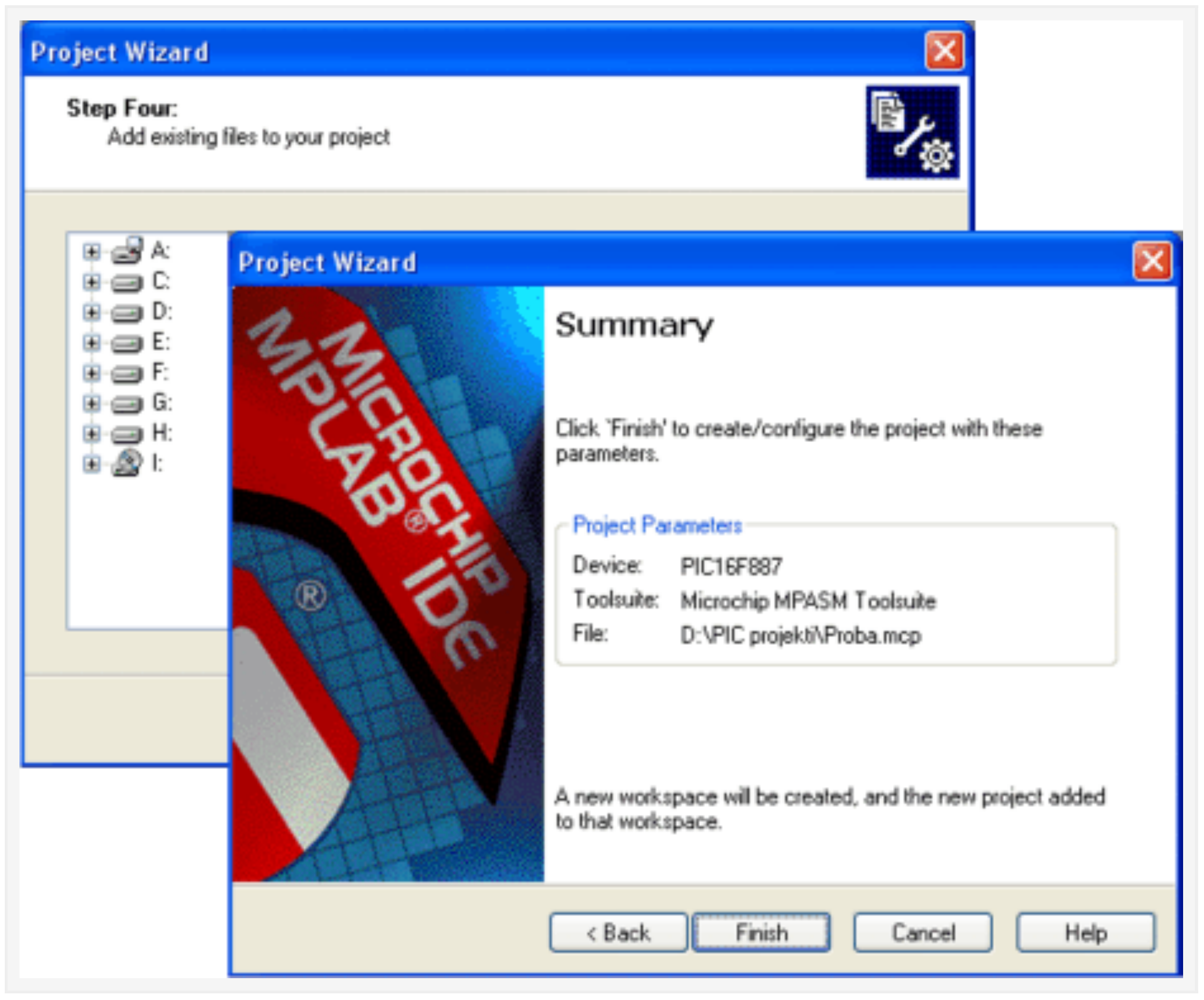
Project Wizard

Step Three:
Create a new project, or reconfigure the active project?

Create New Project File: Proba

Reconfigure Active Project
Make changes without saving
Save changes to existing project file
Save changes to another project file

Documents contained in the project do not always need to be written in MPLAB. Documents written by some other program may also be included in the project. In this example, there are no such documents. Just click Next.


Project Wizard

Step Four:
Add existing files to your project

Project Wizard

Summary

Click 'Finish' to create/configure the project with these parameters.

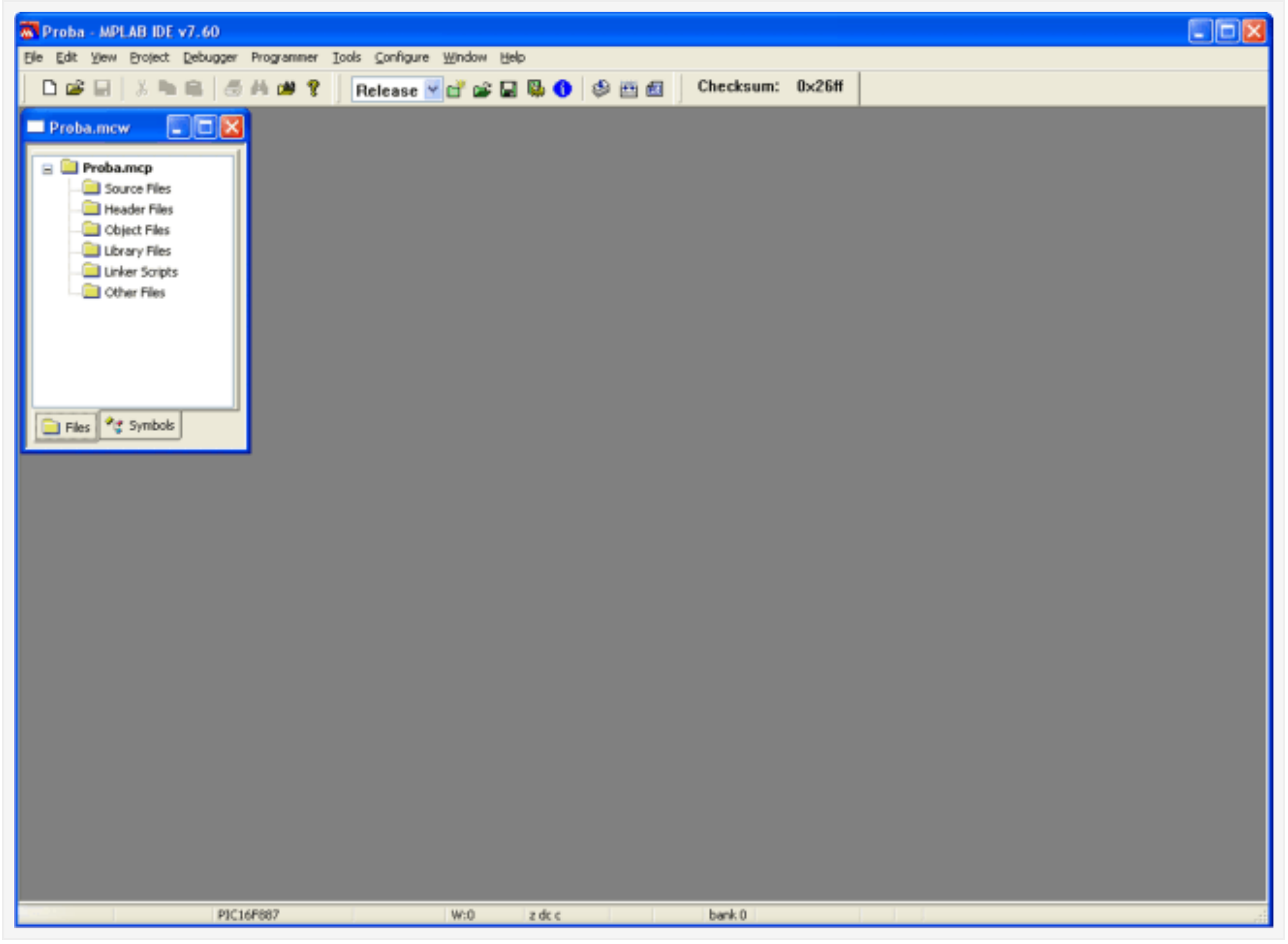
Project Parameters
Device: PIC16F887
Toolsuite: Microchip MPASM Toolsuite
File: D:\PIC projekti\Proba.mcp

A new workspace will be created, and the new project added to that workspace.

Click FINISH to complete the project. The window itself contains project parameters.

WRITING A NEW PROGRAM

When the project is created, a window shown in figure below appears.



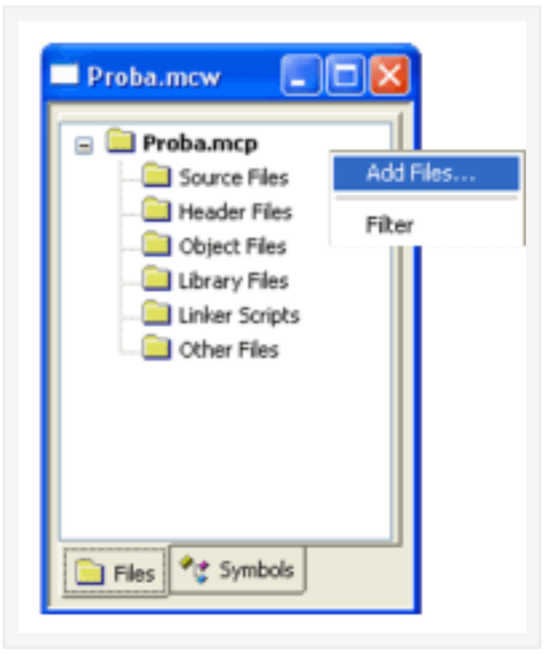
The next step is to write a program. Open a new document by clicking **File>New**. Text Editor in MPLAB environment appears.

Save the document in the folder **D:\PIC projects** by using the **File>Save As** command and name it “Blink.asm” indicating that this program is to be an example of port diode blinking. Obviously you can locate you files wherever you wish, in whichever hard drive you wish. Using a common directory to store all your different projects and subdirectories in makes good sense.

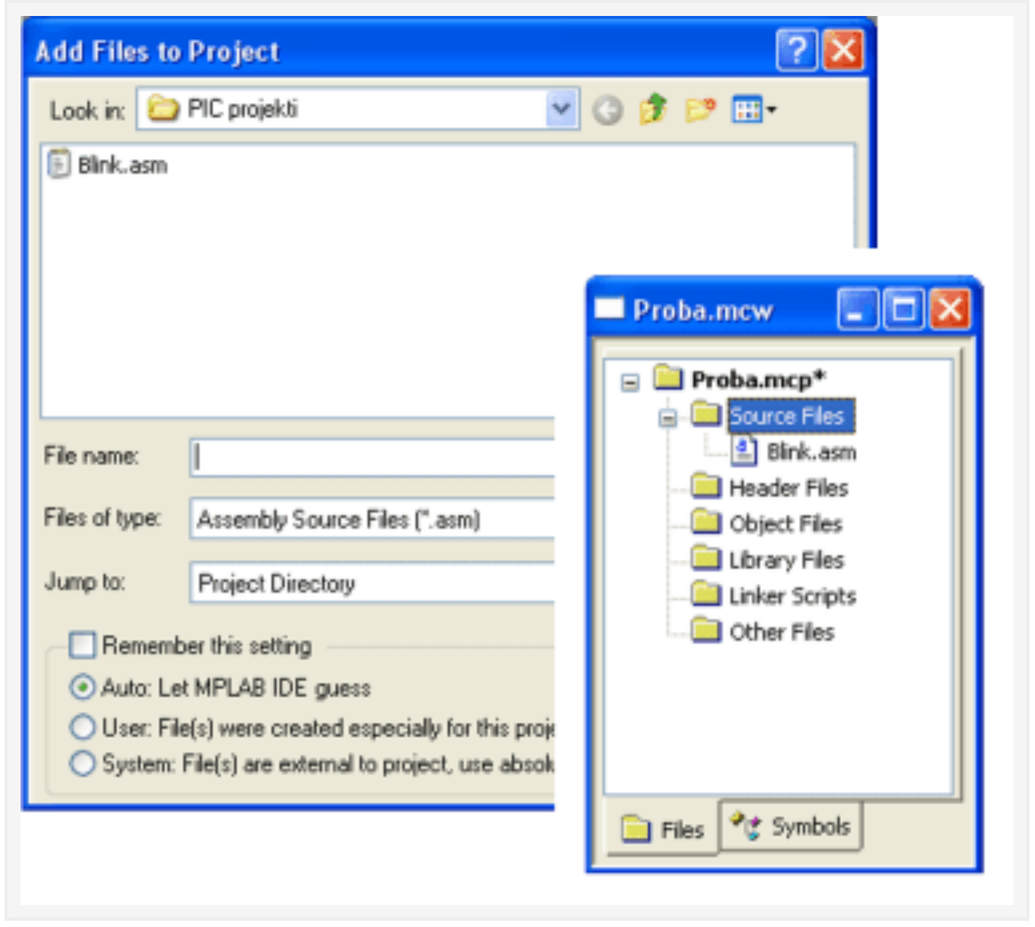
Example:

```
D:\Pic Projects
    LED Flash Project
        All associated files
    Event Count Project
        All associated files
    LED Scanning Project
        All associated files
```

After the “Blink.asm” is created and saved, it should be included in the project by right click on the “Source Files” option in the “Proba.mcw” window. After that, a small window with two options appears. Select the first one “Add Files”.



Click on that option opens another window containing the folder PIC along with the document Blink.asm. See figure below.


Add Files to Project

Look in: PIC projekti

Blink.asm

File name:

Files of type: Assembly Source Files (*.asm)

Jump to: Project Directory

Remember this setting

[✓ Auto: Let MPLAB IDE guess]

User: File(s) were created especially for this proj

System: File(s) are external to project, use absol

Click “Blink” to include the document Blink.asm into the project.

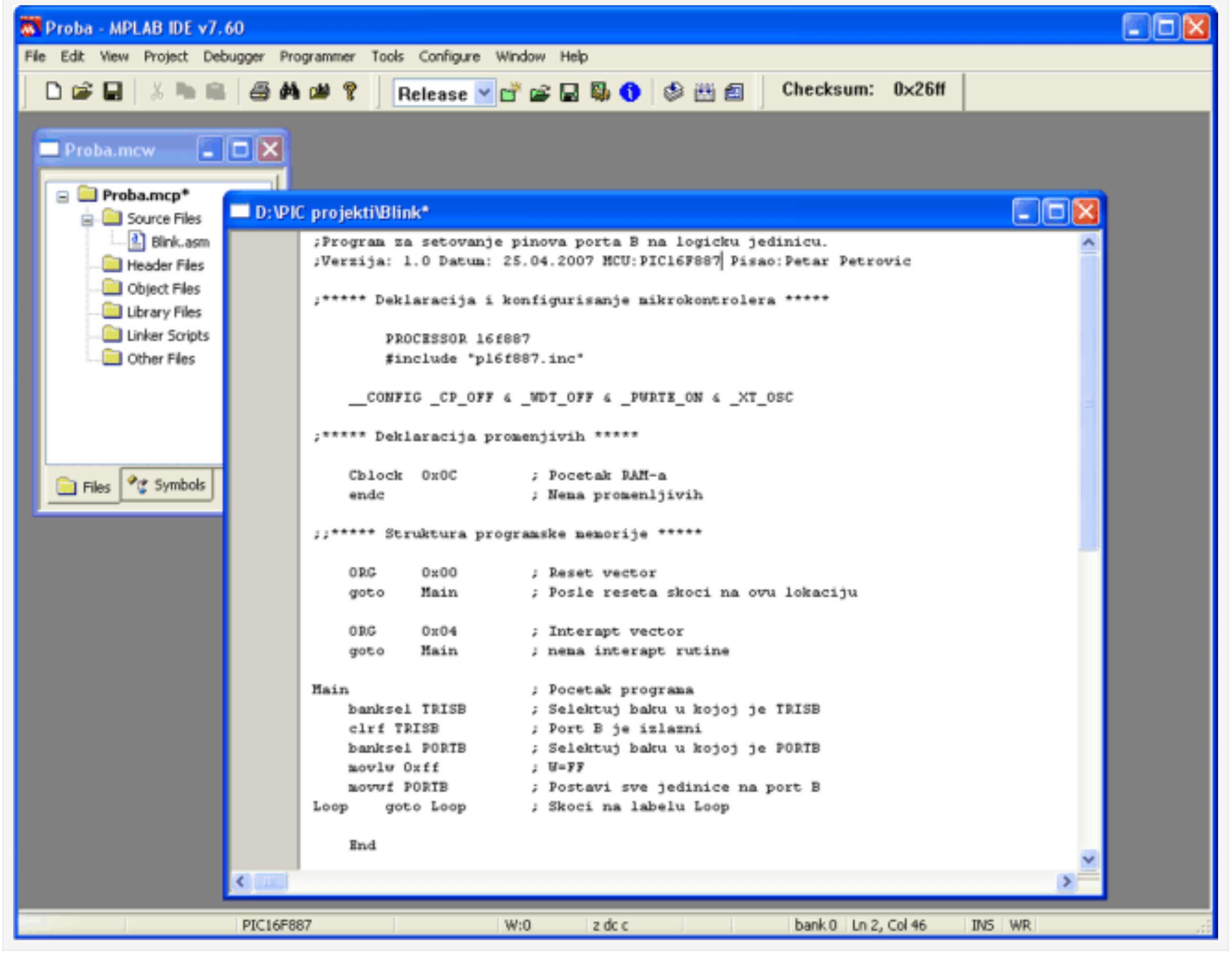
Program writing example

The program writing procedure cannot start until all previous operations have been performed. Program written below is a simple illustration of project-making.

```
;Program to set port B pins to logic one (1).
;Version: 1.0 Date: April 25,2007 MCU: PIC16F887 Programmer: John Smith

;***** Declaration and configuration of the microcontroller *****

        PROCESSOR 16f887
        #include "p16f887.inc"
        __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Variable declaration *****

        Cblock    0x20                   ; First free RAM location
        endc                             ; No variables

;;***** Program memory structure *****

        ORG       0x00                   ; Reset vector
        goto      Main                   ; After reset jump to this location

        ORG       0x04                   ; Interrupt vector
        goto      Main                   ; No interrupt routine
Main                                     ; Start the program
        banksel   TRISB                  ; Select bank containing TRISB
        clrf      TRISB                  ; Port B is configured as output
        banksel   PORTB                  ; Select bank containing PORTB
        movlw     0xff                   ; W=FF
        movwf     PORTB                  ; Move W to port B
Loop    goto      Loop                   ; Jump to label Loop

        End
```

The program should be written to the ‘Blink.asm’ window or copied from disc by means of options *copy/paste*. When copied, the program should be compiled into executable HEX format by using option PROJECT -> BUILD ALL. A new window appears. The last sentence is the most important because it tells us whether compiling has succeeded or not. Clearly, ’**BUILD SUCCEEDED**’ message means that no error occurred and compiling has been successfully done.

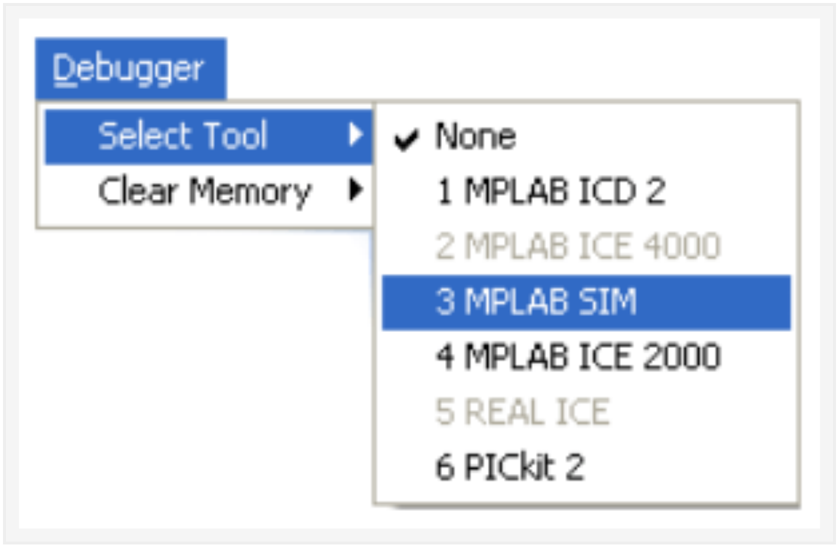
In case an error occurs, it is necessary to click twice on the message referring to it in the ‘Output’ window, which automatically switch you over to assembly program, directly to the line where the error has occurred.



```
;Program za setovanje pinova porta B na logicku jedinicu.
;Verzija: 1.0 Datum: 25.04.2007 MCU:PIC16F887 Pisao:Petar Petrovic

;***** Deklaracija i konfigurisanje mikrokontrolera *****

        PROCESSOR 16f887
        #include "p16f887.inc"

    __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Deklaracija promenjivih *****

    Cblock  0x0C          ; Pocetak RAM-a
    endc                  ; Nema promenljivih

;;***** Struktura programske memorije *****

    ORG     0x00          ; Reset vector
    goto    Main          ; Posle reseta skoci na ovu lokaciju

    ORG     0x04          ; Interapt vector
    goto    Main          ; nema interapt rutine

Main                      ; Pocetak programa
    banksel TRISB         ; Selektuj baku u kojoj je TRISB
    clrf TRISB            ; Port B je izlazni
    banksel PORTB         ; Selektuj baku u kojoj je PORTB
    movlw 0xff            ; W=FF
    movwf PORTB           ; Postavi sve jedinice na port B
Loop    goto Loop         ; Skoci na labelu Loop

    End
```

SIMULATOR

Asimulator is a part of MPLAB environment which provides better insight into the operation of the microcontroller. Generally speaking, a simulation is an attempt to model a real-life or hypothetical situation so that it can be studied to see how the system works. By means of the simulator, it is also possible to monitor current values of variables, registers and port pins states as well. To be honest, a simulator is not of the same importance for all programs. If a program is simpler (as in our example), the simulation is not of great importance because setting port B pins to logic one (1) is not complicated at all. However, in more complex programs containing timers, different conditions and requests (especially mathematical operations), the simulator may be of great use. As the name itself indicates, a simulation means to simulate the operation of microcontroller. Like the microcontroller, a simulator executes instructions one after another (line by line) and constantly updates the state of all registers. In this way, the user simply monitors program execution. At the end of program writing, the user should first test it in the simulator prior to executing it in a real environment. Unfortunately, this is one of many good things being overlooked by the programmer because of its character as such and the lack of high-quality simulators as well.



Simulator is activated by clicking on **DEBUGGER > SELECT TOOL > MPLAB SIM**, as shown in figure. As a result, several icons related to the simulator only appear. Their meanings are as follows:

Starts program execution at full speed. In this example, the simulator executes the program at full (normal) speed until it is halted by clicking the icon below.

Pauses program execution. Program can continue executing step by step or at full speed again.

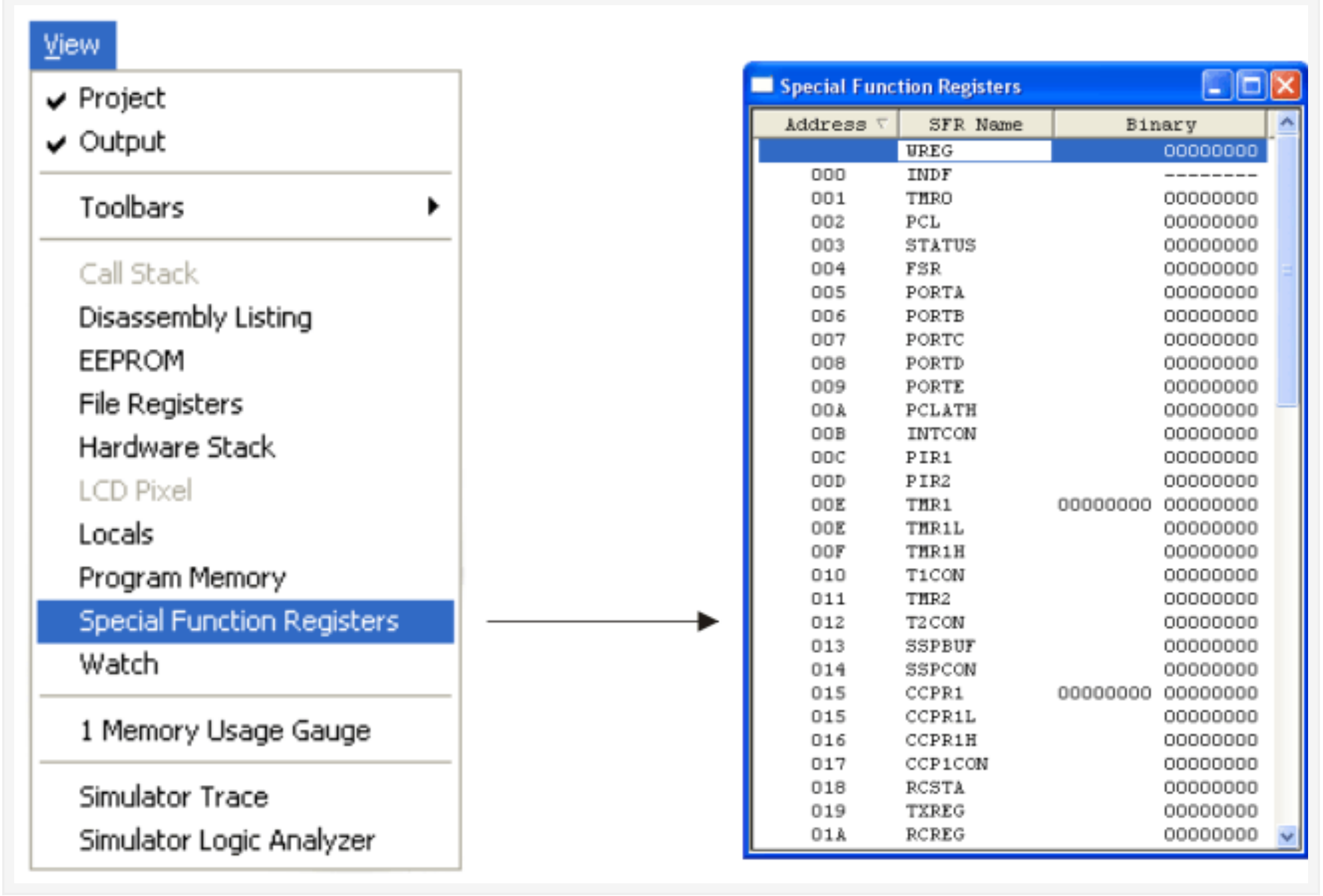
Starts program execution at optional speed. The speed of execution is set in dialog*Debugger/Settings/Animation/Realtime* Updates.

Starts step-by-step program execution. Instructions are executed one after another. Furthermore, clicking on this icon enables you to step into subroutines and macros.

This icon has the same function as the previous one except it has the ability to step into subroutines.

Resets microcontroller. By clicking this icon, the program counter is positioned at the beginning of the program and simulation can start.

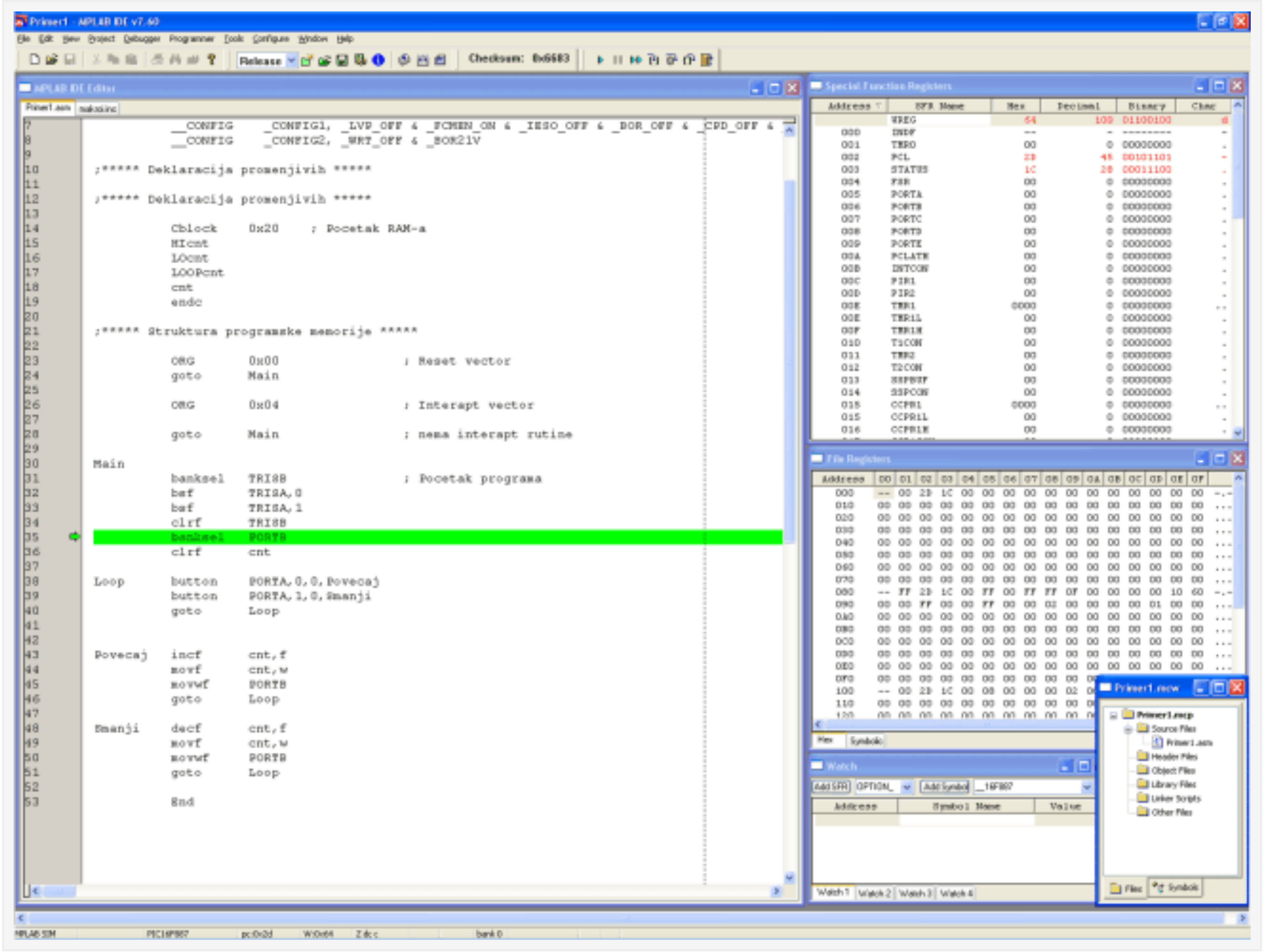
Similar to real environment, the first thing that should be done is to reset the microcontroller using the option DEBUGGER > RESET or by clicking reset icon. As the consequence of this, a green line is positioned at the beginning of the program and program counter PCL is cleared to zero. Refer to the window *Special Function Registers* shown below.



Address	SFR Name	Binary
	WREG	00000000
000	INDF	--------
001	TMR0	00000000
002	PCL	00000000
003	STATUS	00000000
004	FSR	00000000
005	PORTA	00000000
006	PORTB	00000000
007	PORTC	00000000
008	PORTD	00000000
009	PORTE	00000000
00A	PCLATH	00000000
00B	INTCON	00000000
00C	PIR1	00000000
00D	PIR2	00000000
00E	TMR1	00000000 00000000
00E	TMR1L	00000000
00F	TMR1H	00000000
010	T1CON	00000000
011	TMR2	00000000
012	T2CON	00000000
013	SSPBUF	00000000
014	SSPCON	00000000
015	CCPR1	00000000 00000000
015	CCPR1L	00000000
016	CCPR1H	00000000
017	CCP1CON	00000000
018	RCSTA	00000000
019	TXREG	00000000
01A	RCREG	00000000

Apart from SFRs, it is good to have an insight in File Registers. A window containing them appears by clicking the **VIEW->FILE REGISTERS**option.

If the program contains variables, it is good to monitor their values as well. Each variable is assigned a window (Watch Windows) by clicking**VIEW->WATCH** option.


Primer1 - MPLAB IDE v7.60
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
Checksum: 0x6683
MPLAB IDE Editor
Primer1.asm
Special Function Registers
File Registers
Watch
Primer1.mcw
MPLAB SIM
PIC16F887
pc:0x2d
W:0x64
bank 0

```
            __CONFIG    _CONFIG1, _LVP_OFF & _FCMEN_ON & _IESO_OFF & _BOR_OFF & _CPD_OFF &
            __CONFIG    _CONFIG2, _WRT_OFF & _BOR21V

;***** Deklaracija promenjivih *****

;***** Deklaracija promenjivih *****

            Cblock      0x20    ; Pocetak RAM-a
            HIcnt
            LOcnt
            LOOPcnt
            cnt
            endc

;***** Struktura programske memorije *****

            ORG         0x00                ; Reset vector
            goto        Main

            ORG         0x04                ; Interapt vector

            goto        Main                ; nema interapt rutine

Main
            banksel     TRISB               ; Pocetak programa
            bsf         TRISA,0
            bsf         TRISA,1
            clrf        TRISB
            banksel     PORTB
            clrf        cnt

Loop        button      PORTA,0,0,Povecaj
            button      PORTA,1,0,Smanji
            goto        Loop


Povecaj     incf        cnt,f
            movf        cnt,w
            movwf       PORTB
            goto        Loop

Smanji      decf        cnt,f
            movf        cnt,w
            movwf       PORTB
            goto        Loop

            End
```

If the program contains variables, it is good to monitor their values as well. Each variable is assigned a window (*Watch Windows*) by clicking**VIEW->WATCH** option.

After all variables and registers of interest become available on the simulator working area, the process of simulation can start. The next instruction may be either **Step into** or **Step over** depending on whether you want to step into subroutine or not. The same instructions may be set by using keyboard- push-buttons <F7> or <F8>

(generally, all important instructions have the corresponding pushbuttons on the keyboard).