Pic Micro Pascal V1.6

User Manual

Doc date: 2011-05-31, revision: A.

Author: Philippe Paternotte (PPA)

Summary

1 Overview

Pic Micro Pascal (aka PMP) is a language tool that is targeted for small and medium Microchip PIC family of micro-controllers (PIC10, PIC12, PIC16 and PIC18).

PMP is not a commercial compiler that does everything, but is intended to assist developers with the generation of small to medium (both in scope and in code size) applications for the PIC.

Major guidelines of development are:

- Try to be as close as possible to the standard TP / Delphi syntax.
- Do not use special built-in functions and procedures to interface hardware registers; these registers are accessed directly. So the Pascal source code is not portable to other manufacturer processors, but the target is PIC, and will stay PIC!
- Try to make the finest possible code optimizations (well, this is and endless work).

In the present implementation, PMP supports multiple files compilation, by include directives and by a per unit concept, as in TP or Delphi.

PMP supports simple records, bit booleans, signed and unsigned types, long integers, one dimension arrays, strings and floating point variables or operations (floating point is only supported for PIC16, PIC 16 enhanced and PIC18 since the FP code consumes too much).

The philosophy of PMP is to globally limit the number of special keywords or built-in functions and procedures for manipulating special PIC hardware. Ports and other special SFRs are accessed directly (as PORTC or TRISC for example).

PMP does not include an assembler or linker; it is designed to work with MPAsm™ / MPLink™ from MPLAB® suite, and uses their .inc and .lkr files for standard registers definitions and processor / memory mapping.

PMP was initially inspired by the MPTINY implementation by Thomas J. LeMense and "Let's Build A Compiler" series written by Jack Crenshaw in 1988 through 1992. A web search for "Crenshaw" and "compiler" should help you find where this excellent reference may be found.

The PMP scanner was made with the help of TPLex, a Turbo Pascal implementation of LEX. TPLex was made possible by Albert Graef and Berend de Boer. Information on this program and a companion port of Yacc may be found on the TPLY homepage: http://www.musikwissenschaft.uni-mainz.de/~ag/tply. There is also a Delphi port.

The PMP floating point engine is derived from the PicFloat libraries from Mike Gore.

PMP has an IDE, build around the SynEdit package (http://synedit.sourceforge.net).

PMP is still under construction. Source code is not available yet.

PMP main core (not including std libraries, IDE and low level code generators) is about 100K lines of code.

- **Note that there's an index at the end of this document!**

1.1 Legal stuff

1.2 BSD License

2 Operation

2.1 Operating system

PMP is a 32 bits Windows program, so it cannot run in pure MS/DOS or old 16 bit Windows (even for the command line version).

It has been tested on Windows 2000 and Windows XP SP3.

PMP compiler has no memory limitations: all strings and tables are dynamic, and so PMP is only limited by the OS limits.

2.2 MPLAB® suite

PMP is intended to be used with the Microchip Technology MPLAB® suite installed.

It cannot compile anything if it is not able to find processors include files and linker mapping files, along with MPASM™ and MPLINK™ programs.

2.3 Console version

Console version is no more distributed since previous versions had not been updated to integrate new PMP rewriting (a release is scheduled).

Nevertheless, hereafter this document refers to it and contains some information about its interfaces (managing a future console version that may be released again).

2.3.1 Command line usage

The PMP console version uses command line arguments. It accepts the following syntax:

PMP <source file> <output file> <options>

Arguments order does not matter, but <source file> is always the first given file name.

The <source file> may be any valid filename. If the file extension is omitted, PMP assumes that the source file extension is ".pas".

The <output file> is optional; if it is omitted, PMP generates a file with the same name as the source file, but with an ".asm" extension, in the same directory.

The options supported at this time (not case sensitive) are as follows:

/c	generate commented code (defaults to off)
/s	generate symbol information (defaults to off)
/v	generate verbose source lines (defaults to off)
/w	all warnings off (defaults to on)
/wn	warning n off
/i<path>	specify a global path for include files (default is the source file directory)
/p<processor>	defines processor part number; this name must match MPAsm™ syntax. It is mandatory to compile a unit.
/f<frequency>	defines processor operating frequency. It is mandatory to compile a unit.
/d<Identifier>	defines a conditional compilation identifier.
/?	display usage information
/h	same as /?

Note that include files are always searched in the source file directory before to search in the global path if any.

For MPAsm™ file search, PMP finds the path in the registry. Path for processor include file is not emitted in .asm file for MPAsm™ include directive since it has its own path search mechanism.

2.3.2 Message output from PMP compiler

The PMP console version outputs messages to the screen and to an .erp file witch syntax is similar to the MPAsm™ error file syntax.

2.3.3 Exit code

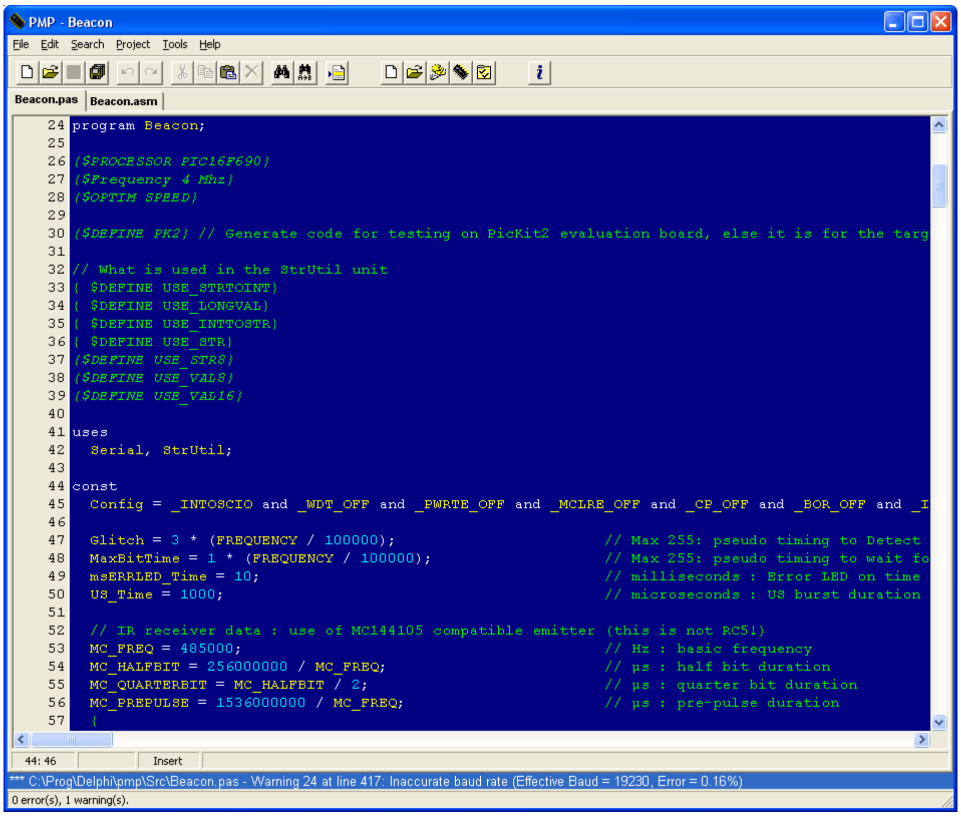
In the current implementation Exit Code is 0 if no warnings or errors, 1 if warnings but no errors, 2 if there are errors.

2.4 *Windows version*

The PMP Windows version starts directly with the PMP IDE.

This version may accept one command line argument that is a project file name (.pmp) to open.

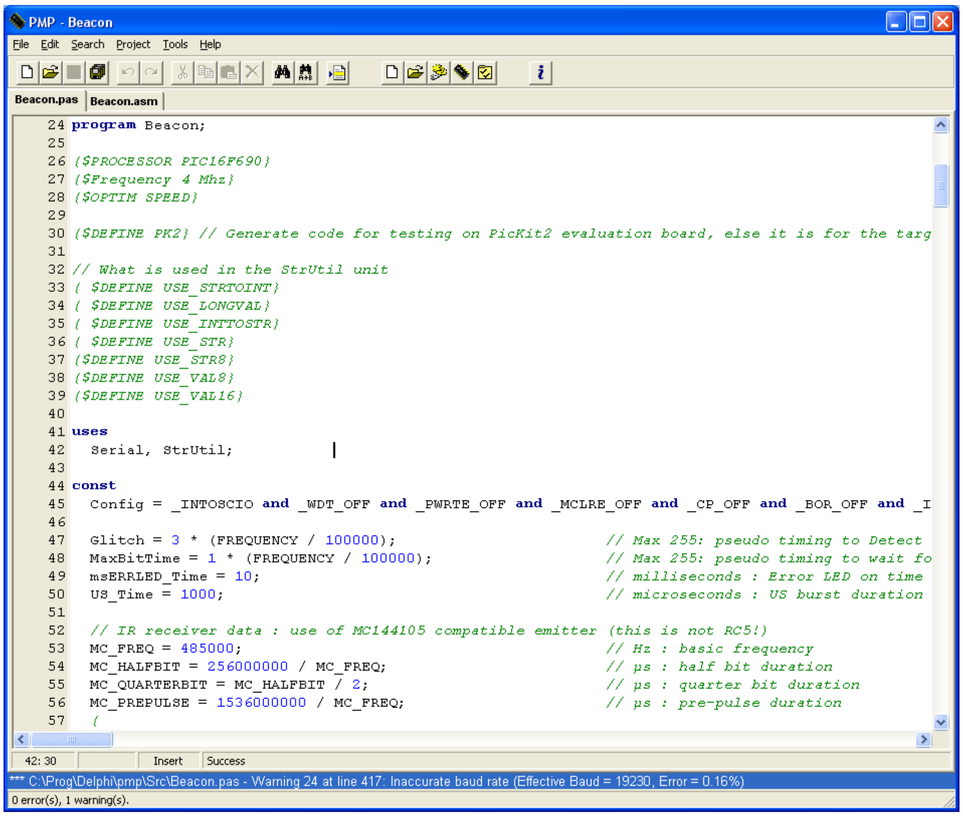
Here is a typical screen shot of PMP IDE (Pascal file) with “old fashioned” colors:



```
program Beacon;

{$PROCESSOR PIC16F690}
{$Frequency 4 Mhz}
{$OPTIM SPEED}

{$DEFINE PK2} // Generate code for testing on PicKit2 evaluation board, else it is for the targ

// What is used in the StrUtil unit
{ $DEFINE USE_STRTOINT}
{ $DEFINE USE_LONGVAL}
{ $DEFINE USE_INTTOSTR}
{ $DEFINE USE_STR}
{$DEFINE USE_STR8}
{$DEFINE USE_VAL8}
{$DEFINE USE_VAL16}

uses
  Serial, StrUtil;

const
  Config = _INTOSCIO and _WDT_OFF and _PWRTE_OFF and _MCLRE_OFF and _CP_OFF and _BOR_OFF and _I

  Glitch = 3 * (FREQUENCY / 100000);                        // Max 255: pseudo timing to Detect
  MaxBitTime = 1 * (FREQUENCY / 100000);                    // Max 255: pseudo timing to wait fo
  msERRLED_Time = 10;                                       // milliseconds : Error LED on time
  US_Time = 1000;                                           // microseconds : US burst duration

  // IR receiver data : use of MC144105 compatible emitter (this is not RC5!)
  MC_FREQ = 485000;                                         // Hz : basic frequency
  MC_HALFBIT = 256000000 / MC_FREQ;                         // µs : half bit duration
  MC_QUARTERBIT = MC_HALFBIT / 2;                           // µs : quarter bit duration
  MC_PREPULSE = 1536000000 / MC_FREQ;                       // µs : pre-pulse duration
  {
```

```
44: 46    Insert
*** C:\Prog\Delphi\pmp\Src\Beacon.pas - Warning 24 at line 417: Inaccurate baud rate (Effective Baud = 19230, Error = 0.16%)
0 error(s), 1 warning(s).
```

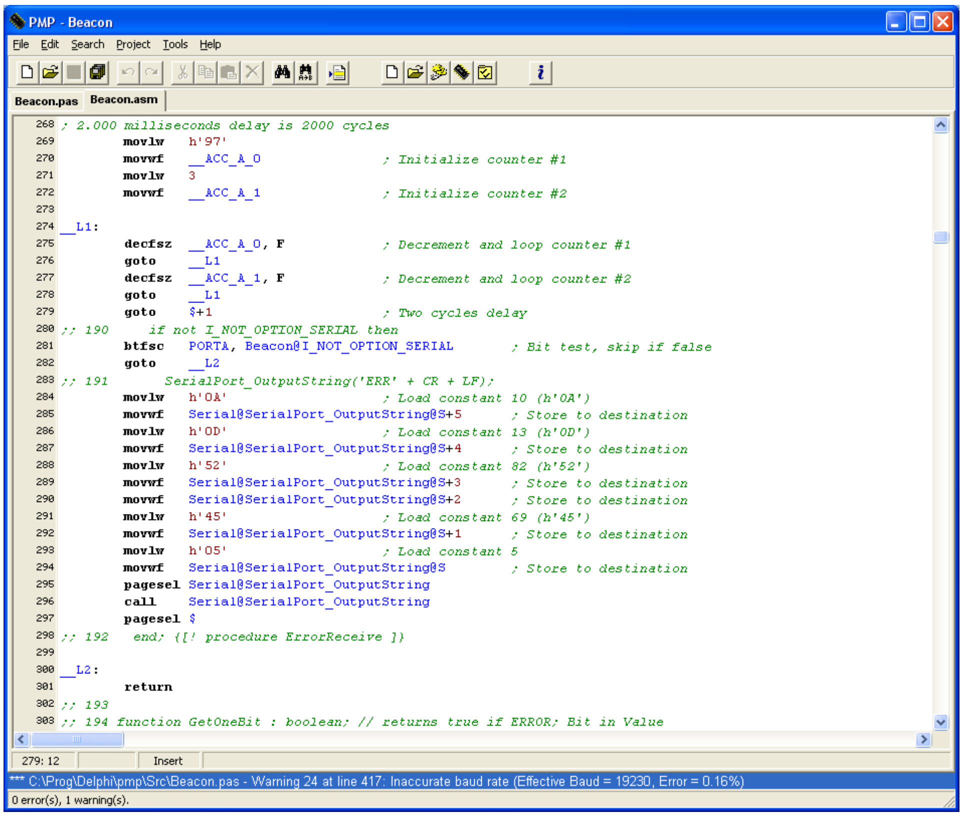
And the same one with Delphi 2006 colors:



```
program Beacon;

{$PROCESSOR PIC16F690}
{$Frequency 4 Mhz}
{$OPTIM SPEED}

{$DEFINE PK2} // Generate code for testing on PicKit2 evaluation board, else it is for the targ

// What is used in the StrUtil unit
{ $DEFINE USE_STRTOINT}
{ $DEFINE USE_LONGVAL}
{ $DEFINE USE_INTTOSTR}
{ $DEFINE USE_STR}
{$DEFINE USE_STR8}
{$DEFINE USE_VAL8}
{$DEFINE USE_VAL16}

uses
  Serial, StrUtil;

const
  Config = _INTOSCIO and _WDT_OFF and _PWRTE_OFF and _MCLRE_OFF and _CP_OFF and _BOR_OFF and _I

  Glitch = 3 * (FREQUENCY / 100000);                        // Max 255: pseudo timing to Detect
  MaxBitTime = 1 * (FREQUENCY / 100000);                    // Max 255: pseudo timing to wait fo
  msERRLED_Time = 10;                                       // milliseconds : Error LED on time
  US_Time = 1000;                                           // microseconds : US burst duration

  // IR receiver data : use of MC144105 compatible emitter (this is not RC5!)
  MC_FREQ = 485000;                                         // Hz : basic frequency
  MC_HALFBIT = 256000000 / MC_FREQ;                         // µs : half bit duration
  MC_QUARTERBIT = MC_HALFBIT / 2;                           // µs : quarter bit duration
  MC_PREPULSE = 1536000000 / MC_FREQ;                       // µs : pre-pulse duration
  {
```

```
42: 30    Insert    Success
*** C:\Prog\Delphi\pmp\Src\Beacon.pas - Warning 24 at line 417: Inaccurate baud rate (Effective Baud = 19230, Error = 0.16%)
0 error(s), 1 warning(s).
```

Here is a typical screen shot of PMP IDE (assembler file):



```
; 2.000 milliseconds delay is 2000 cycles
        movlw   h'97'
        movwf   __ACC_A_0               ; Initialize counter #1
        movlw   3
        movwf   __ACC_A_1               ; Initialize counter #2

__L1:
        decfsz  __ACC_A_0, F            ; Decrement and loop counter #1
        goto    __L1
        decfsz  __ACC_A_1, F            ; Decrement and loop counter #2
        goto    __L1
        goto    $+1                     ; Two cycles delay
;; 190     if not I_NOT_OPTION_SERIAL then
        btfsc   PORTA, Beacon@I_NOT_OPTION_SERIAL       ; Bit test, skip if false
        goto    __L2
;; 191       SerialPort_OutputString('ERR' + CR + LF);
        movlw   h'0A'                   ; Load constant 10 (h'0A')
        movwf   Serial@SerialPort_OutputString@S+5      ; Store to destination
        movlw   h'0D'                   ; Load constant 13 (h'0D')
        movwf   Serial@SerialPort_OutputString@S+4      ; Store to destination
        movlw   h'52'                   ; Load constant 82 (h'52')
        movwf   Serial@SerialPort_OutputString@S+3      ; Store to destination
        movwf   Serial@SerialPort_OutputString@S+2      ; Store to destination
        movlw   h'45'                   ; Load constant 69 (h'45')
        movwf   Serial@SerialPort_OutputString@S+1      ; Store to destination
        movlw   h'05'                   ; Load constant 5
        movwf   Serial@SerialPort_OutputString@S        ; Store to destination
        pagesel Serial@SerialPort_OutputString
        call    Serial@SerialPort_OutputString
        pagesel $
;; 192   end; {[! procedure ErrorReceive ]}

__L2:
        return
;; 193
;; 194 function GetOneBit : boolean; // returns true if ERROR; Bit in Value
```

279: 12   Insert

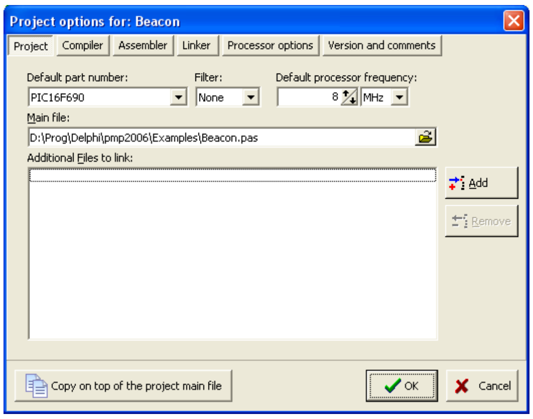
*** C:\Prog\Delphi\pmp\Src\Beacon.pas - Warning 24 at line 417: Inaccurate baud rate (Effective Baud = 19230, Error = 0.16%)

0 error(s), 1 warning(s).

2.4.1 PMP Project

A project is stored in a .pmp file that is internally an .ini file. This file stores all information about the project.

2.4.1.1 Project general options

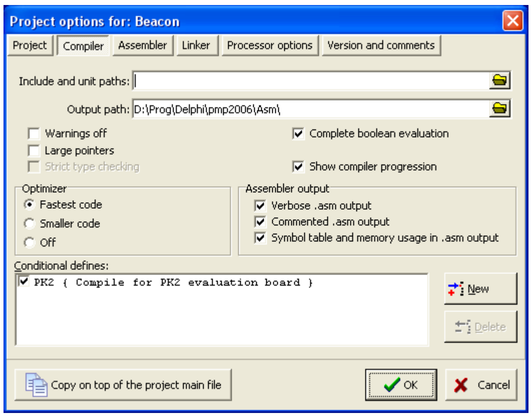


Default part number: PIC16F690
Filter: None
Default processor frequency: 8 MHz
Main file: D:\Prog\Delphi\pmp2006\Examples\Beacon.pas
Additional Files to link:

The default part number is the reference that is used by PMP if there is no “processor” directive in the main program, or if a unit is compiled separately.
The default processor frequency is the value that is used by PMP if there is no “frequency” directive in the main file, or if a unit is compiled separately. This field may be grayed if the processor has a fixed frequency.
The main file is the root file of the project, typically the “program” file.

PMP includes automatically all the files listed in “uses” or “external” declarations, but other files may be added.
The additional files list should contain the list of all these other files and it is used to build the project.
It may include .asm files (that should be assembled) or object files (passed directly to the linker).

2.4.1.2 Compiler options


Project | Compiler | Assembler | Linker | Processor options | Version and comments
New
Delete
Copy on top of the project main file
OK
Cancel

Project options for: Beacon

Include and unit paths:

Output path: D:\Prog\Delphi\pmp2006\Asm\

- ☐ Warnings off
- ☐ Large pointers
- ☐ Strict type checking
- ☑ Complete boolean evaluation
- ☑ Show compiler progression

Optimizer:
- ● Fastest code
- ○ Smaller code
- ○ Off

Assembler output:
- ☑ Verbose .asm output
- ☑ Commented .asm output
- ☑ Symbol table and memory usage in .asm output

Conditional defines:
- ☑ PK2 { Compile for PK2 evaluation board }

Include and unit paths is the list of paths (separated by a semicolon) that the compiler searches for an include file ($I directive) or a unit file (USES section).

Output path is the path where the compiler will output all the generated files. This is also the sources path for the assembler.

Toggle options:

Warnings off: {$W-} Suppress all warnings from the compiler; for individual warnings, see § 3.7.

Large pointers: {$POINTERS LARGE} Use large pointers (16 bits). For PIC18 and enhanced PIC16 this is default and cannot be changed. For other devices, it may be useful to use 8 bits pointers in small applications, since this generates smaller and faster code and uses less ram.

Strict type checking: {$STRICT} When activated, PMP will use strict type checking as a normal Pascal compiler.

Complete boolean evaluation: {$B+} Defaults to ON; if ON, every operand with AND and OR operators is guaranteed to be evaluated, even when the result of the entire expression is already known; if OFF, evaluation stops as soon as the result of the entire expression becomes evident in left to right order of evaluation.

Optimizer mode: Using the “fastest code” mode {$O SPEED} generates less calls to standard subroutines that are generated “inline”; this option consumes more program memory. Using the “smaller code” mode {$O MEMORY} generates systematically the calls to standard subroutines; average program memory size is shorter, but execution speed is a bit slower. Using the “Off” mode {$O OFF} disables any optimization functionality (not recommended): this is for debugging purpose since the generated code is a bit coarse.

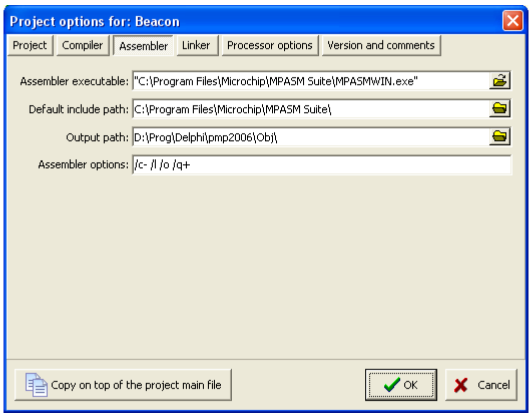
Verbose .asm output: The assembler output from the compiler will contain the Pascal source file echo and compiler information.

Commented .asm output: The assembler output from the compiler will contain comments on each assembler line of code.

Symbol table and memory usage in .asm output: Self commented.

Conditional defines: This list contains all conditional defines that may be used in the project. If an item is checked, it will be **Defined** for the compiler and cannot be re-**Defined** in code.

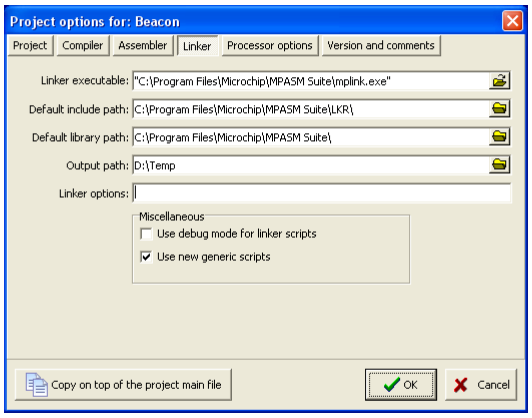
2.4.1.3 Assembler options



Assembler executable: "C:\Program Files\Microchip\MPASM Suite\MPASMWIN.exe"
Default include path: C:\Program Files\Microchip\MPASM Suite\
Output path: D:\Prog\Delphi\pmp2006\Obj\
Assembler options: /c- /l /o /q+

Assembler executable is the full path of the assembler that will be invoked to assemble assembly source files generated by PMP. By default this is the path of the MPLAB® general purpose assembler (typically MPAsmWin, found in the MPLAB® registry entries).
Default include path is the path of the assembler include files. Mainly this is where the assembler will find the .inc files that define all processors registers and constants, or generally where it will find any “include” file. PMP searches MPAsm™ files relatively to the MPAsm™ executable path.
Output path is the path where the assembler will produce the relocatable object files that will be the default path for the linker inputs.
Assembler options may contain additional assembler options. Default is options that are necessary for assembling PMP files.

2.4.1.4 Linker options

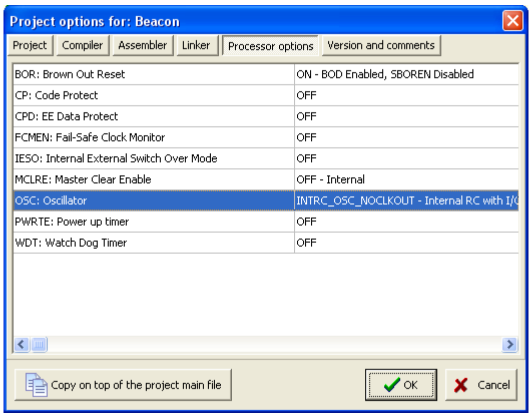


Linker executable: "C:\Program Files\Microchip\MPASM Suite\mplink.exe"
Default include path: C:\Program Files\Microchip\MPASM Suite\LKR\
Default library path: C:\Program Files\Microchip\MPASM Suite\
Output path: D:\Temp
Linker options:
☐ Use debug mode for linker scripts
☑ Use new generic scripts

Linker executable is the full path of the linker that will be invoked to link all the files of the project. By default this is the path of the MPLAB® general purpose linker (typically MPLink™, found in the MPLAB® registry entries).
Default include path is the path of the linker include files. Mainly this is where the linker will find the .lkr files that define the processors memory maps, or generally where it will find any “include” file. PMP searches MPLink™ files relatively to the MPLink™ executable path.
Default library path is the path of the linker library files. Mainly this is where the linker will find the .lib files used in the project.
Output path is the path where the linker will produce the output files.
Linker options may contain additional linker options. By default no additional option is necessary for PMP projects.

The "Use debug mode for linker scripts" check box forces the compiler to use the debug version of the .lkr script file, that reserves special memory areas for ICD usage. If the "Use new generic scripts" is checked, PMP will use the new scripts introduced since V8.30 of MPLAB, otherwise the "normal" script or the "i" version are used depending on the debug mode check box.

2.4.1.5 Processor options

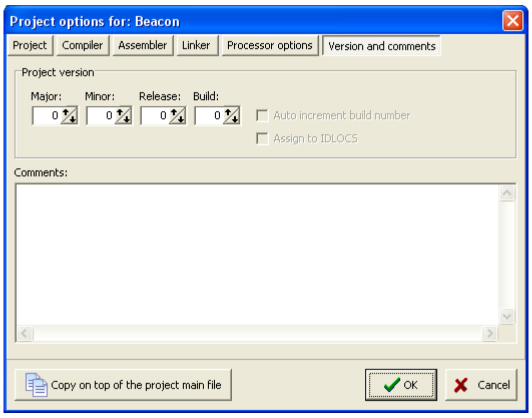


BOR: Brown Out Reset	ON - BOD Enabled, SBOREN Disabled
CP: Code Protect	OFF
CPD: EE Data Protect	OFF
FCMEN: Fail-Safe Clock Monitor	OFF
IESO: Internal External Switch Over Mode	OFF
MCLRE: Master Clear Enable	OFF - Internal
OSC: Oscillator	INTRC_OSC_NOCLKOUT - Internal RC with I/(
PWRTE: Power up timer	OFF
WDT: Watch Dog Timer	OFF

This tab lists all the selected processor's configuration bits and their possible values. The possible values are displayed and selected by right clicking on the value.

Warning: On PIC18 processors the extended instruction set must not be selected. PMP code is not compatible with extended mode.

2.4.1.6 Version and comments



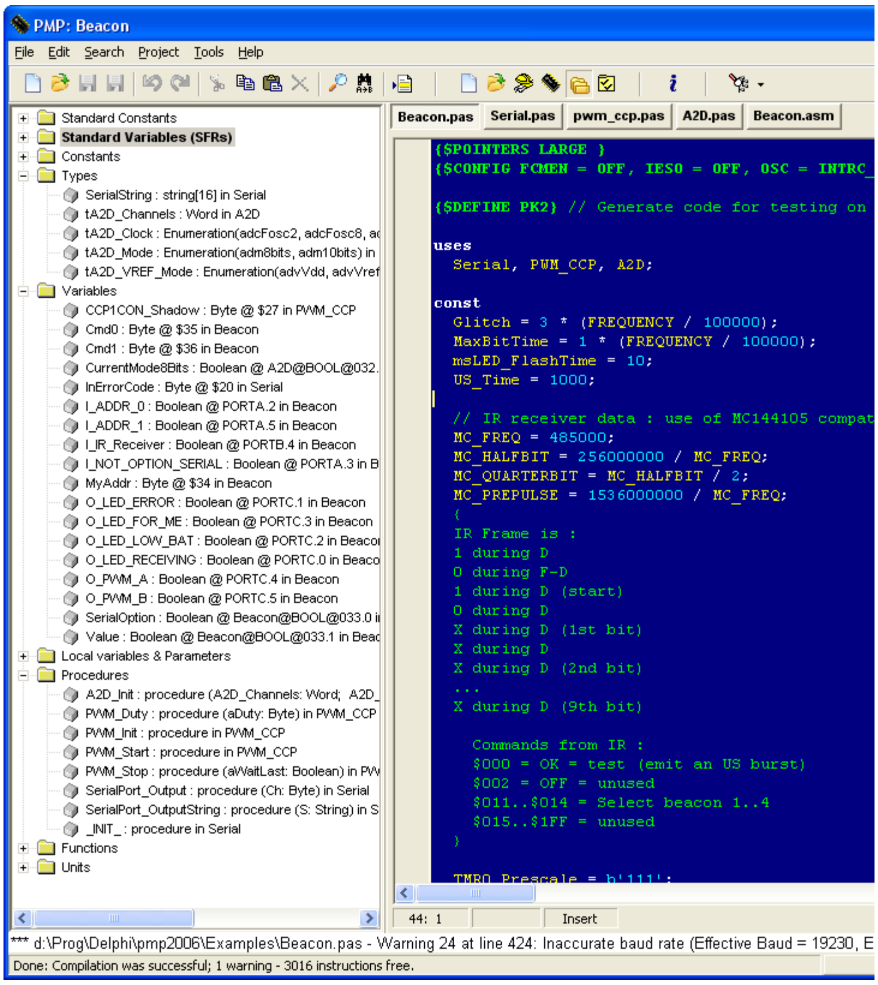
This tab gives access to the project's version information and comments.
Comments is free length text, saved with the project; PMP does not do anything with this text.

The build number may be auto-incremented after each explicit successful full build.

The version information may be automatically hard wired to the processor's IDLOCS so that this version information will be written to the device. The build written value is the value during the build, before the optional auto-incrementation.

2.4.2 Code Explorer

PMP IDE maintains a code explorer tree at the form's left (may me disabled in the "Tools" menu):



```
+ Standard Constants
+ Standard Variables (SFRs)
+ Constants
- Types
    SerialString : string[16] in Serial
    tA2D_Channels : Word in A2D
    tA2D_Clock : Enumeration(adcFosc2, adcFosc8, ac
    tA2D_Mode : Enumeration(adm8bits, adm10bits) in
    tA2D_VREF_Mode : Enumeration(advVdd, advVref
- Variables
    CCP1CON_Shadow : Byte @ $27 in PWM_CCP
    Cmd0 : Byte @ $35 in Beacon
    Cmd1 : Byte @ $36 in Beacon
    CurrentMode8Bits : Boolean @ A2D@BOOL@032.
    InErrorCode : Byte @ $20 in Serial
    I_ADDR_0 : Boolean @ PORTA.2 in Beacon
    I_ADDR_1 : Boolean @ PORTA.5 in Beacon
    I_IR_Receiver : Boolean @ PORTB.4 in Beacon
    I_NOT_OPTION_SERIAL : Boolean @ PORTA.3 in B
    MyAddr : Byte @ $34 in Beacon
    O_LED_ERROR : Boolean @ PORTC.1 in Beacon
    O_LED_FOR_ME : Boolean @ PORTC.3 in Beacon
    O_LED_LOW_BAT : Boolean @ PORTC.2 in Beacon
    O_LED_RECEIVING : Boolean @ PORTC.0 in Beaco
    O_PWM_A : Boolean @ PORTC.4 in Beacon
    O_PWM_B : Boolean @ PORTC.5 in Beacon
    SerialOption : Boolean @ Beacon@BOOL@033.0 i
    Value : Boolean @ Beacon@BOOL@033.1 in Beac
+ Local variables & Parameters
- Procedures
    A2D_Init : procedure (A2D_Channels: Word;  A2D_
    PWM_Duty : procedure (aDuty: Byte) in PWM_CCP
    PWM_Init : procedure in PWM_CCP
    PWM_Start : procedure in PWM_CCP
    PWM_Stop : procedure (aWaitLast: Boolean) in PW
    SerialPort_Output : procedure (Ch: Byte) in Serial
    SerialPort_OutputString : procedure (S: String) in S
    _INIT_ : procedure in Serial
+ Functions
+ Units
```

```
{$POINTERS LARGE }
{$CONFIG FCMEN = OFF, IESO = OFF, OSC = INTRC_

{$DEFINE PK2} // Generate code for testing on

uses
  Serial, PWM_CCP, A2D;

const
  Glitch = 3 * (FREQUENCY / 100000);
  MaxBitTime = 1 * (FREQUENCY / 100000);
  msLED_FlashTime = 10;
  US_Time = 1000;

  // IR receiver data : use of MC144105 compat
  MC_FREQ = 485000;
  MC_HALFBIT = 256000000 / MC_FREQ;
  MC_QUARTERBIT = MC_HALFBIT / 2;
  MC_PREPULSE = 1536000000 / MC_FREQ;
  {
  IR Frame is :
  1 during D
  0 during F-D
  1 during D (start)
  0 during D
  X during D (1st bit)
  X during D
  X during D (2nd bit)
  ...
  X during D (9th bit)

    Commands from IR :
    $000 = OK = test (emit an US burst)
    $002 = OFF = unused
    $011..$014 = Select beacon 1..4
    $015..$1FF = unused
  }

  TMR0_Prescale = b'111';
```

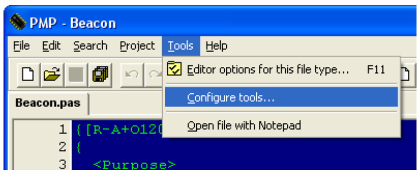
```
*** d:\Prog\Delphi\pmp2006\Examples\Beacon.pas - Warning 24 at line 424: Inaccurate baud rate (Effective Baud = 19230, E
Done: Compilation was successful; 1 warning - 3016 instructions free.
```

This tree presents all the project's constants, types, variables and procedures/functions used in the project.
By clicking on an object, the source code where this object is defined may be displayed.

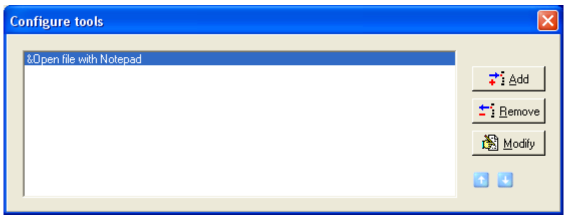
Warning: All information is extracted from the last compilation and so may be not accurate if changes had been made in the sources.

2.4.3 The tools menu

PMP has a tools menu that may be fully configured (as in TP or Delphi):

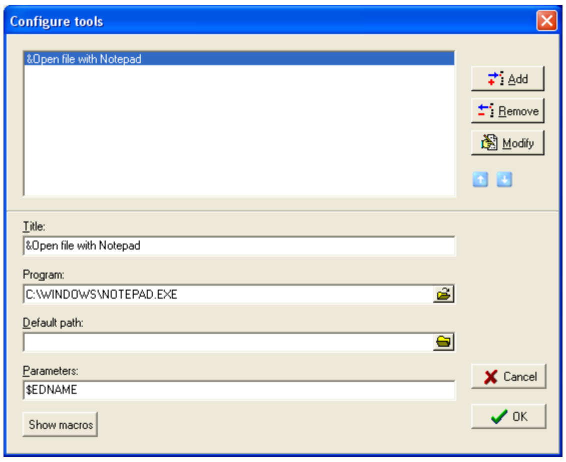


The menu option opens the configuration form:



In this short form, items may be moved up and down with the two small blue buttons.

The “Add” and “Modify” buttons extend the form:



Title: &Open file with Notepad

Program: C:\WINDOWS\NOTEPAD.EXE

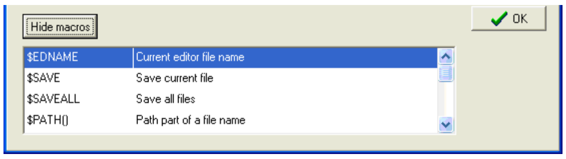
Default path:

Parameters: $EDNAME

The “Title” field may contain a standard “&” character to give the keyboard shortcut.

The “Parameters” field may contain macros as in TP and Delphi.

The “Show macros” button extends the form at bottom:



$EDNAME	Current editor file name
$SAVE	Save current file
$SAVEALL	Save all files
$PATH()	Path part of a file name

The “OK” button validates the modifications and returns to the short form.

<u>Available macros:</u>

Some macros are equivalent to TP/Delphi:

$EDNAME	returns the current edited file name, fully qualified (full path), with double quotes.
$SAVE	requests a save of the current file before further processing.
$SAVEALL	requests a save of all files before further processing.
$PATH()	returns only the directory part of the file name between the parenthesis. Warning: it is not quoted.
$NAME()	returns only the name part of the file name between the parenthesis. Warning: it is not quoted.
$EXT()	returns only the extension part of the file name between the parenthesis. Warning: it is not quoted.

Other macros are specific to PMP:

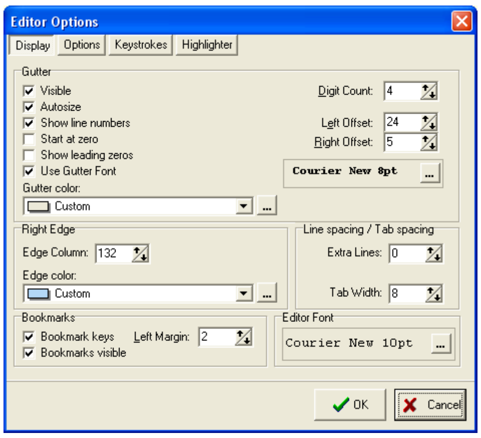
$PRJNAME	returns the current project file name, fully qualified (full path), with double quotes.
$ASMPATH	returns the compiler output path as defined in the current project options. Warning: it is not quoted.
$HEXPATH	returns the linker output path as defined in the current project options. Warning: it is not quoted.
$PROCESSOR	returns selected default processor.

2.4.4 Syntax highlighting and other editor options

PMP has syntax highlighting for several types of files (Pascal, C/C++, VB, ASM, …). The appropriate pattern is loaded according to the file extension; the user may change the colors and several behaviors.

All options may be modified through the Tools menu | "Editor options for this file type..." command that opens the options dialog:

General display options:



Gutter
- ✓ Visible
- ✓ Autosize
- ✓ Show line numbers
- ☐ Start at zero
- ☐ Show leading zeros
- ✓ Use Gutter Font

Digit Count: 4
Left Offset: 24
Right Offset: 5
Courier New 8pt
Gutter color: Custom

Right Edge
Edge Column: 132
Edge color: Custom

Line spacing / Tab spacing
Extra Lines: 0
Tab Width: 8

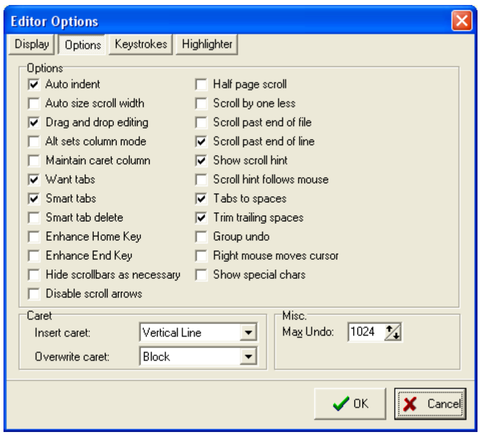
Bookmarks
- ✓ Bookmark keys
- ✓ Bookmarks visible

Left Margin: 2

Editor Font
Courier New 10pt

Some tips:
Bookmarks may be set by keyboard: Ctrl-Shift-n sets the bookmark n, where n is a digit key 0,,9. Then Ctrl-n jumps to the bookmark n. The result of several of these options may be shown directly by switching to the Highlighter tab, witch displays a mini editor.

Features options:



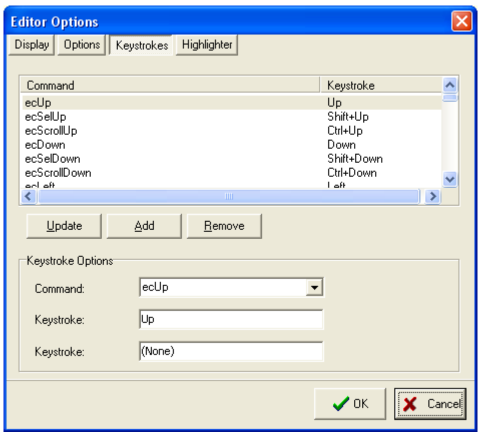
Options
- ✓ Auto indent
- ☐ Auto size scroll width
- ✓ Drag and drop editing
- ☐ Alt sets column mode
- ☐ Maintain caret column
- ✓ Want tabs
- ✓ Smart tabs
- ☐ Smart tab delete
- ☐ Enhance Home Key
- ☐ Enhance End Key
- ☐ Hide scrollbars as necessary
- ☐ Disable scroll arrows
- ☐ Half page scroll
- ☐ Scroll by one less
- ☐ Scroll past end of file
- ✓ Scroll past end of line
- ✓ Show scroll hint
- ☐ Scroll hint follows mouse
- ✓ Tabs to spaces
- ✓ Trim trailing spaces
- ☐ Group undo
- ☐ Right mouse moves cursor
- ☐ Show special chars

Caret
Insert caret: Vertical Line
Overwrite caret: Block

Misc.
Max Undo: 1024

Some tips:
"Right click moves cursor" is obsolete since right click displays a pop-up menu that contains functions as " do xxx on the yyy at current cursor location", so right click always moves the cursor.

Keyboard commands:

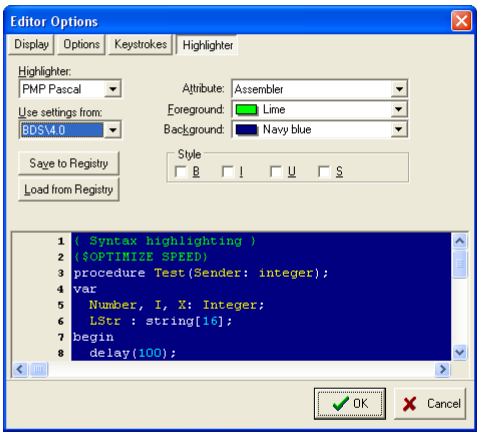


Command	Keystroke
ecUp	Up
ecSelUp	Shift+Up
ecScrollUp	Ctrl+Up
ecDown	Down
ecSelDown	Shift+Down
ecScrollDown	Ctrl+Down
ecLeft	Left

Keystroke Options
Command: ecUp
Keystroke: Up
Keystroke: (None)

This tab may be used to modify all keyboard commands.

Syntax highlighting:



Highlighter: PMP Pascal
Use settings from: BDS\4.0
Attribute: Assembler
Foreground: 🟩 Lime
Background: 🟦 Navy blue
Style: ☐ B ☐ I ☐ U ☐ S

```
{ Syntax highlighting }
{$OPTIMIZE SPEED}
procedure Test(Sender: integer);
var
  Number, I, X: Integer;
  LStr : string[16];
begin
  delay(100);
```

This tab may be used to modify all colors and appearance of the editor for the current type of file.

Clicking into the mini editor will update the selections according to the type of element recognized at the cursor position. Changes in the selections are shown immediately in the mini editor.

Some tips:

For Pascal highlighting, if the machine has a Delphi programming environment installed, the "Use settings from" feature allows settings to be imported from the Windows registry records for these IDEs, so that you can select the scheme you already use in your Delphi (for my personal use as shown, this is the old fashioned TP scheme).

Also for Pascal highlighting, the local PMP settings may be saved to the registry and restored.

For general highlighting, there are check boxes for selecting how comments should be treated (not shown here); several comment types may be active at a time.

Current file type scheme is saved in a separate .ini file upon dialog validation (OK button).

2.5 Generated files

PMP generates several files for each Pascal program or Unit that is compiled:

- ✓ A `.asm` file that contains the generated assembler language source file.
- ✓ A `.sym` file that contains information about a compiled unit for memory allocation and global variables (not readable by an editor).

PMP generates also a `pmp.lkr` file that serves as a relay for calling the linker.

PMP generates also a `<project name>.tree` file that stores the project symbols tree for the code explorer.

The windows version of PMP generates several `.ini` files that tracks user setup for each file type (syntax highlighting …) and a main `.ini` file that holds global user settings and opened files, and a `.pmp` project file that holds user setup for the project.

For the MPAsm™ and MPLink™ generated files, see MPLAB® documentation.

2.6 Configuration file

PMP uses a configuration file named `processors.cfg` that must be in the same directory as the PMP executable.

This file has an `.ini` structure; it is used to describe special behaviors of processors that cannot be deducted from the MPAsm™ and MPLink™ files.

It can be customized to declare new processors than have not been evaluated yet.

If a processor is not explicitly described in the configuration file, it is assumed to have no specific behaviors and will use defaults.

The structure of each entry is:

```
[Processor Part Number]
Item_1=Value_1
…
Item_n=Value_n
```

Or:

```
[Processor Part Number Family]
Item_1=Value_1
…
Item_n=Value_n
```

Where “Processor Part Number Family” may contain “?” or “*” wild-card characters to match several processors. In this case, the rule is that a processor name that matches a family name will primarily default to this family parameters. If the processor has specifics regarding its family, these specifics (differences from the family) are defined further in a specific section.

Item	**Type**	**Default**	**Comment**
StackSize	Integer	0	Defines the hardware stack deep; some processors have only a 2 levels stack. 0 if standard 8 levels stack.
NoReturn	Boolean	0	Defines if there is no return instructions (only retlw x instructions)
NoAddlw	Boolean	0	Defines if there is no addlw / sublw instructions
NoInterrupts	Boolean	0	Defines if there is no interrupts
Call256	Boolean	0	Defines if calls and PCL assignments are only possible to the first 256 program locations.
FixedFrequency	Integer	0	Defines the processor fixed clock frequency; in this case the $FREQUENCY compiler directive is forbidden.
HasUart	Boolean	0	Defines if the processor has an UART.
Uart_Port	String	PORTB	If the processor has an UART, defines on which port.
Uart_RxBit	Integer	5	If the processor has an UART, defines Rx on witch pin.
Uart_TxBit	Integer	7	If the processor has an UART, defines Tx on witch pin.
OscCal	String		For processors that have a oscillator calibration feature; possible values are movlw or retlw; defines witch type of instruction returns the calibration value in the W register at reset. If retlw, a call to the last program code location is inserted at start-up; if movlw, it is assumed that there is a PC wrap to 0 after last program location execution.
Enhanced	Boolean	0	Identifies the processor as a PIC16 enhanced mid-range processor

Tip:

NoReturn=1 and NoAddlw=1 together force PMP to use the PIC10/PIC12 code generator; this is used for some very reduced instruction set PIC16 processors.

3 Language features

3.1 BNF representation in this document

This document uses a simplified BNF style representation that came from TpLex styles:

- ✓ Characters in **bold** face are literal characters.
- ✓ Items between parentheses are optional.
- ✓ The vertical bar | denotes an option (synonym of "or").
- ✓ A star after an item denotes the possible repetition of this item.
- ✓ A ::= is a synonym of "is".

Examples:

<Literal Comment> ::= **(* This is a comment *)**	This is a comment that is fully literal.
<Another Comment> ::= **(*** <Your Comment> ***)**	This is a comment that is described by a reference to another item that should be described further.
<Optional Comment> ::= (**(* Comment *)**)	This is a literal comment that is optional.
<Uses declaration> ::= (**USES** <Identifier>(**,** <Identifier>)***;**	This is a unit **USES** declaration that is optional and may contain several unit references, separated by a comma and terminated by a semicolon.
<Source File header> ::= **PROGRAM|UNIT** <Identifier>;	This is a source file header that can be either a PROGRAM main source file or an UNIT source file.

3.2 Keywords

The following keywords are implemented and / or reserved by the PMP language (**bold** faced if not in standard TP/Delphi Pascal):

ABS[4], ABSOLUTE, AND, ARCCOS[5], ARCSIN[5], ARCTAN[5], ARRAY, ASM, ASSIGN, **BAUD**, BEGIN, **BITS**, BOOLEAN, BREAK, BYTE, CASE, CHAR, **CLR**, **CLRWDT**, **CODESEG**[3], CONFIG, CONST, CONTINUE, COS[5], DEC, **DECLARED**, **DEFINED**, **DELAY**, **DELAY_MS**, **DELAY_NS**, DISPOSE[5][6], DIV, DO, DOUBLE[5], DOWNTO, **DWORD**, **EEREAD**, **EEWRITE**, ELSE, END, **ERROR**, EXCLUDE, EXIT, EXP[5], EXTERNAL, FOR, FORWARD, **FP_CLR**[5], **FP_IOP**[5], **FP_OVR**[5], **FP_UND**[5], FREEMEM[5][6], **FREQUENCY**, FUNCTION, GETMEM[5][6], GOTO, **HEX**, HI, HIGH, **IDLOC**, IF, IMPLEMENTATION, IN, INC, INCLUDE, INITIALIZATION, INLINE[1], INPUT, INTEGER, INTERFACE, INTERRUPT, LENGTH, LIBRARY[1], LN[5], LO, LONGINT, LONGWORD, LOW, MAXINT, MAXLONGINT, MAXWORD, MEMAVAIL[5][6], MOD, **MOD16**, **MOD16S**, **MOD32**, **MOD8**, MOVE, **MUL18**[2], NEW[5][6], NIL, NOP, NOT, ODD, OF, ON, **OPTION**, OR, ORD, OUT, OUTPUT, OVERLOAD[3], PACKED[1], PI[5], POINTER, POW[5], PRED, PROCEDURE, PROGRAM, **PWM**, READ, READLN, REAL[5], RECORD, REPEAT, **RESET**, **ROL**, **ROR**, ROUND[5], SET, **SFR**, SHL, SHORTINT, SHR, SIN, SINGLE[5], SIZEOF, **SLEEP**, SQR[4], SQRT[5], STR, STRING, SUCC, **SWAP**, TAN[5], THEN, TO, **TRIS**, TRUNC[5], TYPE, UNIT, UNTIL, UPCASE, USES, VAL, VAR, **VERSION**, VOLATILE, WHILE, WITH, WORD, WRITE, WRITELN, XOR.

(1): For future compatibility of source codes, some keywords that are used in other Pascal implementations such as TP or Delphi are already reserved even if not implemented. Even if not reserved by PMP, it should be avoided to use such keywords as identifiers.
(2): Not implemented for all processors.
(3): Same as (1) but the implementation is already scheduled in a future version.
(4): Limited to integer types for processors other than PIC16 and PIC18.
(5): Only for PIC16 and PIC18 processors.
(6): Not for PIC16 non-enhanced processors.

3.3 Constant formats

PMP supports standard Pascal constant formats as well as additional ones that programmers like (sorry but octal is not supported):

$<hex>	hexadecimal constant (8..32 bits) (TP/Delphi format)
0x<hex>	hexadecimal constant (8..32 bits) (C format)
h'<hex>'	hexadecimal constant (8..32 bits) (MPAsm™ format)
0b<binary>	binary constant (8..32 bits) (C format)
b'<binary>'	binary constant (8..32 bits) (MPAsm™ format)
'<character>'	single character constant (ASCII), evaluated as **CHAR**.
#n	single character constant (ASCII), given for compatibility only since PMP type checking is limited, #n is equivalent to n, but evaluated as **CHAR**.
#$<hex>	single character constant (ASCII), given for compatibility only since PMP type checking is limited, #$<hex> is equivalent to $<hex>, but evaluated as **CHAR**.
'<string>'	string constant **(*)**

(*): Character literals (TP / Delphi ^n form, or C \n form) are not allowed**;** for concatenation use the + operator construction with a literal numeric.

Note that PMP is poorly typed on variables and constants (except for strings, records and arrays).

There is a special syntax behavior for b'<binary>' and h'<hex>': spaces, dots, dashes and underscores are accepted inside the quoted string for improving readability; they are ignored.

3.4 Program structure

A typical PMP program contains the following elements:

```
Program <program_name>;

<Hardware declarations>
(<USES declaration>)
(
(<Constant declaration>)* |
(<TYPE declaration>)* |
(<Variable declaration>)* |
(<Procedure or function declaration>)*
)*

Begin
        (<Statement>)*
End.
```

PMP syntax is as close as possible to the TP / Delphi syntax.

All programs must begin with the **PROGRAM** keyword, followed by the name of the program.

<Program name> must match the file name and cannot be expressed as a string, so long file names are allowed but they must contain only valid identifier characters.

The program body must contain some declaration about the target processor, contained in the hardware declaration section.

PMP is a single pass compiler**;** as required by standard Pascal, all identifiers that are to be used in the remainder of the code must be declared in a declaration section of the program or in a procedure or function prior to their use.

The program may contain some optional special procedures qualified with the **INTERRUPT** modifier (see § 3.15), which is used to define sets of statements that is intended to serve as an interrupt service routine.

3.5 Unit structure

A typical PMP unit contains the following elements:

Unit <Unit_name>**;**

Interface
(<USES declaration>)

(
(<Constant declaration>)* |
(<Type declaration>)* |
(<Variable declaration>)* |
(<Procedures and functions declaration>)*
)*

Implementation

((<Constant declaration>)* | (<Type declaration>)*| (<Variable declaration>)* | (Procedures and functions declaration)*)*

(**Initialization**
        (<Statement>)*
)
End.

<Unit name> must match the file name and cannot be expressed as a string, so long file names are allowed but they must contain only valid identifier characters.

Symbols, procedures and functions that are declared in the interface section will be visible outside the unit (declared as "global" for the linker).

Procedure and function arguments declarations in the implementation section must match the declarations of the interface section, or may be omitted (as in TP / Delphi).

PMP does not support "USES" declaration in the implementation section, so take care about circular references between units.

PMP supports "initialization" section in units. Initializations are called before execution of the main program statements.

A unit cannot have a processor definition, since it should be build for a main program. So if a unit is compiled directly, PMP uses the default selected options in the Project Options form. This allows a compilation for syntax checking.

When a unit is compiled during a project "make" or "build", the unit uses the main program processor definition.

A unit generates its own .asm file and a .sym file**;** .sym file format is specific to PMP, it lists all exported and imported functions, procedures and variables. PMP takes care about unit .pas and .asm file time stamp match, and will rebuild units that are out of date.

Since some optimizations are processor dependant, if the processor declaration or frequency declaration of a previous compilation of a unit does not match the main program declaration, PMP automatically rebuild the unit with the main program processor and frequency declarations.

The unit may contain some optional special procedures qualified with the **INTERRUPT** modifier (see § 3.15), which is used to define sets of statements that is intended to serve as an interrupt service routine.

3.6 Comments

PMP programs may contain comments in several syntaxes:

Standard Pascal syntax:

{ (<Comment>) **}** | **(*** (<comment>) ***)**

Delphi syntax (all characters until end of line is a comment):

// (<Comment>)

They may appear on a line by themselves, or combined with program statements.

3.7 Directives

- **Directives cannot be inserted in expressions or assignments; they must be outside of statements.**

Directives are special comments that change compiler code generation. The syntax is standard Pascal syntax:

<Directive> ::= **{$** <Directive Item> (<Any comment>) **}** | **(*$** <Directive Item> (<Any comment >) ***)**

A space may be inserted after the $ sign (not standard Pascal). Spaces are not significant in directive items.

In below descriptions, some directives or arguments has synonyms, they are separated by a |.

+|- toggles may be replaced by (are aliases of) ON|OFF toggles for directives that apply.

<Any comment> is to specify that any text after the directive parameters (if any) is considered as comment.

3.7.1 $C | $CODEGEN - PLC mode code

1+ | **1-** or **PLC** | **NORMAL**

Toggles special PLC mode bit operations generation (default is off or NORMAL)**;** bit operations and statements are limited, see chapter 4.1.5.

3.7.2 $CONFIG - Configuration bits

<ConfigItem>(**,** <ConfigItem>)*|NONE

NONE keyword – used alone – tells the compiler to not generate the CONFIG directive for the assembler. This is useful when using a bootloader that supersedes automatically all options.

The $CONFIG NONE directive may coexist with other $CONFIG directives, but simply they will not be generated.

This is the new syntax for **configuration bits** that matches the assembler `config` directive multi-byte complexity (see § 3.8 for the old alternate CONFIG syntax).
This directive is not allowed in units.
This directive may be used several times, but cannot be used after code generation has started.
<ConfigItem> ::= <ConfigName> **=** <Value>
Where <ConfigName> is a legal configuration bit known by MPASM include file. The different possible names and values may be found in the Project Option's dialog.
Where <Value> is a legal configuration bit / group of bits known by MPASM include file. Identifiers starting with a letter may be used as is; names starting with a digit should be quoted to be read as a string. Nevertheless, any identifier may be quoted.
For processors smaller than PIC18, this directive also accepts definition of configuration bits, with an equivalent <ConfigName> equal to the left part of the name (without leading underscore character) and <Value> equal to right part of the name (see example below).

Example:
```
{$CONFIG CPUDIV = OSC1_PLL2, FOSC = XT_XT, BORV = '1_9'}
{$CONFIG CPD=OFF, BOR=NSLEEP}
// This last one is equivalent to the old format:
CONST CONFIG = _CPD_OFF and _BOR_NSLEEP;
```

3.7.3 $DEPRECATED – Define a unit, procedure or function as obsolete

<Message>

Define the current unit, procedure or function as obsolete. If it is used, the compiler will output a warning with the given <Message> (quoted string).

Example:

```
{$DEPRECATED 'This procedure is kept for compatibility, use XYZ instead' }
```

3.7.4 $EEPROM - Define useable EEPROM area

<Address expression> [**TO** <Address expression>]

Defines EEPROM memory area to use for variables that will be allocated after this point (default is all available EEPROM). The first parameter defines only the EEPROM start area, the optional second parameter defines EEPROM top too (last usable address). This directive is local to the current module.

Example:

```
{$EEPROM $10 TO $FF }  // $0 to $0F reserved for direct use
```

3.7.5 $EOL – Define end of line behavior

CR | CRLF

Defines behavior for end of line in WRITELN. CR puts only a CR ($0D), and CRLF (default) puts a CR and LF pair ($0D-$0A). If this directive is present before a USES statement, it propagates to the used units. For READLN, CR only matters and LF characters are ignored.

Example:

```
{$EOL CR }
```

3.7.6 $EXTENDED – Enable / disable extended Pascal syntax - NEW! (V1.6.0)

ON | OFF

Tell the compiler to enable or disable some non-standard Pascal syntax. Default is ON.
If the extended syntax is enabled PMP implements some nice features taken from other Pascal-like languages such as MODULA or OBERON. These extended syntax are:

- ✔ A BY clause in the FOR loop statement,
- ✔ A LOOP... END statement,
- ✔ A RETURN statement in functions.
- ✔ An ELSEIF statement for the IF statement.

Example:

```
{$EXTENDED OFF }  // Standard Pascal
```

3.7.7 $FREQUENCY - Processor frequency

<frequency>

Defines processor operating frequency**;** this directive overrides the default frequency defined in the Project Options form. For the command line version of PMP, it may be defined on command line arguments (or defaults to 4 MHz). This directive is not allowed in units since they use main program declaration. A frequency multiplier can be used (4000 KHz, 8 MHz …), but dots are not allowed (4.5 MHz is forbidden, use 4500 KHz instead). Case is not sensitive (since mHz should not be 1/1000th of Hertz!). A CONST pseudo variable named FREQUENCY is generated that holds the value and may be used in expressions.

Example:

```
{$IFDEF LOW_POWER}
  {$FREQUENCY 31000}
{$ELSE}
  {$FREQUENCY 20 MHz}
{$ENDIF}
```

3.7.8 $IDLOC Define user ID bytes

[**BYTE**] <IdLocString> **|** <IdLocItem>(**,** <IdLocItem>)*

New syntax for setting IDLOC bits.
This directive is not allowed in units.
This directive may be used several times, but cannot be used after code generation has started.
The optional BYTE tag forces the compiler to generate a byte for each item, for processors where ID locations may be byte wide.

<IdLocString> may be a single quoted string, limited in size to the max number of IDLOC of the processor (typically 4); this syntax is maintained for compatibility with MPASM old syntax format.
Special behavior: if the ID locations limits for the current processor is 0..15, than each digit is interpreted as hexadecimal digits and translated as 0..9, A..F (0..15).

<IdLocItem> ::= <IdLocIndex>**:** <IdLocValue>

<IdLocIndex> is an IDLOC integer expression that returns an index in the 0..n range, where n depends to the current processor configuration.

<IdLocValue> is an IDLOC integer expression that returns a value compatible with the IDLOC format of the current processor configuration.

- Depending to the processor IDLOC values may be read and / or written by code (see IDLOC pseudo variable).
- If not defined, an IDLOC item is set to it's maximum value.
- If enabled, the project's version values always override the values defined by this directive.

Example:

```
CONST MY_IDLOC_3 = $0A;
{$IDLOC 0:1, 1:$FF, 2:$AA, 3:MY_IDLOC_3 }
```

3.7.9 $I | $INCLUDE - Include source file

<FileName>

Include another file in place. <FileName> may be a simple file of the form Name.Ext or a string constant between quotes (mandatory if the file name contains a path and/or spaces). If the file name does not contain an extension, .pas is assumed. See global include path declaration in chapter 2.1.

Example:

```
{$INCLUDE TEST1 } // include TEST1.PAS
{$INCLUDE TEST2.INC }
{$INCLUDE 'This is a long file name.pas' }
```

3.7.10 $INIT | $INITIALIZE - Define start-up initializations

<Item>[(, <Item>)*]

Initialize some processor features at start-up (reset). <Item> may be one of the following:

ALL:

Include all possible initializations (see below).

PORTS [**INPUT**|**OUTPUT**]:

If INPUT is specified or by default, set all LATx or PORTx registers to zero then initialize all TRISx registers to $FF (all inputs); otherwise, set all LATx or PORTx registers to zero then initialize all TRISx registers to $00 (all outputs)

ANALOGS:

Initialize all analog registers to zero so that the processor is ready to do digital I/O on all pins. Depending on the processor this includes: ANSEL, ANSELH, ADCON1 (or sets ADCON1 or ANCON0/ANCON1 to all digital inputs for PIC18). Depending on the processor, digital I/O may need to initialize COMPARATORS too, see below.

RAM:

Initialize all used ram variables (explicit variables only) to zero. By default PMP does not initialize memory. Note that if used all the HEAP memory pointers / structures are always reset during start-up.

INTERRUPTS:

Initialize all interrupt registers to zero. Depending on the processor this includes INTCON, INTCON2, INTCON3, PIE1, PIE2, PIR1 and PIR2.

COMPARATORS: **NEW! (V1.5.4):**

Initialize all comparators enabling registers to set all I/O as digital. Depending on the processor, actually it includes only CMCON.

This directive cannot be used inside the main program block since initialization call is done just before.
This directive cannot be used in units.
This directive supersedes a previous one (the last one sets initialization flags, any previous one is discarded).

Example:

```
{$INIT RAM, ANALOGS }
```

3.7.11 $INTERRUPTS – Enable / disable / define interrupts

ON | OFF

Tell the compiler that we don't need interrupts so that interrupt entry points are not generated. This directive cannot be found after an interrupt procedure had been defined, or after the begin of the main program block. Default is ON.

Example:

```
{$INTERRUPTS ON }
```

UNIQUE | MULTIPLE

Tell the compiler that we need only one interrupt procedure (UNIQUE) or more than one (MULTIPLE), so that interrupt entry points are optimized (see interrupt procedure format). This directive cannot be found after an interrupt procedure had been defined, or after the begin of the main program block. Default is MULTIPLE. This has nothing to do with PIC18 low/high interrupts and priorities (we may have an unique interrupt or multiple ones per priority).

3.7.12 $JUMPS – Define jumps range

SHORT | LONG (processor PIC18 only)

By default, PMP generates BRA instructions for jumps within a block to minimize code size; this is enough for most programs but in some circumstances the range of a BRA instruction may be not enough. SHORT generates BRA after this point, LONG generates GOTO instructions after this point, until another $JUMPS directive.

Example:

```
{$JUMPS LONG }
```

3.7.13 $O | $OPTIM | $OPTIMIZE - Define optimization mode

S | SPEED

Optimize for speed; some internal function calls are replaced by inline code (call instructions to these functions are suppressed: less use of processor hardware stack) and code optimization is more accurate. It invalidates the memory optimization (see below).

M | MEMORY

Optimize for memory size and calls (default); internal functions are mostly implemented as subroutines; processor hardware stack is used for calls. It invalidates the speed optimization (see above).

+ | -

Globally activate or deactivate the optimizer. Defaults to the Project options setting.

Example:

```
{$OPTIMIZE SPEED}
```

3.7.14 $OPTIMIZE_PARAMS - Define parameter passing optimization mode

ON | OFF

By default parameters passing may be optimized (passed in registers). In some cases it may be necessary to turn off this optimization, for pure assembler routines or procedures assigned to **output** channel. Defaults to ON.

Example:

```
{$OPTIMIZE_PARAMS OFF}
```

3.7.15 $OSCCAL – Activates or deactivates OSCCAL processor feature

ON | OFF

For processors that have an Oscillator Calibration feature, tells to the compiler to generate code at reset to get calibration value in W register and to set-up OSCCAL register. Cannot be used after startup code generation (in program main block). Defaults to OFF.

Example:

```
{$OSCCAL ON}
```

3.7.16 $P | $POINTERS - Define pointers size

- **For PIC18 and PIC16 Enhanced processors the LARGE model is forced and cannot be changed.**

S | SMALL

Optimize for using one byte for pointers and VAR arguments (this is default); this limits allocation of pointed to variables in the lowest 256 bytes of memory. PMP does not search to optimize space for such variables. This directive cannot be used after code has been generated. This directive is only available in a program (project wide directive); units will be automatically compiled with the same option as the main program.
This directive is ignored for PIC18 and PIC16 enhanced processors, for witch pointers and VAR argument addresses are always 16 bits wide (LARGE). A warning will be issued and the $POINTERS LARGE will remain active.

L | LARGE

Optimize for using two bytes pointers and VAR arguments; this permits allocation of such variables in the whole memory, but code is less efficient since parameter passing consumes more memory and code, and using of FSR must be coupled with the use of IRP flag. PMP does not search to optimize space for such variables. This directive is ignored for small devices that have not more than 256 bytes of memory. This directive cannot be used after code has been generated. This directive is only available in a program (project wide directive); units will be automatically compiled with the same option than the main program.
This directive is not allowed for PIC18 and PIC16 enhanced processors, for witch pointers and VAR argument addresses are always 16 bits wide.
This directive is ignored for processors that have no upper ram (all ram is within 0,,255 address range). A warning will be issued and the $POINTERS SMALL will remain active.

Example:
```
{$POINTERS LARGE}
```

3.7.17 $POP – Restore compiler options

This directive restores the previously saved options (see details in $PUSH, below).

3.7.18 $PROCESSOR - Define processor

<processor>

Defines the processor part number; this directive overrides the default processor part number defined in the Project Options form. For the command line version of PMP, it must be used if the processor is not defined in command line arguments. This directive is not allowed in units since they use the main program declaration. In main programs this directive should be the first to be declared. The processor part number is an identifier and must match the MPAsm™ processor and linker script include file names[1], which give the standard register definitions and the memory mapping of the processor. Imported identifier definitions are visible to the program (such as standard registers and bit definitions), and cannot be redefined.

- Tip: The given <processor> is automatically defined as a conditional compilation symbol.

Example:
```
{$PROCESSOR PIC16F690}
…
{$IFDEF 'PIC16F*' }
{$DEFINE MIDRANGE}
{$ENDIF}
```

[1] For mysterious considerations .inc and .lkr file names are different in the MPLAB® suite; PMP translates automatically the file names: PIC16F84 is translated to P16F84 for .inc file and translated to 16F84 for .lkr file. This is not very portable…

3.7.19 $PUSH – Save compiler options

This directive saves the current compiler options so that some of them may be modified locally, then restored with a $POP directive.
Several $PUSH may be nested (options are saved in a stack).
The saved options are: $WARNING (global and individual), $CODEGEN, $CONFIG (ON/OFF), $INITIALIZE, $INTERRUPTS, $JUMPS, $OPTIMIZE, $OSCCAL, $POINTERS and $SPACE.

3.7.20 $RESERVED RAM | EEPROM - Specify a memory region as unusable

<Address expression> [**TO** <Address expression>]

Defines a RAM or EEPROM address or consecutive addresses that are not usable by PMP. This directive may be used several times to define more than one area, or different areas may be defined in one directive, separated by a comma. If an address is already used by PMP or does not exist, an error occurs. Warning: in the current implementation, this directive is local to the current unit, and it is global if defined in the main program, before the "uses" section.

Examples:

```
{$RESERVED RAM $100, $110 TO $11F comment: reserved for XYZ module }
{$RESERVED EEPROM $10, $11, $F0 TO $FF}
CONST
  MOD2_RAM_START = $200;
  MOD2_RAM_LENGTH = $10;
{$RESERVED RAM MOD2_RAM_START TO MOD2_RAM_START + MOD2_RAM_LENGTH - 1 }
```

3.7.21 $SCRIPT – Define the linker script to use

<String expression>

Define (override) the linker script file to use. <String expression> is the file name to use, without character wild-cards. It may contain a full path. If there is no extension .lkr is assumed.
This directive had been added to make it possible to use special case scripts (user modified linker scripts), for using a boot loader for example.
This directive must be used BEFORE any other processor characteristics-related directive except $PROCESSOR.
This directive overrides the script that may be defined in the project's options.

Example:

```
{$IFDEF BOOT_LOADER}
{$SCRIPT 'P18_BOOT_g.LKR'}
{$ENDIF}
```

3.7.22 $S | $SPACE - Switch memory allocation to RAM or EEPROM

RAM

Defines that variables are generated in RAM area (default). This directive is effective until another SPACE directive.

EEPROM

Defines that variables are generated in EEPROM area. This directive is effective until another SPACE directive or function or procedure declaration.

Example:

```
{$SPACE EEPROM}
VAR
  EE_SAVED_INDEX: BYTE; // Index saved in EEPROM
{$SPACE RAM}
```

3.7.23 $STRINGS - Set default string size

<expression>

Defines the default size of strings when there is no size qualifier. This new size will be used after this line of code. This directive is local.

Example:

```
{$STRINGS 32} // Set default string size to 32 characters
```

3.7.24 $UERROR - Generate a compiler error

<Message>

Raise a user error message with the given text. <Message> must be enclosed by standard character string quotes. The error will count in the total compiler error count so that linker will not be called.

Example:

```
{$IFDEF 'PIC18*'}
{$UERROR 'Sorry this module cannot be used with a PIC18 processor'}
{$ENDIF}
```

3.7.25 $UWARNING - Generate a compiler warning

<Message>

Display a user warning message with the given text. <Message> must be enclosed by standard character string quotes.

Example:

```
{$IF FREQUENCY < 4000000}
{$UWARNING 'Inaccurate timings will occur at this frequency'}
{$IFEND}
```

3.7.26 $V $VARIABLES Define the memory region for RAM variables

<Address expression> [**TO** <Address expression>]

Defines RAM memory area to use for variables that will be allocated after this point (default is all available RAM). The first parameter defines only the RAM start address, the optional second parameter defines RAM top too (last usable address). Check is made regarding to the processor configuration. This directive is local to the current module.

Example:

```
{$VARIABLES $20 TO $6F}
```

3.7.27 $VECTORS - Define reset and interrupt vectors

➢ **This directive replaces the $RESET directive that was too restricted and have been removed.**

PIC18: [**RESET=**<Address>][**,INT_LOW=**<Address>][**,INT_HIGH=**<Address>]
Others: [**RESET=**<Address>][**,INT=**<Address>]

Defines the RESET and/or the interrupt vectors to the given <Address> expression. This is where PMP places a jump to the appropriate code. All items are optional, if not given the default address is used. The programmer is allowed to change the vectors so that he is able to install he's own start-up code (a boot-loader or any) that installs itself at $0000. This directive cannot be used if some code has been already generated. This directive is only available in a program (project wide directive).

Example:

```
{$IFDEF BOOTLOADER}
  {$VECTORS RESET=$800, INT=$804}
{$ENDIF}
```

3.7.28 $W | $WARN | $WARNING - Define a compiler warnings behavior

n (**+** | **-**)

Validate or invalidate the compiler warning number n after this point.

+ | -

Validate or invalidate all the compiler warnings after this point of source code. This does not reset individual warning off (as defined above).

Example:

```
{$W 10-}
```

3.7.29 Conditional compilation

PMP allows conditional compilation by means of conditional directives.

Conditional directives are special comments that change the compiler code generation. The syntax is a standard TP / Delphi syntax:

<Conditional directive> ::= **{$** <Conditional Directive Item> **}** | **(*$** < Conditional Directive Item > ***)**

Where:

<Conditional Directive Item> ::= <Conditional Define Item>|<Conditional Expression Item>

<Conditional Define Item> ::= **DEFINE** <Conditional Identifier> | **UNDEF** <Conditional Identifier> | **IFDEF** <Conditional Identifier> | **IFNDEF** <Conditional Identifier> | **IFOPT** <Directive Item> | **ELSE** | **ENDIF**

<Conditional Expression Item> ::= **IF** <Constant Conditional Expression> | **ELSEIF** <Constant Conditional Expression> | **ELSE** | **IFEND**

<Constant Conditional Expression> is a standard <Constant Integer Expression> with additional built-in function capabilities: **Defined(**<Conditional Identifier>**)** and **Declared(**<Identifier>**)**, see below.

<Constant Conditional Expression> is finally evaluated as a boolean (anything not zero is true).

<Conditional Identifier> is not a Pascal identifier and cannot be referenced in real program code. Likewise, Pascal identifiers cannot be referenced as <Conditional Identifier>.

A <Conditional Identifier> works like boolean variables: it is true (defined) or false (not defined). Any valid conditional identifier is assumed to be false until it is defined. The **$DEFINE** affects the true value to the specified <Conditional Identifier> and the **$UNDEF** affects it the false value.

<Directive Item> is any switch option directive (CODEGEN, OPTIMIZE, SPACE, POINTERS, WARNING) as defined in the previous chapter.

$DEFINE <Conditional Identifier>
Defines a conditional identifier.

$UNDEF <Conditional Identifier>
Un-defines a conditional identifier.

$IFDEF <Conditional Identifier> | **$IFDEF '**<Conditional Identifier String>**'**
If the given conditional identifier is defined, the next code compiles until a **$ELSE, $ENDIF** or **$IFEND** conditional directive. The second form accepts '?' and '*' wildcards in the string to match more than one symbol.

$IFNDEF <Conditional Identifier> | **$IFNDEF '**<Conditional Identifier String>**'**
If the given conditional identifier is not defined, the next code compiles until a **$ELSE, $ENDIF** or **$IFEND** conditional directive. The second form accepts '?' and '*' wildcards in the string to match more than one symbol.

$ELSE
If all the previous **$IF**, **$ELSEIF**, **$IFDEF** or **$IFNDEF** conditional directives had been evaluated to False, the next code compiles until a **$ENDIF** or **$IFEND** conditional directive.

$ENDIF
Terminates a **$IFDEF, $IFNDEF** or **$IF** conditional directive block (synonym to **$IFEND**).

$IF <Conditional Expression>
If the given expression evaluates to TRUE (anything not zero is true), the next code compiles until a **$ELSE, $ENDIF** or **$IFEND** conditional directive. <Conditional Expression> may contain calls to additional built-in functions: **Defined (**<Conditional Identifier>**)** and **Declared (**<Identifier>**)**, see below.

$ELSEIF <Conditional Expression>

If all the previous **$IF**, **$ELSEIF**, **$IFDEF** or **$IFNDEF** conditional directives had been evaluated to FALSE, and if the given expression evaluates to TRUE (anything not zero is true), the next code compiles until a **$ELSE, $ENDIF** or **$IFEND** conditional directive. <Conditional Expression> may contain calls to additional built-in functions: **Defined (**<Conditional Identifier>**)** and **Declared (**<Identifier>**)**, see below.

$IFEND

Terminates a **$IF**, **$IFDEF** or **$IFNDEF** conditional directive block (synonym to **$IF**).

$IFOPT <Directive Item>

If the given directive item is in the given state, the next code compiles until a **$ELSE** or **$ENDIF** conditional directive.

Example:
```
{$IFOPT SPACE RAM}
  …
{$ENDIF}
{$IFOPT WARNING 2-}
  …
{$ENDIF}
```

Defined(<Conditional Identifier>**) | Defined('**<Conditional Identifier String>**')**

Special built-in function that can be used in conditional expressions (**$IF** and **$ELSEIF** directives) and in any expression in normal code. It returns TRUE if <Conditional Identifier> is known as a defined symbol at this point (formerly equivalent to the **IFDEF** directive). The second form accepts '?' and '*' wild card characters in the string to match more than one symbol.

Example:
```
{$IF DEFINED(BOOT) and DEFINED('PIC16*')}
  …
{$ENDIF}
```

Declared(<Identifier>**)**

Special built-in function that can be used in conditional expressions and in any expression in normal code. It returns TRUE if <Identifier> is a Pascal identifier (constant, type, variable, function or procedure name) that has been declared at this point.

Example:
```
{$IF DECLARED(PORTA) or DECLARED(TMR0H)}
  …
{$ENDIF}
```

<u>Remember that:</u>

- Conditional compilation directives are not accepted inside expressions (as any directive).
- Conditional compilation identifiers are local to units**;** they are global if **Defined** in the project's options or **Defined** in the main program before the **USES** keyword.
- If re-**Defined** in units, they become local to the unit after the point of code where they are re-**Defined** (they don't propagate to the main program).
- If **Defined** in project options, they cannot be redefined in code.

3.8 Configuration bits declaration as CONST

- This syntax is maintained for compatibility, since V1.2+ there is the new $CONFIG directive that is more universal (see § 3.7.2).

This syntax is not compatible with processors with a multi-word configuration.

The basic CONFIG declaration in PMP is defined as a **CONST** section pseudo variable definition. This pseudo variable cannot be used as identifier and is not defined in the program.

```
CONST
  CONFIG = <Numeric Expression>;
```

This variable maps directly to the __CONFIG directive of MPAsm™ assembler; declaration accepts any constant expression that returns a 16 bits value, witch is generated as a single value for the assembler.

Example:

```
  CONFIG = _INTRC_OSC_NOCLKOUT AND _WDT_OFF AND _PWRTE_OFF AND _MCLRE_OFF
           AND _CP_OFF AND _BOR_OFF AND _IESO_OFF AND _FCMEN_OFF;
```

Will generate this code in the .asm file:

```
  __CONFIG h'30D4'
```

3.9 Identifiers

PMP identifiers may start with a single underscore character (access to some special declarations in MPAsm™ syntax, such as CONFIG constants), but identifiers starting with two underscores are illegal (reserved for PMP internal use).

PMP identifiers may contain unaccented letters, numeric characters and the underscore character: [‘A’..’Z’, ‘a’..’z’, '0'..'9', ‘_’]. The underscore is always significant.

As in standard Pascal language, PMP identifiers (and any keyword) are not case sensitive.

PMP identifiers are not limited in length, but should be compatible with the assembler that is used (MPAsm™ or other).

Identifiers may also generate internal larger composite identifiers for which the total length may exceed the limits of the assembler (32 characters for MPASM). PMP will use a special squeezing technique to fit within the assembler limit of 32 characters, while trying to maintain enough visual information for debugging.

Examples:

The following identifiers are legal:

```
  _MyVar, TheVar, Var4, The_var
```

But these ones are illegal:

```
  _, __MyVar, 4TheVar, Var4$, The_var%
```

3.10 Constant declaration

PMP supports the declaration of symbols of type constant that may be simple values, strings or arrays.

Constants must be declared in a **CONST** section.

The syntax of a constant declaration is:

<Constant> **=** <Constant expression>**;**

Or:

<Constant>**:** <Simple type> **=** <Constant Expression>**;**

Or

<Constant>**:** <Type Identifier> **=** < Constant Expression>**;**

Or:

<Constant>**: ARRAY[**<Range Expression>**..**<Range Expression>**] OF** <Simple Type> **= (**<Constant Expression>(**,** <Constant Expression>)*)**;**

Where:

<Constant> is a valid identifier name.

<Constant Expression> is a valid numeric, character or string expression that may be computed at this point.

<Range Expression> is a valid numeric constant that may be computed at this time.

<Simple Type> may be **BOOLEAN**, **BYTE**, **SHORTINT**, **WORD**, **LONGWORD** (**DWORD**) or **LONGINT** and also **SINGLE** or **REAL** for PIC16+.

- Simple-typed constants syntax is for convenience only, it forces a range checking; such a variable declaration does not generate initialization code or variable allocation as in TP or Delphi (these strange “variable constants”).

<Type Identifier> is an identifier of a previously defined TYPE. The resulting type should be <Simple Type> or an ARRAY type. String types are not allowed, only direct string literals may be used.

<Value Expression> is a valid numeric constant expression that may be computed at this time. If the value does not match <simple type>, a truncation occurs and a warning is produced.

Expressions may be any valid expression that may be evaluated by the compiler at this point (PMP uses 32 bits arithmetic for computing integer constant expressions, and 64 bits doubles for computing floating point expressions):

```
CONST
  ConstOne = (123 + 4) * $10;
  ConstTwo = ConstOne - 1;
  ConstThree = 'hello ' + 'world'; // string made by concatenation
  ConstThreeCrLf = ConstThree + 13 + 10; // string with CR/LF
  ConstFour = 4;
  ConstFive = PI * 1.234E-3; // defaults to REAL type
  ConstSix: single = PI / 4.0; { force SINGLE precision, but computed as
                                  REAL (due to the expression that starts
                                  with PI that is REAL) }
  MyConstArray: ARRAY[1..ConstFour] OF INTEGER = (1, 1 SHL 1, 4, 2 * 4);
  MyFloatArray: ARRAY[1..ConstFour] OF REAL = (1.0, 1 SHL 1, 4.0, 2 * 4);
TYPE
  tMyConstArray = ARRAY[1..4] OF INTEGER;
CONST
  MyConstArray2: tMyConstArray = (10, 20, 30, 40);
```

PMP is a single pass compiler. Constants are declared prior to their use.

Assembler equivalence of constants:

Internally, all constants are prefixed by the module name (program or unit) as:

<Program or unit name>**.**<Constant name>.

Constants declared in a procedure or function declaration are internally prefixed with the procedure or function name too, as:

<Program or unit name>**.**<Procedure or function name>**.**<Constant name> and they cannot be accessed outside.

- Internal representation of symbols may be squeezed by a special algorithm to fit the 32 characters limitation of MPASM.

3.10.1 Special constants behaviors

The only operator allowed between **STRING** constants is **+**. String construction with implicit concatenation is not allowed (TP / Delphi syntax like 'hello'^M^J'world!'). Anything that is not a string is converted to a single char (numeric 10 is converted to LF character).

Constant strings are implemented as **ARRAY[**0**..SIZEOF(STRING)-**1**] OF BYTE.** First byte at index 0 stores the string length as for a variable string.

Except for strings or arrays, a symbol of type constant does not consume RAM space**;** they are only used by the compiler.

Contrary to rules used by TP or Delphi, a typed constant is not modifiable at runtime (it is not in RAM).

Constant string and arrays are stored in program code space (use of a set of RETLW instructions for PIC10..PIC16, and DATA directives for PIC18+). PMP tries to optimize constant strings or arrays if they are short or indexed by a constant (can generate no code space at all).

Characters of a string may be accessed through an index like for an **ARRAY**:

```
CONST
  MyString = 'Hello ' + 'World!' + 13 + 10;
…
  A := MyString[TheIndex];
```

If a string is only referenced with a constant index (S[constant]) it will not generate a block of code for storing the constant string.

A small enough string move (assignment) is implemented as individual byte moves.

If a constant string is used more than once it will generate only one block of code to store it.

3.10.2 Pseudo SET, IN keyword

A special construction is implemented for bit masks, using the syntax of standard Pascal SET:

[BitA, BitB**]** defines a bit mask where the bit positions BitA and BitB are set.

Special behavior: If an element is a BOOLEAN variable, the bit number of this variable is used.

The standard **IN** keyword may be used to check a bit within any expression result as a pseudo SET.

Example:

```
VAR
  IO_Bit: boolean @PORTC.1;
  VarLong: longint;
  VarByte: byte;

CONST
  BitA = 2; BitB = 4;

  BitMask = [BitA, BitB, IO_Bit]; { Equivalent to [2, 4, 1]:
                                    constant b'00010110' }
…
  VarLong := BitMask;
  VarByte := BitB;

  IF 2 IN BitMask THEN // Evaluated as always True at compile time
    IF 4 IN [1, BitB..31] THEN // Evaluated as always True at compile time
      IF BitA IN VarLong THEN  // True since bit 2 is set in VarLong
        IF VarByte IN VarLong THEN        // True since VarByte=4 and
                                          // bit 4 is set in VarLong
…
```

Since this special construction applies only to simple variables or expressions, it is limited to 32 bits values and may be used in constant declarations or in any expression.

Standard PIC registers and bit values:

During MPAsm™ include file importation, PMP automatically defines all SFR registers as VOLATILE BYTE variables and all bit definitions as constants (as C, IRP, TMR0IE …) so these identifiers have not to be declared and so cannot be redefined by the user code.

3.10.3 System constants and pseudo variables

PMP defines some system constants:

FALSE	Returns 0.
TRUE	Returns 1.
NIL	Returns 0.
MAXINT	Returns **HIGH(**integer**) =** 32767 ($7FFF).
MAXWORD	Returns **HIGH(**word**) =** 65535 ($FFFF).
MAXLONGINT	Returns **HIGH(**longint**) =** 2147483647 ($7FFFFFFF).
MAXLONGWORD	Returns **HIGH(**longword**) =** 4294967295 ($FFFFFFFF).
PI	This constant returns the value of PI, in the maximum FP precision in PMP (REAL format) that gives a 32 bits mantissa plus the sign bit. This is a true constant, not a computed value.

PMP defines some pseudo variables:

FREQUENCY This constant returns the current selected processor frequency in Hz.

VERSION This constant returns the PMP compiler version in a numerical form that may be used in expressions. This is especially useful in $IF conditional defines to compile for specific PMP versions (well, not very useful for now...).

If the version is Major.Minor.Revision, the returned value is:
Major * 100 + Minor * 10 + Revision.

Note: this is not similar to the standard TP or Delphi that defines a conditional symbol like VERxxx.

IDLOC[<Constant Expression>**]** This array returns or sets the defined IDLOC value of index <Constant Expression> that is evaluated as 0..x, depending on the processor. On some processors this pseudo variable is writable. On small processors this pseudo array is not readable: in this case PMP uses values defined at compile time, otherwise the current IDLOC's are read from the code flash memory.

MEMAVAIL This variable (read only) returns the currently available dynamic memory (heap), in bytes. Dynamic memory is implemented for PIC18 and PIC16 enhanced only.

Division / modulo optimization:

One thing that is frustrating / not optimal in standard Pascal is the need to perform two divides to extract the quotient and the remainder of an integer fraction (one DIV and one MOD). In PMP, the remainder of the last divide operation may be available through a pseudo-variable if not optimized out. It is up to you to store it somewhere before the next divide operation:

MOD8: BYTE	This variable returns the last BYTE divide modulo. MOD8 also returns the LSB of the last WORD, INTEGER or LONGINT / LONGWORD divide.
MOD16: WORD	This variable returns the last WORD (unsigned 16 bits) divide modulo. MOD16 also returns the LSW of the last LONGINT / LONGWORD divide.
MOD16S: INTEGER	This variable returns the last INTEGER (signed 16 bits) divide modulo. MOD16S also returns the LSW of the last LONGINT / LONGWORD divide.
MOD32: LONGWORD	This variable returns the last LONGWORD (unsigned 32 bits) divide modulo.
MOD32S: LONGINT	This variable returns the last LONGINT (signed 32 bits) divide modulo.

- Note that MOD8, MOD16, MOD16S, MOD32 and MOD32S share the same memory locations.
- Note that system constants and pseudo constants cannot be redefined locally, these keywords are reserved words.

3.11 Type declaration

Types are limited in PMP; for the current implementation, the allowed type declarations are:

- Fixed length string.
- One dimension array of simple types.
- Simple record definition.
- Dynamic record definition.
- Pointer definition.
- Enumerated type.
- Range type.

<Type Declaration> ::=

TYPE

(<String Type Declaration> |

<One Dimension Array Type Declaration> |

<Record Type Declaration> |

<Dynamic record Type Declaration> |

<Pointer type definition> |

<Enumeration type definition> |

<Range type definition>)*

Where:

<String Type Declaration> ::= <Type Identifier> **= STRING [** <Numeric Constant> **];**

<One Dimension Array Type Declaration> ::= <Type Identifier> **= ARRAY[** <Array Dimension Range> **] OF** <Simple Type>**;**

<Array dimension Range> ::= <Numeric Constant> **..** <Numeric Constant> | <Range type definition>

<Record Type Declaration> ::= <Type Identifier> **= RECORD** <Simple field declaration>* **END;**

<Dynamic record Type Declaration> ::= <Type Identifier> **= RECORD** <Simple field declaration>* <Method declaration>)* **END;**

<Pointer type definition> ::= <Type Identifier> **= ^**(<Simple type> | <Type Identifier>)**;**

<Enumeration type definition> ::= <Type Identifier> **= (** <Identifier> (**,** <Identifier>)* **);**

<Range type definition> ::= <Type Identifier> **=** <Constant expression> .. <Constant expression>**;**

<Method declaration>) ::= <Procedure declaration> | <Function declaration>

Note: Since PMP has no type checking, ranges may contain mixed values. In the example below, the syntax `tMyRange3 = Monday..12;` would be accepted.

Examples:

```
TYPE
  tMyString = STRING[10];
  tMyArray = ARRAY[1..10] OF BYTE;
  tMyRecord =
    RECORD
      X, Y: BYTE;
      B1, B2: BOOLEAN;
    END;
  tMyWordPtr = ^WORD;
  tMyRecordPtr = ^tMyRecord;
  tMyEnum = (Sunday, Monday, Tuesday, Thursday, Friday, Saturday);
  tMyRange1 = 12..24;
  tMyRange2 = Monday..Saturday;

  tMyArray2 = ARRAY[tMyRange] OF BYTE;
```

For an example of a dynamic record, see § 3.11.2.1

3.11.1 Pointer types

Pointer types may be defined forward as in standard Pascal:

```
TYPE
  pMyType = ^tMyType;
  tMyType = BYTE;
```

Restriction: the forward declared pointer must be found in the current **TYPE** block.

3.11.2 Records

Records must be declared in a **TYPE** section prior to be used.
Direct variable declaration as **RECORD** is not allowed.
A PMP record is a simple subset of a standard Pascal record; it cannot be nested and must contain only simple type fields or strings or multi-bits fields (see below).
BOOLEAN fields in a record use only one bit as simple booleans, but they are grouped in one or several consecutive bytes (the programmer must not use code that presumes of a record field position in memory).
Unused bits in bytes that are used for booleans are not reused by PMP for another variable, so it is safe to move a record as a full block of bytes.

Record multi-bits fields:

Introduced in V1.3.13, to match SFR particular multi-bits fields (such as ADCON0.CHS), a new syntax has been defined:

<Multi-bits Field Declaration> ::= <Type Identifier>: BITS [<Numeric Constant>];
Where:
<Number of bits> is an expression that returns a value in the 1..8 range.

Multi-bits fields may be accessed as any other record field. They are assumed to be unsigned.

A multi-bit field cannot cross a BYTE boundary. Also if a multi-bits field is followed by a simple type (as BYTE, INTEGER or whatever), this one cannot cross a byte boundary too.

Memory representation: Individual bits and bit slices are allocated LSB first (from right to left).

A multi-bits declaration is only allowed within a RECORD declaration:

```
TYPE
  tMyRecordBits =
    RECORD
      X: BYTE;       // bits 0..7 of first byte
      B1: BOOLEAN;    // bit 0 of second byte (same as B1: BITS[1])
      BF1: BITS[3]; // bits 1..3 of 2nd byte
      BF2: BITS[4]; // bits 4..7 of 2nd byte
    END;
```

This is **not** allowed:

```
TYPE
  tMyRecordBits =
    RECORD
      X: BYTE;       // bits 0..7 of first byte
      B1: BOOLEAN;    // bit 0 of second byte (same as B1: BITS[1])
      Y: BYTE;       // bits 0..7 of 3rd byte
      BF1: BITS[3]; // bits 1..3 of 2nd byte
      BF2: BITS[4]; // bits 4..7 of 2nd byte
    END;
```

3.11.2.1 Dynamic records

- Dynamic records are introduced in V1.4.9 as an alpha feature that needs to be tested and upgraded further. It is implemented for all processors and do not need a $POINTERS LARGE directive.

What is called a “dynamic record” is a special construction of a standard record that accepts “methods”.

This is not Object Oriented programming but a midway between classic programming and OOP. It was fully integrated in Delphi some years ago and I found that it may be seen as a new way to manipulate data without all the stuff needed by OOP, so the footprint stays quite small in PIC programming.

Example of a dynamic record :

```
TYPE
  tMyDynRecord =
    RECORD
      X, Y: BYTE;
      B1, B2: BOOLEAN;
      procedure Init(iX, iY: BYTE);
    END;
…
procedure tMyDynRecord.Init(iX, iY: BYTE);
  begin
    X := iX;
    Y := iY;
    B1 := true;
    B2 := true;
  end;
```

- A method declaration is only allowed after the fields declaration, mixing fields and methods declarations is forbidden.
- A method's body must be declared in the implementation section.

What dynamic records may do:

Well, first a dynamic record is a record, so it may be used as a record: it may be copied, cleared, used in a WITH statement, passed as a parameter and so on.

Then methods of a dynamic record may manipulate any field of the record without referring to the record itself (implicit WITH SELF clause).

What dynamic records do not have:

- ✗ As this is not OOP, a dynamic record cannot inherit anything from an “ancestor”, so there's no constructors, destructors, virtual methods and any OOP concepts.
- ✗ Compared to Delphi's records, they have no constructors, operators or methods overloading.
- ✗ Dynamic records are not initialized except by the $INIT RAM directive (as any other variable in PMP).

How a dynamic record works:

A dynamic record method has an implicit VAR parameter that is always passed from the caller, called SELF: it is a pointer to the record itself.

To be clear: the method in the example above is implicitly equivalent to:

```
procedure tMyDynRecord.Init(iX, iY: BYTE; var SELF: tMyDynRecord);
  begin
    with SELF do
      begin
        X := iX;
        Y := iY;
        B1 := true;
        B2 := true;
      end;
  end;
```

- Note: SELF is a new reserved word that refers to the record itself within a dynamic record method only.

3.12 Variables declaration

PMP has the following variable types:

- **BYTE**: unsigned byte. This is the most effective format in PMP.
- **CHAR**: unsigned byte (see below).
- **SHORTINT**: integer (8 bits, signed).
- **WORD**: unsigned integer (16 bits, unsigned).
- **INTEGER**: integer (16 bits, signed).
- **LONGINT**: long integer (32 bits, signed).
- **LONGWORD** or **DWORD**: long word (32 bits, unsigned).
- **SINGLE**: floating point value in the standard 32 bits (23 bits mantissa) IEEE format (see FP chapter).
- **REAL**: floating point value in the special proprietary format that PMP uses internally; it is 48 bits wide (6 bytes), with 32 bits mantissa (see FP chapter).
- **BOOLEAN**: boolean, uses only one bit in memory.
- **ARRAY OF BYTE|CHAR|SHORTINT|WORD|INTEGER|LONGINT|LONGWORD|SINGLE|REAL**: one dimension array of simple type.
- **ARRAY OF BOOLEAN**: one dimension array of bits (not very efficient if indexed by a variable).
- **STRING**: variable length array of a maximum of 32 characters (bytes) by default (see $STRINGS directive).
- **STRING [**n**]**: variable length array of a maximum of n characters (bytes), with n in 0..255.
- **RECORD**: set of simple type fields (see below).
- **POINTER**: untyped pointer.
- **Declared TYPE**.
- **SFR**: unsigned byte located at a processor special function register. SFRs are always treated as VOLATILE.

3.12.1 Special behaviors

RECORD variables must use a declared type in a **TYPE** section before to be used. Direct variable declaration as **RECORD** is not allowed.

RECORD types cannot contain a case statement (polymorphic records) and cannot contain nested records (only one level of variables).

CHAR is implemented, but since PMP has low type checking, this is strictly equivalent to a **BYTE** type. Single character literals are always treated as **BYTE** (Except for WRITE/WRITELN statements).

Pointers are implemented but features are limited to a subset of the standard Pascal features in the current version of PMP (future development in the to-do list).

Arrays or **strings** cannot cross a RAM bank boundary, so PMP may flag a memory overflow if one cannot fit in one bank. For PIC18+ arrays and strings may cross a RAM bank boundary.

Arrays are limited to one dimension and 256 bytes, regardless to the type of the elements. Low and high bounds may be defined as wanted. For PIC18+ the maximum size is 64K bytes (if the processor can hold it!).

Strings are limited to 255 characters. High bound may be defined as wanted.

3.12.2 Special considerations about memory allocation

Variables may be allocated either in processor RAM or EEPROM.

Variables in EEPROM (a great PMP feature!) should be used with care since they are time consuming (write is 5 ms typical) and the number of writes is limited.

Reading bit variables in EEPROM is implemented as byte read and masking; writing bit variables in EEPROM is implemented as byte read, masking and write back. This is time consuming.

- **Strings, pointers or CONST/VAR procedure/function parameters cannot be allocated in EEPROM.**

Arrays may be allocated in EEPROM.

A whole EEPROM array can be read or assigned.

An EEPROM VAR array can have initial values.

3.12.3 Some VAR Examples

```
CONST
  AnArrayMax = 2;
VAR
  AnArray: ARRAY[-2 .. AnArrayMax] OF BYTE; { boundaries may be negative }
  {$SPACE EEPROM} // next in EEPROM
  EE_BYTE_1, EE_BYTE_2: BYTE;
  EEArray: ARRAY[1 .. 2] OF BYTE = (100, 200); { EE arrays may have
                                                  initial values }
  {$SPACE RAM} // next in RAM
  MyString: STRING; { defaults to 32 characters wide (see $STRINGS),
                      so it uses 33 bytes }
  MyString2: STRING[4]; { 4 characters string, uses 5 bytes. }

TYPE
  tMyRecord =
    RECORD
      X, Y: BYTE;
      B1, B2: BOOLEAN;
    END;
VAR
  MyRecord: tMyRecord;
```

PMP is a single pass compiler. Variables are declared prior to their use.

Variables declared within procedures or functions are local.

Variables declared in a unit interface section are global to other units and to the main program.

Variables declared in the main program are local to the main program.

By default variables are NOT initialized by PMP; this task is left to the programmer (see CLR built-in procedure). Nevertheless a special directive may be used to force variables initialization at startup, see $INITIALIZE directive.

3.12.4 Banking

Most PIC micro controllers use memory banking to access the internal RAM or registers; this is not too difficult to handle; PMP optimizes banking instructions usage because it knows anything about variable locations since it allocates variables at compile time (not given to the assembler or linker).

3.12.5 Variables internal names (as seen by the assembler)

All variables are prefixed by the module name (program or unit), followed by a dot.

Variables declared in the global sections are internally named as declared, with the module name prefix**;** variables declared within a procedure or function declaration are internally named <Module Name>**.**<Procedure or function name>**.**<Variable name> and cannot be accessed outside.

Examples:

MyUnit**.**MyVar is a variable declared in a global section of the MyUnit unit.

MyUnit**.**MyProc**.**MyVar is a variable declared in the MyProc procedure of the MyUnit unit.

- Internal representation of symbols may be squeezed by a special algorithm to fit the 32 characters limitation of MPASM.

3.12.6 Declaration at an absolute address

Another construction may be used to force declaration at a specific address or to assign an alias to another variable or variable bit:

The variable type may be followed by **ABSOLUTE** <Address> | <Variable> [<OffsetExpression>]

The keyword **@** is a PMP synonym of the **ABSOLUTE** keyword for variables declaration (@ is not standard Pascal).

It is possible to add an offset expression after the aliased variable.

Examples:

```
CONST
  Offset = 2;     // Offset into TheVar
VAR
  IoBit1: BOOLEAN @ PORTA.0;     // bit of I/O on port A bit 0
  CarryBit: BOOLEAN @ STATUS.C; // alias to status carry
  TheVar2Lo: BYTE @ TheVar + (Offset + 0);  // alias into TheVar
  TheVar2Hi: BYTE @ TheVar + (Offset + 1);  // alias into TheVar
```

3.12.7 Declaration as VOLATILE

A variable may be declared as VOLATILE; this means that the optimizer will never optimize this variable due to the knowledge of it's content, as it can change at any time between two Pascal statements.

Example:

```
VAR
 IntCount: VOLATILE BYTE; // Can change in interrupt
```

Warning: VOLATILE does not mean “atomic”: if the variable is of a type greater than one byte, PMP does not care about if one byte may be modified between two PIC instructions. It is up to the programmer to code special constructions to avoid atomic problems.

Note: Most of the processor's standard registers (SFR) are automatically assumed as VOLATILE, along with variables declared as SFR.

3.12.8 Special usage of bit number or reference

PMP's special (not standard Pascal): bits within an identifier may be referenced with their bit number, numerical or constant.

Examples:

```
CONST
  TheBit = 5;
VAR
  ABit: boolean;
…
  Abit := _MyVar.4;
  ABit := TheVar.TheBit;
  ABit := PORTA.0;
```

PMP's special (not standard Pascal): when referenced in a "set" or "dot" syntax, boolean variables return their bit number; this is useful for I/O mappings.

Examples:

```
VAR
  MyInput1: boolean @ PORTA.3;
  MyInput2: boolean @ PORTA.4;

  TRISA := [MyInput1, MyInput2]; // Equivalent to [3, 4]
  TRISA.MyInput1 := TRUE;        // Set TRISA.3 to 1
```

3.13 SFR (Special Function Register) declaration

The SFR type is a special case of the standard **BYTE** type, designing a processor Special Function Register, at an absolute address.

The SFR keyword allows the programmer to declare and manipulate a Special Function Register (SFR) of the micro-controller, if it is not defined in the standard MPAsm™ include file.

To the program, an SFR appears to be an ordinary RAM variable. The syntax for an SFR declaration is:

VAR <Identifier>: **SFR @**<address constant>**;**

Mainly, this is equivalent to:

VAR <Identifier>: **BYTE @**<address constant>**;**

Differences with a simple BYTE variable:

- The variable name is not internally prefixed by the module or procedure/function name, and is always global.
- The variable is automatically assumed as VOLATILE.

The <Identifier> is treated as a variable name, and the <address constant> is the file register address of the particular SFR. User declared SFR variables are always treated as bankable by PMP. The compiler will automatically handle bank switching as needed.

Note: During MPAsm™ include file importation, PMP automatically defines all register definitions as absolute **BYTE** variables (such as INDF, TMR0 …), so these identifiers cannot be redefined in the source code.

It is possible to declare a pointer to an SFR:

VAR <Identifier>: **^SFR;**

- ➢ A user declared SFR is always assumed to be volatile: no read/write optimizations on SFRs.
- ➢ A pointer to SFR or a procedure / function argument of type SFR are also treated as volatile.

3.14 Procedure and function declaration

Procedures and functions are implemented very closely to the standard Pascal language:

Procedure <Identifier >(<Argument list>)**;** (**forward** | **external** (<Filename>)**;**)
(
<**CONST** declaration> |
<**TYPE** declaration> |
<**VAR** declaration>
)*
BEGIN
        (<Statement>)*
END;

Function <Identifier>(<Argument list>): <VAR TYPE>**;**
(
<**CONST** declaration> |
<**TYPE** declaration> |
<**VAR** declaration>
)*
BEGIN
        (<Statement>)*
END;

Constants, types and variables declared in a procedure or function declaration are local and cannot be accessed outside.

Function result type is limited to **BYTE**, **CHAR**, **SHORTINT**, **INTEGER**, **LONGINT**, **LONGWORD**, **BOOLEAN** and **STRING** simple types. For PIC16+ results can be also of type **SINGLE** or **REAL;** it may be referenced as a pseudo variable that is defined with the name of the function (standard Pascal syntax), or as the **RESULT** pseudo variable (Delphi syntax) and is internally declared as:

<function name>**.**RESULT.

RESULT may not be used elsewhere as a symbol name in PMP, even outside of a function (reserved word).

Since PMP is not intended for recursion (even if it is accepted in procedures: use it with care), if the function result name is found in an expression, the current function value is always loaded instead of generating a recursive call.

Procedure and function arguments and local variables consume RAM space**;** they are internally declared as:

<procedure or function name>**.**<argument name>.

Procedure and function local constants do not consume RAM space (except for strings)**;** they are internally declared as <procedure or function name>**.**<constant name>.

Since the local variables and arguments are implemented in static RAM space, not on a stack, recursion should be used with care.

- **Procedures and functions cannot be nested as in standard Pascal.**

3.14.1 Function RETURN statement - NEW! (V1.6.0):

In some Pascal-like languages such as MODULA or OBERON, there's a special statement for functions: **RETURN** <Expression>**;**

It is implemented in PMP as an "extended syntax". If the “Extended syntax” project's option is not active it will produce a compilation error.

The following code:

```
IF <Condition> THEN
  RETURN <Expression>;
```

is strictly equivalent to:

```
IF <Condition> THEN
  BEGIN
    RESULT := <Expression>;
    EXIT;
  END;
```

3.14.2 Side effect with string buffer and function calls

In processors smaller than PIC18, PMP uses only one string temporary buffer for space reduction, there is a special side effect in function calls in string expressions if there is a string expression in the called function; consider the following code:

```
Function Func1: string;
  Begin
     // assume that Var1 contains 13
     Func1 := '-foo' + Var1 + 'bar-';
  End;

StringVar := 'hello' + Func1 + 'world';
```

StringVar will contain '-foo'+13+'bar-'+'world' because 'hello' was pushed in the string buffer and the string buffer has been destroyed in the Func1 function.

Note that an expression such as 'hello'+13+10+'world'+13+10 does not use the string buffer since it is evaluated at compile time. String buffer is used only if the expression contains non-constant sub expressions.

3.14.3 Parameter passing convention

By default, parameters are passed by value, even arrays or records.

An argument may be optionally **CONST**, **VAR** or **OUT**.

As boolean are implemented as single bits, there's no pointers to booleans, so **VAR BOOLEAN** arguments are implemented by the compiler as a copy before and after the procedure or function call. Warning: If the boolean is an I/O or any SFR bit, it will be updated at procedure/function return only.

CONST <Simple type> is always passed by value in current implementation.

Other **CONST**, **VAR** or **OUT** arguments are passed by address in the procedure or function arguments pseudo variables. Addresses are generated as **BYTE** or **WORD**, depending to the processor memory configuration and **$POINTERS** directive.

Argument types must match exactly if they are **CONST**, **VAR** or **OUT**, or must be compatible otherwise.

OUT arguments are a special **VAR** form; PMP may optimize the code, or emit a warning if used in an expression since it knows that the variable is not initialized at entry (future implementation).

Parameters can be of type **SFR**. An SFR is always passed by address, even if declared as CONST. An SFR parameter is always treated as volatile and not optimized.

Parameters can be of type “open array” (see details below); this type of parameter is always passed by address, along with a stealth upper bound parameter.

NEW! (V1.6.0):
A **CONST** parameter may be qualified as **ROMABLE** (not standard Pascal). In this case, PMP generates a special code that may deal either with RAM or ROM pointers. This code will be a bit less efficient but may be useful in some cases (a good example is to pass ROM strings or arrays.

Examples:

```
PROCEDURE Test(X: BYTE; VAR Y: BYTE; VAR Z: BOOLEAN; S: SFR);
  BEGIN
    …
  END;

PROCEDURE Init(CONST ROMABLE T: ARRAY OF BYTE; VAR Dest: STRING);
  VAR I: BYTE;
  BEGIN
    FOR I := low(T) TO high(T) DO SomeProc(T[I], Dest);
  END;

VAR A: BYTE; B: BYTE; C: BOOLEAN;
    S: ^SFR;
…
Test(A + 1, B, C, S^);
```

The call to `Test(A + 1, B, C, S^)` will generate the following sequence:

Compute A+1.
Move result to Test.X,
Move B address to Test.Y,
Move C value to Test.Z,
Move S content to Test.S,
Call Test,
Move Test.Z value to C.

3.14.4 Open array parameters

An open array parameter is a way to pass an array for which the dimension is not known, or to pass arrays with different sizes.

Example:

```
PROCEDURE Test(X: ARRAY OF integer);
  BEGIN
    FOR I := LOW(X) TO HIGH(X) DO
      X[I] := I * I;
  END;

VAR
  X0: ARRAY[1..5] OF integer;
  X1: ARRAY[-2..20] OF integer;
…
Test(X0);
Test(X1);
```

When an array is passed as an open array parameter, its bounds are shifted so that low bound is zero, so the low() pseudo function applied to an open array always returns 0.

An Open array parameter is always passed by address. The open array pseudo variable (X in the example) is allocated as a pointer but occupies also an additional RAM space that is used to pass the actual high bound of the array.

Open arrays may be passed as an argument to another procedure or function.

Open arrays cannot be assigned globally, only elements may be accessed.

3.14.5 Forward procedures and functions

Procedures and functions may be declared as forward (standard Pascal); the forwarded declaration argument list or function result must match the first declaration, or may be omitted.

Example:

```
PROCEDURE ForwardProc2(VAR X: INTEGER; Y: BYTE); FORWARD;

PROCEDURE Proc1;
VAR
  A: INTEGER;
BEGIN
  ForwardProc2(A, 12); // sets A to 12
END;

PROCEDURE ForwardProc2;
BEGIN
  X := Y;
END;
```

3.14.6 External procedures and functions

Procedures and functions may be declared as external:

Syntax 1:

```
PROCEDURE ExternalProc1a; EXTERNAL 'd:\MyProgram\Externals.xyz';
PROCEDURE ExternalProc1b; EXTERNAL 'd:\MyProgram\Externals.xyz';
```

In this syntax, PMP assumes that the two procedures are implemented externally in a separate source file in assembler language. PMP will generate an `include` directive for the assembler, at the end of the module. Several procedures and function may reside in the same include file. No check is made by PMP on the given file name or path that must be fully qualified with its extension. If there is no path, the assembler default include path will be used.

Note that PMP does not generate a separate .asm file for externals; they are included in the current module; they should NOT contain an END assembler directive.

Warning: if declared in a unit with this syntax, the assembler `include` directive is always generated and the parameters consume RAM space. If declared in the main program, the assembler `include` directive is only generated if the procedure or function has been used. Nevertheless the parameters always consume RAM space, even if they are not used.

Syntax 2:

```
PROCEDURE ExternalProc2; EXTERNAL;
```

In this syntax, the procedure is declared external for the assembler and will be solved by the linker. The file must be explicitly given to the linker (add it to the additional files list in the project options).

External procedures and functions may have arguments; they are always allocated in RAM by the compiler, even if they are not used. Assembler instructions may access these variables and global variables as well (see naming convention in chapter 3.14). Note that the parameters are always prefixed with the module name where the external function or procedure is defined.

3.14.6.1 Memory allocation in assembler modules

External procedures and functions may have local storage for variables, but since they are not controlled by the compiler, this may generate troubles.

Memory allocation should not be explicitly addressed to avoid memory overlapping with compiler generated variables, so use MPLAB® UDATA, UDATA_SHR or UDATA_ACS simple directives without addressing, like this MPLINK® will allocate the storage in areas unused by the compiler; since real address is unknown at writing time, bank is also unknown: banksel pseudo instructions should be used.

The **$RESERVED** directive may be used to reserve storage that is forbidden to the compiler.

Also the **$VARIABLES** directive may be used to limit compiler RAM space and manage storage for absolute variables (not recommended).

3.15 Interrupt special procedures

A procedure may be defined with the INTERRUPT qualifier (with no arguments):

```
PROCEDURE InterruptProc; INTERRUPT;
```

And for PIC18+ processors that have 2 levels of interrupts (note that “LOW” is assumed if omitted):

```
PROCEDURE InterruptProc; INTERRUPT; LOW;
PROCEDURE InterruptProc; INTERRUPT; HIGH;
```

This procedure is to be executed in the case of a processor interrupt.

Interrupt procedures may be multiple and in several units or in the main program (see also the **$INTERRUPTS** directive).

The compiler inserts the necessary context save / restore code for common registers and system variables.

According to the **$INTERRUPTS** directive, the code scheme is different:

If **$INTERRUPTS MULTIPLE** (default):

There may be several interrupt procedures (per priority if PIC18). Each interrupt procedure is called in turn. The order is the order of unit compilations. It is the responsibility to the programmer to check interrupt conditions and pertinence at the beginning of each procedure.

Interrupt code generated by the compiler:

```
<Context save>
<Call to the 1st interrupt procedure>
…
<Call to the nth interrupt procedure>
<Context restore>
<RETFIE>
```

If **$INTERRUPTS UNIQUE**:

This option saves one hardware stack level and may be useful for simple treatments. There may be only one interrupt procedure (per priority if PIC18). It is the responsibility to the programmer to check interrupt conditions and pertinence at the beginning of the procedure.

Interrupt code generated by the compiler:

```
<Context save>
<Interrupt procedure body>
<Context restore>
<RETFIE>
```

The context saving is made regarding of the PIC registers and PMP internal variables that are used in the interrupt procedures and **by one level** of called procedures and functions or internal subroutines.

This does not include procedure and functions parameters and internal variables, so they are definitely not reentrant if such variables are defined.

The "one level" scheme is not waterproof; within an interrupt procedure it should be avoided to call procedures and functions that may call another procedures and functions.

Likewise, PMP internal subroutines should be used with care since most of them may be not reentrant and some special functions passing arguments are not saved and would be destroyed.

Using the **$OPTIMIZE SPEED** directive inside interrupt procedure may help, but special features should not be used, this includes (not limited to): FP math, all block moves and some built-in functions and procedures.

Generally PMP warns about possible reentry issue but again, this is not bulletproof.

There is no special limitation in interrupt procedures, local variables, constants and types may be used. Assignments to global variables greater than a single byte or boolean should be used with care since global variable read or writes may be interrupted in the middle of the variable move and the result may be unpredictable (“atomic” problem; this is not PMP or PIC specific).

3.16 Main program block

The main program block is the last defined block of a program, and has the following standard syntax:

```
BEGIN
        (<Statement>)*
END.
```

These statements are executed in serial fashion immediately upon processor reset (after main start-up code and unit initializations, see below). The execution section is terminated with an END keyword, followed by a dot.

- At the end of the execution section, the compiler automatically inserts a jump to the beginning of the main program block, so a WHILE TRUE DO BLOCK; construction is not necessary, a program is an implicit forever loop.

3.17 Unit initialization block

The unit initialization block is an optional section and must be the last defined block of a unit. It has the following standard syntax:

```
INITIALIZATION
        (<Statement>)*
END.
```

These statements are executed once in serial fashion immediately upon processor reset, before execution of the main program block (The order of execution is the order of the units compilation).

The execution section is terminated by the normal unit END keyword, followed by a dot.

4 Statements

4.1 Assignments and expressions

PMP parses expressions containing constants, variables and function calls, and obeys the normal rules of precedence.

PMP has no type checking for simple variables, so any combination of types may be used (possible truncation will produce a warning).

Boolean variables or bits are internally used as single bits but if used in non-boolean expressions they are converted by an implicit **ORD** to byte values of TRUE (1) or FALSE (0).

In non-strict mode, for boolean expressions any value other than zero is treated as TRUE, so this construction is allowed, even for non-boolean variables:

```
IF Variable THEN
```

It will generate the same code than:

```
IF Variable <> 0 THEN
```

Use of simple constants may be optimized by PMP:

A "while <Constant expression evaluated to false> do <block>" will not generate any code.

A "if <Constant expression evaluated to false> then <block>" will not generate any code, if an "else <block>" exists, only this "else <block>" will be generated.

A "while <Constant expression evaluated to true> do <block>" will generate only <block> that loops forever.

This may help for conditional compilation if configuration constants are used (may be used the same way as $IFDEF/$ENDIF blocks).

In PMP two boolean constants are predefined: TRUE = 1 and FALSE = 0. Since there is no type checking on booleans in non-strict mode, either TRUE and FALSE or 0 and 1 may be used for booleans.

The syntax for a variable assignment is:

<Variable> **:=** <Expression>**;**

Where:

<Expression> ::= <Relation> | <Term> (**+** | **-** <Term>)*
<Relation> ::= <BoolExpression> (<BoolOp> <BoolExpression>)
<Term> ::= <Factor> (***** | **DIV** | **/** | **MOD** <Factor>)*
<Factor> ::= (**+** | **-**)* <Constant> | (<Expression>) | <Variable> | <Variable>**.**<bit> | <Function Call>
<Constant> ::= any valid numeric or character constant
<BoolExpression> ::= <Expression> (<relop> <Expression>)
<Relop> ::= **=** | **<>** | **<** | **>** | **<=** | **>=** | **IN**
<Boolop> ::= **AND** | **OR** | **XOR** | **SHL** | **SHR**
<Function Call> ::= <Function Name> (**(**<FunctionArgs>**)**)
<Function Name> ::= <UserFunctionName> | <BuiltinFunctionName>

4.1.1 SHR & SHL "normal" behaviors

As in standard Pascal, SHR and SHL applied to signed numbers implies an automatic cast to unsigned, so shifts are always "logical" - never "arithmetic".
This is Pascal standard. Period.
This explanation is to anticipate discussions on this subject. The C standard says there is no standard for >> and << operators on signed numbers. This is left "implementation dependent" and the consequence is that their behaviors differ between compilers and the subject generates long discussions in forums and discussion lists...

Note that the PMP optimizer engine is smart enough to use "logical shifts" or "arithmetic shifts" when an unsigned or a signed number DIV by a power of two is required. Direct use of SHR will not improve the code efficiency. PMP also uses SHL to multiply by a power of two in some circumstances.

4.1.2 Divide operator /

For processors < PIC16 it can be used instead of the **DIV** keyword since PMP does not deal with floating point on these processors, so it is an alias (Pascal rule overload). This behavior is kept for compatibility with times where there was no floating point in PMP, but it is not recommended since there's source compatibility issues.

For processors >= PIC16, this is the normal floating point divide operator, so the result is always float.

4.1.3 Logical operators

AND, **OR**, **XOR** operators are 8 bits **BYTE / CHAR** / **SHORTINT**, 16 bits **WORD / INTEGER** or 32 bits **LONGWORD** / **LONGINT** wide.

4.1.4 Operand size promotion, signed or unsigned

PMP uses the smallest possible size for evaluating each term in an expression and promotes its internal accumulator size as needed, so mixing types in expressions is not reversible. Possible truncations will produce a warning message. **Possible overflows never produce any warning.**

A typical example is also to compare an INTEGER to a WORD. Since a WORD may be >32767, both are expanded to LONGINT before to compare, so the result is accurate.

Since PMP has no 64 bits integers (even internally), this cannot apply to LONGINT / LONGWORD compares, so such compare is made SIGNED.

For math operators, if one term is signed, the result is signed else it is unsigned.

When needed, PMP may expand to floating point if one term is a FP. Assignment of a FP to an integer variable (in $STRICT OFF mode) or an integer to a FP variable is allowed in PMP (with warning in the FP → integer way); the compiler generates automatically the appropriate conversions. Please note that PMP internal FP routines use REAL format for precision; SINGLE values or variables are always converted to REAL before computing. SINGLE is provided for 32 bits IEEE compatibility and variable storage reduction.

Example:

In:

```
VAR B1, B2, B3, B4: BYTE; W1, W2: WORD;

// possible overflow on byte multiplication if (B1 * B2) > 255
B1 := B1 * B2 * W1 + B3 * W2;
```

The expression is evaluated as:

```
B1 := BYTE(WORD(B1 * B2) * W1 + WORD(B3) * W2);
```

This is not the same as:

In:

```
VAR B1, B2, B3, B4: BYTE; W1, W2: WORD;

// No overflow on byte multiplication
B1 := B1 * W1 * B2 + B3 * W2;
```

Where the expression is evaluated as:

```
B1 := BYTE(WORD(B1) * W1 * WORD(B2) + WORD(B3) * W2);
```

Note: both cases will produce a warning message for byte truncation on B1 assignment.

Be careful to use the highest precision term at a leftmost possible position if intermediate results may overflow.

4.1.5 Bit results and related special behaviors

Relations will generate boolean wide results (FALSE **=** 0, TRUE **=** 1). PMP has a special internal Boolean accumulator / stack, so it can deal with pure Boolean operators.

A means to set/clear/test a bit within a variable also exists, through the "dot notation": the <variable>.<bit> form allows the program to test or assign a single bit in any integer variable. <bit> is a numeric constant value or an identifier that was declared as a numeric constant and <variable> may be of either a variable or constant type. Bit must be between 0 and the maximum bit number of the variable (7..31), where 0 is the LSB. For bit tests, if the bit is set, the factor evaluates as TRUE (1), else it evaluates as FALSE (0).

<Variable>.<bit> := <Expression>

<Expression> is reduced to a **BOOLEAN** nature (anything other than zero is considered as TRUE), the state of <bit> within <variable> will be set to match the outcome of <Expression>.

Usually <bit> may be an integer literal or constant.

PMP allows referring to a boolean variable for <bit>; the code generated will use the bit number of the boolean variable. PMP does not check the pertinence of such construction…

Example:

```
VAR
  Input1: boolean @ PORTC.4;

  TRISC.Input1 := true; // set input mode for PORTC.4
```

Relations will produce bit Booleans; the NOT instruction will invert a bit Boolean. For other types of variables, the NOT operator will invert the whole value so consider the differences:

```
VAR
  Bit1: Boolean;
  Byte1, Byte2: Byte;

  Bit1  := TRUE; // single bit Boolean, set to 1 (Boolean true)
  Byte1 := Bit1; // byte variable set to 1 (Boolean true)
  Byte2 := NOT Byte1; // byte variable set to $FE (NOT $01)
  Byte2 := NOT Bit1; // byte variable set to 0 (NOT TRUE)
```

4.1.6 Bit expressions and statements

If the $CODEGEN 1+ directive is used, the compiler switches to a special bit expressions and statements mode.

All operators or function calls are supported in bit expressions except the negate operator.

The difference in bit expressions, is that the * operator (multiply) is interpreted as an alias of **AND**, the **+** operator (add) as an alias of **OR** and the **/** operator is an alias of **NOT**, so that expressions like ones used in PLCs may be used.

Example:

```
{$CODEGEN 1+ PLC MODE}
Out1 := (/In1 + In2) * In3; // Equivalent to (NOT In1 OR In2) AND In3;
{$CODEGEN 1- END OF PLC MODE}
{ Note:
  operator * has the maximum precedence in expressions evaluations, so: }
Out1 := In1 + In2 * In3 + In4;
{ Is equivalent to: }
Out1 := In1 + (In2 * In3) + In4;
```

4.2 WITH statement

PMP allows a limited form of the standard **WITH** statement:

WITH <Identifier> (**,** <Identifier>)* **DO**
    <Block>

This statement does not generate code in PMP; it is only a programming facility.

Precedence is last to first identifier.

Restrictions:

In the present implementation of PMP, the following forms are not accepted:

WITH <Identifier>**^ DO** …
WITH Identifier **DO** … where <Identifier> is a pointer variable.
WITH <Identifier>**.**<SubField> **DO** ... where <Identifier> is a record type.

4.3 CASE statement

PMP allows the following standard form of the **CASE** statement:

CASE <Expression> **OF**
    (<ConstRange> (**,** <ConstRange>)***:**
        <Block>)*
    (**ELSE** <Block>)
END;

Where:

<ConstRange> ::= <ConstExpr> (**..** <ConstExpr>)
<ConstExpr> is any constant expression that the compiler can evaluate at this point.

Restrictions:

<Expression> and <ConstExpr> should evaluate to a **BYTE** result (else a byte truncation warning will be produced).

CASE constructions are implemented as a set of compare values or as a jump table, depending on the complexity, to optimize speed and memory.

4.4 IF statement

IF <Expression> **THEN** <Block>
(**ELSE** <Block>)

IF statement is optimized for bit Boolean use, but in non-strict mode PMP also accepts a non-boolean expression: any value that is not zero is evaluated as TRUE (1).

Optimizations:

If <Expression> evaluates to a constant value equal to 0 (FALSE), no code is generated for the <Expression> evaluation and for the next block and the ELSE block is generated.

If <Expression> evaluates to a constant value not equal to 0 (TRUE), no code is generated for the <Expression> evaluation and the next block is generated and no code is generated for the ELSE block.

These two cases permit the use of constants for generating code that is always checked, in replacement to $IFDEF constructs that are not.

4.5 ELSEIF statement - NEW! (V1.6.0)

ELSEIF is implemented in some Pascal-Like languages such as MODULA or OBERON (as ELSIF keyword).

It is implemented in PMP as an "extended syntax". If the “Extended syntax” project's option is not active it will produce a compilation error.

IF <Expression> **THEN** <Block>
(**ELSEIF** <Expression> **THEN** <Block>)*
(**ELSE** <Block>)

the number of ELSEIF statements is not limited.

The same behaviors as the IF statement apply; if one of the IF or ELSEIF <Expression> is true, its <Block> is executed and the ELSE <Block> is not executed; else if none of the IF or ELSEIF <Expression> is true, the ELSE <Block> is executed.

4.6 WHILE statement

WHILE <Expression> **DO** <block>**;**

The WHILE statement is optimized for boolean use, but in non-strict mode PMP also accepts a non-boolean expression: any value that is not zero is evaluated as TRUE (1).

Optimizations:

If <Expression> evaluates to a constant value equal to 0 (FALSE), no code is generated for the <Expression> evaluation and for the next block.

If <Expression> evaluates to a constant value not equal to 0 (TRUE), no code is generated for the <Expression> evaluation, the next block is generated and loops forever.

The loop flow may also be controlled via BREAK and CONTINUE statements.

4.7 REPEAT statement

REPEAT <Block> **UNTIL** <Expression>**;**

The REPEAT statement is optimized for boolean use, but in non-strict mode PMP also accepts a non-boolean expression: any value that is not zero is evaluated as TRUE (1).

Optimizations:

If <Expression> evaluates to a value of 0 (FALSE), no code is generated for the <Expression> evaluation and the block does not loop.

If <Expression> evaluates to a value not equal to 0 (TRUE), no code is generated for the <Expression> evaluation and the block loops forever.

The loop flow may also be controlled via BREAK and CONTINUE statements.

4.8 LOOP statement

NEW! (V1.6.0):

LOOP <Block> **END;**

This particular form of a loop exists in some other Pascal-like languages such as MODULA or OBERON.

Formally it is equivalent to a REPEAT <Block> UNTIL FALSE; loop.

It is implemented in PMP as an "extended syntax". If the “Extended syntax” project's option is not active it will produce a compilation error.

The loop flow may be controlled via BREAK and CONTINUE statements.

4.9 FOR statement

FOR <Variable> **:=** <Expression1> **TO | DOWNTO** <Expression2> [**BY** <ConstantExpression>] **DO** <Block>

NEW! (V1.6.0):

The BY clause is implemented in some other “Pascal-like” languages such as MODULA or OBERON, it permits to specify the increment value for <Variable>. If present, the BY value must be a constant and evaluated as positive for a TO loop and negative for a DOWNTO loop.

The BY clause is implemented in PMP as an "extended syntax". If the “Extended syntax” project's option is not active it will produce a compilation error.

- In the following pseudo-codes, <Temp> is a stealth variable that is generated by the compiler.

If <Expression2> does not evaluate to a constant, in the current implementation of PMP the FOR – TO loop is implemented as an equivalent of:

```
<Variable> := <Expression1>;
<Temp> := <Expression2>;
IF <Variable> <= <Temp> THEN
  WHILE TRUE DO
    BEGIN
      <Block>
      IF <Variable> < <Temp> THEN
        INC(<Variable>)
      ELSE
        BREAK;
    END;
```

And as well the FOR – DOWNTO loop:

```
<Variable> := <Expression1>;
<Temp> := <Expression2>;
IF <Variable> >= <Temp> THEN
  WHILE TRUE DO
    BEGIN
      <Block>
      IF <Variable> > <Temp> THEN
        DEC(<Variable>)
      ELSE
        BREAK;
    END;
```

If <Expression2> evaluates to a constant, in the current implementation of PMP, the FOR – TO loop is implemented as an equivalent of:

```
<Variable> := <Expression1>;
IF <Variable> <= <Expression2> THEN
  WHILE TRUE DO
    BEGIN
      <Block>
      IF <Variable> <> <Expression2> THEN
        INC(<Variable>)
      ELSE
        BREAK;
    END;
```

And as well the FOR – DOWNTO loop:

```
<Variable> := <Expression1>;
IF <Variable> >= <Expression2> THEN
  WHILE TRUE DO
    BEGIN
      <Block>
      IF <Variable> <> <Expression2> THEN
        DEC(<Variable>)
      ELSE
        BREAK;
    END;
```

There are some special behaviors from other “standard” Pascal implementations:

- End value is evaluated once at the beginning and stored in a stack temporary variable if needed.
- <Variable> is always set to <Expression1> at first evaluation, unless some cases listed in the optimizations, see below.
- <Variable> content at normal loop termination is equal to <Expression2> if one loop has been executed, or <Variable> still contains <Expression1> if <Expression2> was reached at the first evaluation. Nevertheless, standard Pascal specification states that variable value at the end of a loop is not predictable and should not be used.
- A check is made by PMP to forbid <Variable> assignment inside the loop or passing <Variable> as a VAR argument.

Optimizations:

If <Expression1> and <Expression2> evaluates to the same constant value, the <block> is executed once, no loop code is generated.

If <Expression1> and <Expression2> evaluates to constant values, and <Expression1> is over <Expression2>, no code is generated, even for the loop variable assignment.

4.10 FOR iterator NEW! (V1.6.0)

This particular form of the FOR statement has been introduced in the latest versions of Delphi and in FPC. It is implemented in PMP as "standard Pascal".

FOR <Variable> **IN** <Enumerable> **DO** <Block>

<Enumerable> may be:

- An array variable,
- A string variable or a string literal,
- An enumerated type,
- A simple type variable (BYTE, CHAR, SHORTINT, WORD, INTEGER, LONGWORD, LONGINT); this is a PMP special behavior, not a standard Pascal one, see below.

<Variable> is any simple variable compatible with the <Enumerated> element type.

Unlike a classic FOR loop which may loop with index values, the iterator loops on all element values contained in <Enumerable>: this element value will be written in <Variable>.

Both <Variable> and <Enumerable> are read-only during the loop.

- ➢ In the following pseudo-codes, <Temp> is a stealth variable that is generated by the compiler, containing the index in <Enumerable>.

The iterator behavior will differ a bit according to the type of <Enumerable>:

- For an ARRAY <Variable> will contain all values contained in the array, the loop is implemented as an equivalent of:

```
<Temp> := low(<Enumerable>);
WHILE TRUE DO
  BEGIN
    <Variable> := <Enumerable>[<Temp>];
    <Block>
    IF <Temp> < high(<Enumerable>) THEN
      INC(<Temp>)
    ELSE
      BREAK;
  END;
```

- For a STRING <Variable> will contain all characters of the string; if the string is empty, <Block> will not be executed. The loop is implemented as an equivalent of:

```
<Temp> := 0;
WHILE TRUE DO
  IF <Temp> = length(<Enumerable>) THEN
    BREAK
  ELSE
    BEGIN
      INC(<Temp>)
      <Variable> := <Enumerable>[<Temp>];
      <Block>
    END;
```

- For an enumerated TYPE <variable> will contain all possible values of the type. The loop is implemented as an equivalent of:

```
FOR <Variable> := LOW(<Enumerable>) TO HIGH(<Enumerable>)DO
  <Block>
```

- For a simple type variable <Variable> will contain all bit positions that are 1. The loop is implemented as an equivalent of:

```
FOR <Temp> := 0 TO (SizeOf(<Enumerable>) * 8) - 1 DO
  IF <Temp> IN <Enumerable> THEN
    <Block>
```

4.11 BREAK statement

Inside a **WHILE**, **REPEAT, FOR** or **LOOP** control statement, loop termination may be issued with the **BREAK** keyword:

```
  FOR I := 0 TO 10 DO
    BEGIN
      IF A = 0 THEN
       BREAK; // exit for loop
      // Code not executed if A=0

    END;
```

The **BREAK** statement is equivalent to a **GOTO** to the next instruction after the loop block.

4.12 CONTINUE statement

Inside a **WHILE**, **REPEAT, FOR** or **LOOP** control statement, loop continuation may be used with the **CONTINUE** keyword:

```
  FOR I := 0 TO 10 DO
    BEGIN
      IF A = 0 THEN
        Continue; // next loop iteration
      // Code not executed if A = 0

    END;
```

The **CONTINUE** statement is equivalent to a **GOTO** to the evaluation of the loop control expression.

4.13 ASM statement

An assembler block may be inserted at any point, within or outside of a procedure or function or main program block.

A basic assembler block (LOW, see below) is not analyzed by PMP; its content is passed directly to the assembly file and syntax check is made by the assembler:

```
  ASM
    MOVFF  MyProg.MyFunc.I, MyProg.MyFunc.J ; move variable
    CALL   MyProg.OtherProc
  END;
```

No assumption should be made on the current selected ram bank at the block beginning.

As well PMP assumes that the current selected ram bank is unknown at the end of the block, and that any file register content is unknown too (reset of the optimizer data).

Labels should not interfere with PMP ones (all PMP labels starts with two underscores).

An ASM block may be used to declare things outside of a code block:

```
  ASM
MYCONST EQU   128 ; some constant
MYMACRO MACRO X, Y
…
        ENDM
  END;
```

The ASM block has been upgraded to resolve PMP symbols and translate them in fully qualified internal names.

Due to the possibility of conflicts, the syntax has been modified to accept a parameter:

- ASM(LOW) is a low level assembler block (old behavior), no analysis / translation of symbols.
- ASM(HIGH) (without parameter this is the default) is a high level assembler block, where symbols are automatically translated, along with some reformatting features:

```
  ASM
    MOVFF  I,J; move variable
    CALL   OtherUnitProc
  END;
```

will be translated to:

```
  ASM
    MOVFF  MyProg.MyFunc.I, MyProg.MyFunc.J ; move variable
    CALL   OtherUnit.OtherUnitProc
  END;
```

Special warning for interrupt procedures:

ASM blocks should be used with care within an interrupt procedure, or in a procedure or function called by an interrupt procedure, since PMP has no means to analyze the block to know what are the used registers and thus to know which one to save and restore.

4.14 Implied PIC statements

Despite that PMP tries to not have special registers built-in manipulation functions, some special PIC features must be available for the programmer. PMP supports the following implied statements:

CLRWDT	Clear the PIC's watchdog timer.
RESET	Depending on the processor, generates a software RESET instruction or a jump to address 0.
SLEEP	Put the PIC in sleep state. The processor will wake-up on interrupt or WDT if activated.
NOP or **NOP(**<Constant>**)**	Generates a single or multiple processor's NOP instruction.

4.15 Built-in functions

PMP supports the following standard functions:

ABS(<Expression>**)**
Returns the absolute value of the given expression. If the given expression results to an integer size different from INTEGER or LONGINT, the function returns the expression unchanged (unsigned expression). Else the ABS value is generated as Result := - <Expression> if <Expression> is negative.
<u>Side effect for integer values</u>: if Expression is equal to the minimum value of the expression width, the result is <Expression> (ABS(-32768) result is -32768).
If <Expression> returns a floating point value, the result is in the same type as <Expression>.

CHR(<Identifier_or_numeric>**)**
Returns the character value of <Identifier_or_numeric>, truncated to byte. The result type is CHAR.

DEFINED(<Conditional Identifier>**)** | **DEFINED('**<Conditional Identifier String>**')**
Special built-in function that can be used in conditional expressions (**$IF** and **$ELSEIF** directives) and in any expression in normal code. It returns TRUE if <Conditional Identifier> is **$DEFINE**(d) at this point (equivalent to the **IFDEF** directive). The second form accepts '?' and '*' wild card characters in the string to match more than one symbol.

DECLARED(<Identifier>)
Special built-in function that can be used in conditional expressions and in any expression in normal code. It returns TRUE if <Identifier> is a Pascal identifier (constant, type, variable, function or procedure name) that has been declared at this point.

EEREAD(<Address Expression>**)**
EEREAD(<Address Expression>**, BYTE|CHAR|SHORTINT|WORD|INTEGER|LONGWORD|LONGINT)**
EEREAD(<Address Expression>**,** <Size Expression>**)**
Returns EEPROM byte from a direct address**;** uses standard EEPROM processor registers. If <Address Expression> evaluates to a width greater than **BYTE**, a truncation occurs and a warning is produced. If <Address Expression> is a constant, a bounds check is made regarding processor's implementation.
The second form specifies the read length (number of bytes); in this case the returned value is in the specified size.
The third form may be used to specify the size with a constant expression returning the value 1, 2 or 4. According to this size, the returned value is **BYTE**, **WORD** or **LONGWORD**.

HEX(<Expression>**)**
Returns the hexadecimal ASCII digit equivalent ('0'..'9', 'A'..'F') to the low nibble of <Expression> (a mask for the low nibble is not necessary).

HI(<Expression>**)**
Returns the high byte of an expression truncated to **WORD**.

HIGH(<Identifier>**)**
Returns high bound of an **ARRAY,** enumerated or range type (returns max length for a **STRING** type), or max value of a simple variable or type:
HIGH(boolean**)** returns 1.
HIGH(shortint**)** returns 127 ($7F).
HIGH(byte**)** returns 255 ($FF).
HIGH(word**)** returns MAXWORD = 65,535 ($FFFF).
HIGH(integer**)** returns MAXINT = 32,767 ($7FFF).
HIGH(longint**)** returns MAXLONGINT = 2,147,483,647 ($7FFFFFFF).
HIGH(longword) returns MAXLONGWORD = 4,294,967,295 ($FFFFFFFF).
HIGH(single) returns approximately 3.4E+38 (not standard Pascal).
HIGH(real) returns approximately 6.8E+38 (not standard Pascal).

LENGTH(<Identifier>**)**
Returns the length, in elements, of an **ARRAY** type or a **STRING** type. For an **ARRAY** type the length is the number of elements of the array. For a **STRING** type, the length is the actual length of the string, not the total number of bytes (see **SIZEOF**) or the max number of characters (see **HIGH**).

LO(<Expression>**)**
Returns the low byte of an expression result.

LOW(<Identifier>**)**
Returns low bound of an **ARRAY** type or range type (returns zero for **STRING** type and enumerated types), or min value of a simple integer variable or type, or epsilon (smallest value) for a float variable or type:
LOW(boolean|byte|word**)** returns 0.
LOW(shortint**)** returns -128 ($80).
LOW(integer**)** returns -32,768 ($8000).
LOW(longint**)** returns -2,147,483,648 ($80000000).
LOW(longword) returns 0,
LOW(single) returns approximately 1.5E-45 (not standard Pascal).
LOW(real) returns approximately 1.17E-38 (not standard Pascal).

MUL18(<Variable1>, <Variable2>**)**
Optimized multiply of two 8 bits unsigned variables, giving a 16 bits result. This function has been designed to take advantage of the PIC18 8 bits multiply single instruction. On other processors this instruction is enabled but generates standard multiplication code.

ORD(<Identifier_or_numeric>**)**
Returns the numerical value of <Identifier_or_numeric>. This function is actually implemented for compatibility (simply returns <Identifier_or_numeric> value without modification); the result type is the smallest standard integer type that can hold all values of <Identifier_or_numeric>'s type.

PRED(<Identifier_or_numeric>**)**
Returns the previous numerical value of <Identifier_or_numeric>. If <Identifier_or_numeric> refers to a boolean variable, the result is always FALSE. Range checking is made only if <Identifier_or_numeric> refers to a constant value identifier.

SIZEOF(<Identifier>**)**
Returns the total number of bytes used by a constant string or variable in memory (RAM, EEPROM or CODE for literal strings). For simple boolean types, the function returns 1 and a warning is generated since a bit occupies only one bit. For literal strings, SIZEOF returns the total number of characters plus one, even if the string is not generated in code segment as RETLW sequence, due to optimization.

SUCC(<Identifier_or_numeric>**)**
Returns the next numerical value of <Identifier_or_numeric>. If <Identifier_or_numeric> refers to a boolean variable, the result is always TRUE. Range checking is made only if <Identifier_or_numeric> refers to a constant value identifier.

UPCASE(<Identifier_or_numeric>**)** **NEW! (V1.6.0):**
Returns the uppercase character corresponding to the given parameter. The result type is CHAR.

<u>Cast functions:</u>

BYTE|CHAR|SHORTINT|WORD|INTEGER|LONGWORD|LONGINT (<Expression>**)**
If <Expression> returns an integer type value, cast (truncate or expand) the expression result to the specified format.
If <Expression> returns a floating point value, convert (and truncate) the expression result to the specified format. A truncation warning may occur.

SINGLE|REAL (<Expression>**)**
Convert the expression result to the specified format.

4.16 Built-in procedures

PMP supports the following instructions that modify variables, often more efficiently than assignments with expressions (on 8, 16 or 32 bits variables):

CLR(<Variable>**)**
Clear (fill all with zero) <Variable>. <Variable> may be any variable, simple, ARRAY, STRING or RECORD, in RAM or EEPROM. For strings, only the length byte is cleared. This is an equivalent to a FillChar(<Variable>, sizeof(<Variable>), 0) stupid statement in TP/Delphi.

INC(<Variable>**)** or **INC(**<Variable>**,** <Increment>**)**
Increment <Variable> by one or by <Increment>; <Increment> may be any expression.
If <Variable> is a pointer, it is incremented to point to the next item or <Increment> items forward.

DEC(<Variable>**)** or **DEC(**<Variable>**,** <Decrement>**)**
Decrement <Variable> by one or by <Decrement>**;** <Decrement> may be any expression.
If <Variable> is a pointer, it is decremented to point to the previous item or <Decrement> items backward.

ROL(<Variable>**)** or **ROL(**<Variable>**,** <Constant expression>**)**
Rotate <Variable> left by one position or by <Constant expression> positions, inserting MSB to LSB.

ROR(<Variable>**)** or **ROR(**<Variable>**,** <Constant expression>**)**
Rotate <Variable> right by one position or by <Constant expression> positions, inserting LSB to MSB.

Next, PMP supports the following standard procedures:

ASSIGN(OUTPUT, <Output procedure>]*)
Or:
ASSIGN[**(INPUT,** <Input function>]*)]
Or:
ASSIGN[**(ERROR,** <Error procedure>]*)]

Where:
<Output procedure> may be any procedure name that is defined with one and only one BYTE or CHAR argument.
<Input function> may be any function name that is defined with a BYTE or CHAR result type, with no arguments.
<Error procedure> may be any procedure name that is defined with one and only one BYTE or CHAR argument.

Purpose:
This is the way a user defined console I/O routine may be assigned to the standard console READ, READLN, WRITE and WRITELN built-in procedures.
The ERROR routine is called when there is a unrecoverable runtime error (eg: memory allocation failure). Since program stability is engaged, after the procedure call the MCU is "halted" by an infinite loop (interrupts are not disabled, WDT may trigger). The argument receives the error code as an ASCII digit ($30, ...).

PMP specificities:
By default nothing is assigned to INPUT, OUTPUT and ERROR pseudo files. If a pseudo file is not assigned, the related console I/O calls will do nothing.
Pseudo files cannot be closed. They may be re-assigned (even with NIL).

Example:
```
procedure MyOutput(Ch: char);
…
  assign(output, MyOutput); { This will assign the procedure MyOutput for
                              all further WRITE / WRITELN procedure calls }
```

BAUD(rate**)**

Generates all initializations necessary to use asynchronous only communications at the given baud rate using the TXSTA/RCSTA registers. If the processor has no such registers, an error will occur. The generated code is optimized regarding the current **FREQUENCY** and does not validate interrupts; this is left to the programmer. After this procedure call, all asynchronous communications are ready to use (TRIS register bits, asynchronous mode, TX/RX mode). If the given baud rate cannot be implemented within 1% accuracy, a compiler error will occur. The real baud rate and percent of error is output as comment in the .asm file.

DELAY(microseconds**), DELAY_MS(**milliseconds**), DELAY_NS(**nanoseconds**), DELAY_CY**(cycles**)**

Wait the given number of microseconds, milliseconds, nanoseconds or cycles; delays are implemented as NOP instructions and / or loops, according to the required number of cycles. The code is generated according to the declared processor **FREQUENCY**. Impossible or inaccurate delays will generate warnings. Due to internal mechanisms of PMP, resolution, minimum and maximum delay may be limited according processor frequency. Delays do not use interrupts and so may be non accurate if there is lot of interrupts.
For DELAY_MS and DELAY_CY, the argument may be a WORD sized variable or computed expression; else argument must be a constant expression.
For DELAY_CY minimum value is 32. Note that **NOP**(Count) may be used for small timings.

DISPOSE(<Identifier>**)**

Where:
<Identifier> is a typed pointer variable.

Purpose:
Free the dynamic variable pointed to by the given variable. <Identifier> is any typed pointer variable. The size of the freed memory block depends of the size of the type pointed by <Identifier>.
There is no heavy check for pointer coherency (there is only a check that this is a valid RAM address). The allocated memory should have been allocated by a call to the NEW or GETMEM standard built-in procedures.
This procedure is implemented for PIC18 and PIC16 enhanced mid-range only. See also: PMP dynamic memory allocation (§ 4.20).

EEWRITE(<Address Expression>**,** <Expression>**)**
EEWRITE(<Address Expression>**,** <Expression>**, BYTE|CHAR|SHORTINT|WORD|INTEGER|LONGWORD|LONGINT)**
EEWRITE(<Address Expression>**,** <Expression>**,** <Size Expression>**)**

Write a byte value to EEPROM memory to a direct address**;** it uses standard EEPROM processor registers. If <Address Expression> or <Expression> evaluates to a width greater than byte, a truncation occurs and a warning is produced. If <Address Expression> is a constant, a bounds check is made regarding processor's implementation.
The second form specifies the write length (number of bytes); in that case <Expression> is truncated or expanded to match the specified size.
The third form may be used to specify the size with a constant expression returning the value 1, 2 or 4; in that case <Expression> is truncated or expanded to match the specified size.

EXCLUDE(<Integer Variable>**,** <Bit>**)**

Where:
<Integer Variable> is any integer variable (BYTE to DWORD size).
<Bit> is a bit number constant 0..n applicable to <Integer Variable>.
Purpose:
Reset bit <Bit> from <Integer Variable>. This is equivalent to Delphi's exclude statement that applies to SETs. PMP applies to any integer variable that is assimilated to a SET.

FP_CLR

Purpose:

Initializes floating point flags (FP_FLAGS). These flags are only cleared once at program starting and never reset until FP_CLR is called by the programmer. With this method, if there is an error in a complex FP expression, flags may be tested after the whole expression evaluation, and reset before or after.

FREEMEM(<Identifier>**,** <Size>**)**

Where:

<Identifier> is a typed pointer variable.
<Size> is the allocated byte count.

Purpose:

Free the dynamic variable pointed to by the given variable. <Identifier> is any pointer variable. <Size> should be the same size that was given at allocation time.
There is no heavy check for pointer coherency (there is only a check that this is a valid RAM address).
The allocated memory should have been allocated by a call to the NEW or GETMEM standard built-in procedures.
This procedure is implemented for PIC18 and PIC16 enhanced mid-range only. See also: PMP dynamic memory allocation (§ 4.20).

GETMEM(<Identifier>**,** <Size>**)**

Where:

<Identifier> is a any pointer variable.
<Size> is the requested byte count.

Purpose:

Create a new dynamic variable and initializes a pointer variable to point on it. <Identifier> is any typed pointer variable. <Size> is the requested size of the allocated memory block. If the heap of free memory does not have enough space to allocate the block, the ERROR procedure is called and the processor is halted. The allocated memory should be freed by a call to the FREEMEM standard built-in procedure.
This procedure is implemented for PIC18 and PIC16 enhanced mid-range only. See also: PMP dynamic memory allocation (§ 4.20).

INCLUDE(<Integer Variable>**,** <Bit>**)**

Where:

<Integer Variable> is any integer variable (BYTE to DWORD size).
<Bit> is a bit number constant 0..n applicable to <Integer Variable>.

Purpose:

Set bit <Bit> from <Integer Variable>. This is equivalent to Delphi's include statement that applies to SETs. PMP applies to any integer variable that is assimilated to a SET.

MOVE(<Source>**,** <Dest> (**,** <Size>)**)**

Move one variable to another variable for the optional given size in bytes. If not given, the moved size is the lowest size of <source> and <destination>. If the given <Size> is greater than <Source> size, an error is produced. This procedure cannot overlap outside <Source>.

NEW(<Identifier>**)**

Where:
<Identifier> is a typed pointer variable.

Purpose:
Create a new dynamic variable and initializes a pointer variable to point on it. <Identifier> is any typed pointer variable. The size of the allocated memory block depends of the size of the type pointed by <Identifier>. If the free memory does not have enough space to allocate the bloc, the ERROR procedure is called and the processor is halted. The allocated memory should be freed by a call to the DISPOSE or FREEMEM standard built-in procedures.
This procedure is implemented for PIC18 only and PIC16 enhanced mid-range only. See also: PMP dynamic memory allocation (§ 4.20).

OPTION(<Expression>**)**

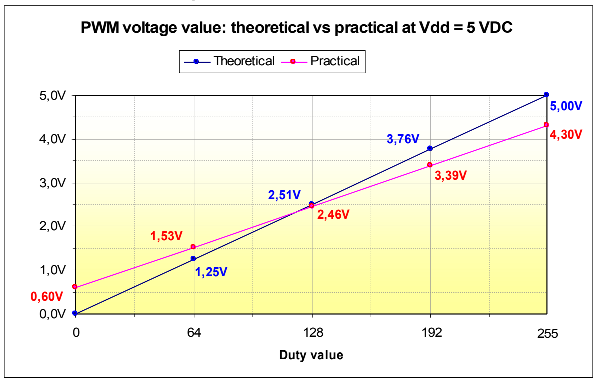
Where:
<Expression> is any byte expression.

Purpose:
Usually PMP does not have special instructions to manipulate registers; to access to the OPTION register the OPTION_REG register should be directly assigned.
Unfortunately, some old and / or small devices do not have OPTION_REG register, but a special instruction OPTION that is a write-only instruction.
The OPTION built-in procedure will automatically map into an OPTION_REG register assignment if such register exists or into an OPTION instruction otherwise.
As said before, OPTION is write-only, so you cannot read back the value, so it should be saved in a global variable if you want to manipulate / mask individual bits.

PWM(<Port>, <PinNumber expression>, <Duty expression>**)**
Or:
PWM(<Identifier>, <Duty expression>**)**

Where:
<Port> is an existing I/O port (PORTA, …),
<PinNumber expression> is a constant expression that gives a 0..7 range,
<Duty expression> is a byte expression that gives a 0..255 range,
<Identifier> is a boolean variable that maps to an I/O pin.

Purpose:
Set the given I/O pin as an output, then send a stream of interleaved 1 and 0 to the specified I/O pin so that the output mean voltage value is theoretically proportional to "Duty" (Vdd * Duty / 255), then set the given I/O pin as an input.
If an RC is wired to the pin, the capacitor will hold the voltage value, so this makes an easy and cheap 8 bits analogical output function. Several PWM statements may be necessary to charge the capacitor, depending on the RC value.
The theoretical voltage value is never achieved, it depends on the pin and on the device; typical 1 is Vdd-0.7V and typical 0 is Vss+0.6V, but the linearity should be acceptable, with some bad excursions at min and max values; experimentation would help…



READ(<Variable [**,** <Variable>]*)**
Or:
READLN[**(**<Variable> [**,** <Variable>]***)**]

Where:
<Variable> may be any simple variable of type BYTE, CHAR, SHORTINT, INTEGER, WORD, LONGINT, LONGWORD, STRING. CHAR variables may be READ individually, they are not treated as BYTE in this case.

Purpose:
This is the standard Pascal READ and READLN procedures that read variables from the current console (see ASSIGN).

PMP specificities:
There is no buffer. If a numeric is followed by a space or tabulation character, this one will be lost for the next READ (see example below).
REAL numbers are not allowed.

Examples:

```
var
  B: byte; C: char; S: string;
…
// Assuming that incoming characters are '*  123  ABCD'
  readln(C, B, S); // This will set C to '*', B to 123 and S to ' ABCD'
                   // (note that one space before ABCD has been lost)
```

STR (<Expression> (: Width Expression>)**,** <String Variable>**)**

Where:
<Expression> may be any expression that may be evaluated by the compiler. REAL numbers are treated but formatting is limited in the current version, the number of decimals cannot be specified and output is always in scientific format (like -1.23456E+23).

<Width Expression> is an optional qualifier (defaults to zero); it is any expression that may be evaluated by the compiler, but result is used as BYTE. If the evaluation of width expression evaluates to a larger value, a byte truncation warning is issued.

Purpose:
This is the standard Pascal STR procedure that converts an expression to a string with optional width qualifier.
If <Width Expression> specifies more than the necessary digits needed to represent <Expression>, spaces are inserted at left to adjust the string to <Width Expression> digits.
If the <String variable> maximum length is less than the number of generated digits, string truncation occurs. Warning: this truncation is not made as in standard Pascal that truncates the string at right and keeps the leftmost digits. PMP truncates the string at left, so that the rightmost digits are kept.

Examples:

```
var
  SX2: string[2]; SX5: string[5];
  W: word; I: integer; B: byte;
…
  B := 3;
  Str(123: B, SX2); // This will set SX2 to '23' regardless to B
  W := $FFFF;
  Str(W, SX5);        // This will set SX5 to '65535'
  I := -123;
  Str(I: 5, SX5);   // This will set SX5 to '- 123'
```

Note: For floating point values, formatting is limited in the current version; the number of decimals cannot be specified and output is always 16 characters wide (or less if specified), in scientific format (like -1.234567890E+23). It is reserved to debug procedures.

SWAP(<Variable>**)**

For **BYTE, CHAR** or **SHORTINT** variables, swaps low and high nibbles. For **INTEGER** or **WORD** variables, swaps low and high bytes. For a **LONGINT** or **LONGWORD** variable, swaps low and high words. Other types are not supported.

TRIS(<Port>, <Mask expression>**)**

Where:

<Port> is an existing I/O port (PORTA, …),
<Mask expression> is a byte expression that gives a BYTE range.

Purpose:

Usually PMP does not have special instructions to manipulate registers; to access to the I/O direction registers the TRISx registers should be directly assigned.
Unfortunately, some old and / or small devices do not have TRISx registers, but a special instruction TRIS that is a write-only instruction.
The TRIS built-in procedure will automatically map into a TRISx register assignment if such register exists or into a TRIS instruction otherwise.
As said before, TRIS is write-only, so you cannot read back the value, so it should be saved in a global variable if you want to manipulate / mask individual bits.

WRITE(<Expression Item> [**,** <Expression Item>]*****)**
Or:
WRITELN[**(**<Expression Item> [**,** <Expression Item>]*****)**]

<Expression Item> ::= <Expression> (: Width Expression>)

Where:

<Expression> may be any expression that may be evaluated by the compiler. Single CHAR may be used (they are not converted to BYTE in this case).
<Width Expression> is an optional qualifier (defaults to zero) witch expression may be evaluated by the compiler, but result is used as BYTE. If the evaluation of width expression evaluates to a larger value, a byte truncation warning is issued.

Purpose:

This is the standard Pascal WRITE and WRITELN procedures that convert expressions to a string and outputs it to the current console (see ASSIGN), with optional width qualifiers.

PMP specificities:

For specificities about string formatting, refer to the **STR** built-in procedure.
At the end WRITELN outputs a pair of CR/LF characters or a simple CR, according to the **$EOL** directive.
REAL numbers are treated but as for the STR procedure, formatting is limited in the current version, the number of decimals cannot be specified and output is always 16 characters wide (or less if specified), in scientific format (like -1.234567890E+23). If size is specified, the string is truncated from the left (specifying 12 characters would output -1.234567890).

Examples:

```
var
  B: byte; C: char
…
  write('This is one:', 1: 3); // This will output 'This is one:  1'
  B := 3;
  write('This is three:', B); // This will output 'This is three:3'
  C := '?';
  write('Somebody there', C: 2); // This will output 'Somebody there  ?'
  writeln; // Output a single CR/LF
```

4.17 Branching statements

Finally, PMP supports the following program branch statements:

<Label>:
GOTO <Label>

Where <Label> must begin with a letter followed by zero or any number of letters or digits or underscore characters.

Forward references are allowed in a PMP program, but all references must be resolved locally by procedure, function or program's end, or else an error will be issued. A GOTO cannot jump outside of the current procedure, function or program.

PMP labels have not to be declared before use (not a standard Pascal Label declaration).

PMP labels translate to an internal representation as for any identifier (prefixed with module name and procedure or function name).

- **In its current implementation PMP does not check for branch to statements that are in FOR / WHILE / REPEAT loops, so this would make unpredictable results, especially in a FOR loop since the loop variable is not initialized:**

```
IF A THEN GOTO Label_A;
FOR I := 1 TO 10 DO
  BEGIN
     // Do Something
    Label_A:
     // Do Something else
  END;
```

Well, everybody would say that this is not a good practice (indeed), but PMP will not flag it…

4.18 Code optimization considerations

PMP tries to strongly optimize the generated code.

Since V1.3 PMP does expression and sub-expressions optimizations before generating code so it produces good code regardless of the complexity or inefficiency of expressions.

Well, in the current real world there are some compromises that may affect the final result, so a minimum help from the programmer is needed.

General rules:
To avoid overflows, put the wider elements to the leftmost position of an expression.
Parenthesis (complex expressions) may use stacks (temporary variables), so if space is an issue, cut complex expressions in several ones.
Do not use signed variables if not absolutely necessary. Signed variables handling costs more memory and MCU cycles.
Do not use floating point variables if not absolutely necessary. Floating point handling costs a lot more memory and MCU cycles.

PMP goes ahead and the future implementations will be more optimized. This is the most difficult goal in writing a compiler…

4.19 Floating point

4.19.1 Overview

- **When floating point (FP) has been introduced, it was for PIC18+ processors only.**
- **Now FP is for PIC16 also!**
- **Using FP on smaller processors will produce compilation errors.**

PMP FP uses routines based on a port of the excellent FP package called PicFloat (from Mike Gore) that may be found on the net. Routines had been rewritten to be adapted to the PMP internal logic.

The internal FP format is an optimized 6 bytes format, giving a pretty 32 bits plus sign mantissa, the full format is:

<SIGN><EXP><MANTISSA> ::= [7..0] [7..0] [31..0]
►9.4 (9 to 10) digits accuracy with an exponent up to +/- 2^127 (+/-1.7E38).

compared to the 23 bits plus sign of the SINGLE 4 bytes IEEE format:

<SIGN><EXP><MANTISSA> ::= [31] [30..23] [22..0]
►7.1 (7 to 8) digits accuracy with an exponent up to +/- 2^127 (+/-1.7E38).

During compilation, PMP internally uses a 64 bits DOUBLE format that is converted to the appropriate format according to generated statements.

The generated code may use a 4 bytes SINGLE format but it systematically converts any format to REAL format when a call to a FP routine is needed (simple + - * / operators or a call to a FP function).

Assignment of a SINGLE variable with a SINGLE expression or constant does not generate a conversion.

PMP does not have an internal representation for NaN (Not a Number) or infinity.
Overflow is trapped as an error and the maximum value is returned instead of infinity.

4.19.2 Supported FP built-in functions

- Note: In all function calls, arguments are always converted to the REAL type before calling the function.

<Angle> represents an angle in radians.

COS(<Angle>**):**
This function returns the cosine of the angle.

ARCCOS(<X>**):**
This function returns the inverse cosine of <X>.

SIN(<Angle>**):**
This function returns the sine of the angle.

ARCSIN(<X>**):**
This function returns the inverse sine of <X>.

TAN(<Angle>**)**
This function returns the tangent of the angle. Tan(X) = Sin(X) / Cos(X).

ARCTAN(<X>**):**
This function returns the inverse tangent of <X>.

EXP(<X>**)**
This function returns the value of e raised to the power of <X> (e^X), where e is the base of the natural logarithms.

LN(<X>**)**
This function returns the natural logarithm (Ln(e) = 1) of <X>.

ROUND(<X>**)**
This function rounds a REAL or SINGLE value to an integer-type value. ROUND returns a LONGINT value that is the value of <X> rounded to the nearest whole number. If <X> is exactly halfway between two whole numbers, the result is always the even number. This method of rounding is often called "Banker's Rounding". If the rounded value of <X> is not within the LONGINT range, the function returns the maximum value (according to the sign) and a run-time FP integer overflow error flag is set (FP_IOV), which can be tested by the programmer.

SQRT(<X>**)**
This function returns the square root of <X>. If <X> <= 0, a runtime FP invalid operation error flag is set (FP_IOP), which can be tested by the programmer.

TRUNC(<X>**)**
This function truncates a real REAL or SINGLE value to an integer-type value. TRUNC returns a LONGINT value that is the value of <X> rounded toward zero. If the truncated value of <X> is not within the LONGINT range, the function returns the maximum value (according to the sign) and a run-time FP integer overflow error flag is set (FP_IOV), which can be tested by the programmer.

SQR(<X>**)**
This function returns the square of <X>. As an exception to the general rule, since SQR(X) is implemented as X*X, the argument may be of any type, integer or FP, and the result is in the same type as the argument (Except for FP expressions, where the result is always in the REAL type). Warning: overflow may occur since <X> is not converted.

POW(<X1>**,** <X2>**)**
This function returns <X1> raised to <X2> ($X1^{X2}$). Implemented as $Y = e^{X1.Ln(X2)}$. If both <X1> and <X2> are zero, or if <X2> is less or equal to zero, a run-time FP invalid operation error flag is set (FP_IOP), which can be tested by the programmer.

4.19.3 FP flags

FP operations may set or test several bits that can be accessed by the programmer as pseudo-booleans:

FP_OVR	Floating point overflow; may be set on any FP operation (also if divide by zero). Read only.
FP_UND	Floating point underflow; may be set on any FP operation. Read only.
FP_IOP	Invalid floating point operation; may be set on any FP operation if argument does not match the function (e.g.: SQRT(-1)). Read only.

- When FP_OVR is set, the offending function returns the REAL maximum value (≈6.8E38), according to the sign.
- When FP_UND is set, the offending function returns the REAL minimum value (≈1.1755E-38), according to the sign.

4.20 Dynamic Memory Allocation

- ➢ This feature is implemented for PIC18 and PIC16 enhanced mid-range only.

4.20.1 Overview

PMP implements standard NEW, DISPOSE, GETMEM, FREEMEM built-in procedures.

If one of these procedures are used, PMP generates dynamic memory allocation structures and procedures. The heap is build from all the available memory blocks that:

- Is not "shared" or "access ram".
- Is not RESERVED.
- Is not BAD (as declared by MPLAB files),
- Has not been used in the program.
- Has a minimum size of 6 bytes.

- ➢ **Heap management is not re-entrant so it should not be called within an interrupt routine.**

4.20.2 How it works

When a NEW or GETMEM statement is invoked, the allocation routine searches a block in this order of priorities:

- ✔ A block that matches exactly the required size.
- ✔ A block that is greater, in this case it will be split as required (see below).
- ✔ After heap defragmentation, the two first steps are tried again; if they don't match again, the ERROR procedure is called and the processor is stopped in an infinite loop.

When DISPOSE or FREEMEM is invoked, the given block is simply linked into the heap; it becomes the first free block.

To minimize the heap fragmentation, the memory blocks are allocated by chunks with a granularity of 4 bytes. The actual block size is the required size plus the memory needed for maintaining a “block size” at the beginning of the block (one byte for PIC16, two bytes for PIC18). If a block is greater than the required size rounded to the next 4 bytes boundary, it is split and the remaining bytes return to the heap.

- ➢ Heap is initialized once at main program start-up.

Heap block memory format:

- <Block size> (one or two bytes)
- Next block address (two bytes)
- n bytes of free memory.

<Block size> is maintained in memory; the pointer returned by NEW or GETMEM is the address next to <Block size>. When the memory block is freed by DISPOSE or FREEMEM, the given pointer is decremented to skip back to <Block Size> and a consistency check is made between the size argument and the current block size; an error is generated if they don't match.

4.20.3 Error treatment

- ➢ **Memory management may generate unrecoverable errors.**

If such an error occurs, the ERROR procedure is called if any has been assigned, then the processor freezes (doing an infinite loop). The interrupts are not disabled and WDT can restart the processor.

Error codes:

$31:	Heap overflow: A block with the required size cannot be allocated.
$32:	Invalid heap block: An attempt to DISPOSE or FREEMEM a block that does not point to a valid RAM address, or the <Block size> field of the block does not match.

5 Libraries

5.1 Overview

PMP libraries are simple free open source software, subjected to the general policy of the PMP license.

Note: A library that is sent to be included in the PMP package becomes a de facto free open source software, free of rights of any nature. The name of the author and his comments are retained unless they go against the rules of PMP or its licensing policy. The content of the library is subject to change, to match PMP's coding rules and future features.

Even if quite limited for now, some of these libraries may need additional information that will be found here.

To be completed...

5.2 Global usage rules

Most of PMP's library units may be parametrized through conditional compilation.

The PMP's ability to pass a $DEFINED symbol from the global project data or from the main program has been used.

If a $DEFINED symbol is not declared, the unit may use a default value.

This is very efficient, because the code is optimized at compile time, rather than passing a lot of parameters to initialization routines and generating all the code for all the cases.

A unit that strongly uses this method is the LCD unit (see below).

5.3 The LCD unit

The LCD unit deals with a HD44780-based compatible LCD display.

Demo project: .\Examples\Test_LCD\Test_LCD.PMP

5.3.1 Supported features

- 4 bits or 8 bits modes.
- 1, 2 or 4 lines (4 lines never tested).
- Any width.
- Character generator functions.
- The character output procedure may be hooked to the standard WRITE/WRITELN procedures by the ASSIGN built-in procedure; it manages CR/LF.

5.3.2 Pin assignments

4 bits mode defaults:

- 0:3 Data bus (low nibble).
- 4 RS pin: Register Select.
- 5 E pin: Enable.
- 6 WR pin: Write Mode (if used, see below).
- 7 Unused.

For 8 bits mode, there's the same assignment for control lines.

Although the PORT assignment may be parametrized, the default pin number assignments cannot be changed via conditional compilation yet; if needed the unit have to be edited.

All control lines must be on the same PORT, all data bits must be on the same port, using the high or the low nibble in 4 bits mode.

5.3.3 Conditional compilation

Parameter / feature	$DEFINE Identifiers	Default
Width	LCD_WIDTH_8, LCD_WIDTH_16, LCD_WIDTH_20, LCD_WIDTH_32;	16 characters.
Number of lines	LCD_LINES_1, LCD_LINES_2, LCD_LINES_4;	2 lines.
Data bus width	LCD_4BITS, LCD_8BITS;	4 bits.
4 bits bus position	LCD_4BITS_UPPER;	Lower bits.
Read mode is possible (WR not permanently grounded)	LCD_READ;	No. Forced as no if PIC10/PIC12.
Use of character generator routine	LCD_GEN;	No.
If LCD_GEN is defined, use of 5x10 characters	LCD_GEN_10;	5x8.
Control port to use	LCD_CNTRL_PORTA, LCD_CNTRL_PORTB, LCD_CNTRL_PORTC, LCD_CNTRL_PORTD;	GPIO for PIC10/12 and PORTB otherwise.
Data port to use	LCD_DATA_PORTA, LCD_DATA_PORTB, LCD_DATA_PORTC, LCD_DATA_PORTD;	GPIO for PIC10/12, otherwise it is PORTB if 4 bits mode, PORTC otherwise.

5.3.4 Constants

Name	**Used as parameter of**	**Comments**
LCD_Width		Nb of characters in width, according to the conditional compilation symbols.
CURSOR_MOVE_LEFT = $00;	LCD_CursorMoveMode	Cursor or text will move to the left at each character write.
CURSOR_MOVE_RIGHT = $02;		Cursor or text will move to the right at each character write (**default**).
CURSOR_MOVE_TEXT = $01;		Text scrolling: the cursor is fixed and the text scrolls.
DISPLAY_TEXT_ON = $04;	LCD_DisplayMode	Show the text (**default**).
DISPLAY_TEXT_OFF = $00;		Hide the text.
DISPLAY_CURSOR_ON = $02;		Show the cursor.
DISPLAY_CURSOR_OFF = $00;		Hide the cursor (**default**).
DISPLAY_CURSOR_FIXED = $00;		Fixed cursor.
DISPLAY_CURSOR_BLINK = $01;		Blinking cursor (**default**).
SHIFT_TEXT = $08;	LCD_ShiftMode	Fixed cursor, text is scrolling
SHIFT_LEFT = $04;		Shift to the left
SHIFT_RIGHT = $00;		Shift to the right (**default**)

5.3.5 Types

LCD_String	Defines a string which size is equal to the display width.

5.3.6 Interface

procedure LCD_Init;	Initialize the LCD system: must me called once before any other operation.
procedure LCD_GotoXY(XPos, YPos: byte);	Cursor addressing (first line / column = 1); does not affect cursor on/off.
procedure LCD_Write_Char(Ch: char);	Write a character to the LCD with line/column management (CR & LF managed too); may be hooked to WRITE/WRITELN, see below.
procedure LCD_Clear;	Clear the display and set the cursor to the home position (1, 1).
function LCD_X: byte;	Return the current X (column) position.
function LCD_Y: byte;	Return the current Y (line) position.
procedure LCD_Home;	Set the cursor to the home position (1, 1); does not affect text.
procedure LCD_WriteString(const romable St: LCD_String);	Write a string to the display. The string may be a string literal (ROM) or a string variable (RAM). Not available for PIC10/PIC12.
procedure LCD_CursorMoveMode(Mode: byte);	Set the cursor move mode (with a combination of the above constants).
procedure LCD_Cursor_On;	Show text and cursor; it is a shortcut for: LCD_DisplayMode(DISPLAY_TEXT_ON + DISPLAY_CURSOR_ON + DISPLAY_CURSOR_BLINK);
procedure LCD_Cursor_Off;	Show the text but hide the cursor; it is a shortcut for: LCD_DisplayMode(DISPLAY_TEXT_ON);
procedure LCD_DisplayMode(Mode: byte);	Set the display move mode (with a combination of the above constants).
procedure LCD_ShiftMode(Mode: byte);	Set the display shift mode (with a combination of the above constants).
procedure LCD_User_Char_5x8(CharCode, Line1, Line2, Line3, Line4, Line5, Line6, Line7, Line8: byte);	Define the CG layout for the given character code, if we are using a 5x8 display.
procedure LCD_User_Char_5x10(CharCode, Line1, Line2, Line3, Line4, Line5, Line6, Line7, Line8, Line9, Line10, Line11: byte);	Defines the CG layout for the given character code, if we are using a 5x10 display.

5.4 The KS107_108 (former GLCD) unit

The KS107_108 unit interfaces a KS107/KS108 type of graphics LCD (128x64).

Demo project: .\Examples\GLCD\LCDGR.PMP

5.4.1 Supported features

- Pixel plot / get state.
- Line, rectangle and circle with outline or fill mode.
- 5x7 and 8x8 fonts.

5.4.2 Conditional compilation

Parameter / feature	**$DEFINE Identifiers**	**Default**
Port to be used for GLCD control, where x = A..E.	GLCD_CTRL_PORTx	PORTB
Port to be used for GLCD data, where x = A..E.	GLCD_DATA_PORTx	PORTD

5.4.3 Pin assignments

Control port defaults:

- 0 CS2: Second display select.
- 1 CS1: First display select.
- 2 RS: Register select.
- 3 WR: Read/Write select.
- 4 E: Enable.
- 5 RST: Reset

Note: bits 6 and 7 are unused and left as inputs.

5.4.4 Constants

Name	**Comments**
GLCD_FONT_WIDTH	Wrapper to the selected font width, currently 5 or 8.
GLCD_FONT_HEIGHT	Wrapper to the selected font height, currently 7 or 8.
GLCD_FONT_SPACING	Wrapper to the selected font spacing, currently 1.
GLCD_FONT_MIN_CHAR	Wrapper to the selected font first implemented character code.
GLCD_FONT_MAX_CHAR	Wrapper to the selected font last implemented character code.
GLCD_CS1	Chip #1 identifier for low level functions.
GLCD_CS2	Chip #2 identifier for low level functions.
GLCD_SET_TOP	Low level command: Specify the first RAM line at the top (see GLCD_Cmd procedure).
GLCD_SET_COLUMN	Low level command: Set the column address (see GLCD_Cmd procedure).
GLCD_SET_PAGE	Low level command: Set the page address (see GLCD_Cmd procedure).
GLCD_ON	Low level command: Turn the display on (see GLCD_Cmd procedure).
GLCD_OFF	Low level command: Turn the display off (see GLCD_Cmd procedure).

5.4.5 Types

tGLCD_InitMode = (glcdOFF, glcdON);	{ 0 (False) = LCD left off after init, 1 (True) = LCD left on after init }
tGLCD_FillMode = (glcdEMPTY, glcdFILLED);	{ 0 (False) = fill empty (black), 1 (True) = filled (WHITE) }
tGLCD_PlotMode = (glcdHIDE, glcdSHOW);	{ 0 (False) = hide pixel (black), 1 (True) = show pixel (WHITE) }
tGLCD_RegisterMode = (glcdINST, glcdDATA);	{ 0 (False) = Instruction register, 1 (True) = data register }
tGLCD_RW = (glcdWRITE, glcdREAD);	{ 0 (False) = write mode, 1 (True) = read mode }
tGLCD_String	{ Widest string usable for GLCD Text }

5.4.6 Variables

None.

5.4.7 Interface

Low level functions:	
procedure GLCD_Init(Mode: tGLCD_InitMode);	Initialize the GLCD system; Mode permits to let the display ON or OFF after initialization.
procedure GLCD_Cmd(Cmd: byte);	Low level procedure: Send a simple command to both chips. The Cmd parameter is one of the low level command values added with an optional data value: GLCD_SET_TOP: Specify the first RAM line at the top 0..63 GLCD_SET_COLUMN: Set the column address 0..63 GLCD_SET_PAGE: Set the page address 0..7 GLCD_ON: Turn the display on (no other data) GLCD_OFF: Turn the display off (no other data)

procedure GLCD_WriteByte(
Mode: tGLCD_RegisterMode;
Chips, Data: byte);
	Low level procedure: write a raw byte to the designed register on the designed chip(s).
	The Chips parameter may be a combination of GLCD_CS1 and GLCD_CS2.

function GLCD_ReadByte(Mode:
tGLCD_RegisterMode;
Chip: byte): byte;
	Low level procedure: read a raw byte from the designed register on the designed chip.
	The Chips parameter may be GLCD_CS1 or GLCD_CS2.

High level functions:	
procedure FillScreen(State: tGLCD_FillMode);	Fill the screen black or white.

procedure PlotPixel(X, Y: byte;
State: tGLCD_PlotMode);
	Plot a pixel, black or white.

function GetPixel(X, Y: byte): boolean;	Get a pixel state, black or white.

procedure PlotData(X, Y, Data: byte;
State: tGLCD_PlotMode);
	Plot a whole data byte, black or white.

procedure Line(X1, Y1, X2, Y2: byte;
State: tGLCD_PlotMode);
	Draw a line, black or white.

procedure Rectangle(X1, Y1, X2, Y2: byte;
FILL: tGLCD_FillMode;
State: tGLCD_PlotMode);
	Draw a rectangle, filled or outlined, black or white.

procedure Circle(X, Y, Radius: byte;
FILL: tGLCD_FillMode;
State: tGLCD_PlotMode);
	Draw a circle, filled or outlined, black or white.

procedure Text(X, Y: byte;
const St: tGLCD_String;
State: tGLCD_PlotMode);
	Draw a text, black or white.

5.5 The SERIAL unit

The SERIAL unit interfaces simple HW asynchronous interface (USART), without interrupts.

It does not support a second HW port.

Demo project: None.

5.5.1 Supported features

- Single character I/O.
- The character input procedure may be hooked to the standard READ/READLN procedures by the ASSIGN built-in procedure.
- The character output procedure may be hooked to the standard WRITE/WRITELN procedures by the ASSIGN built-in procedure..

5.5.2 Conditional compilation

Parameter / feature	**$DEFINE Identifiers**	**Default**
Generate the serial character input routine	USE_SERIALPORT_INPUT	No.
Generate the serial character output routine	USE_SERIALPORT_OUTPUT	No.
Generate the serial string output routine	USE_SERIALPORT_OUTPUTSTRING	No. If yes, the serial output character routine is also generated.

5.5.3 Constants

None.

5.5.4 Types

SerialString	Defines a string which may be used as SerialPort_OutputString parameter.

5.5.5 Variables

Name	**Size**	**Comments**
InErrorCode	byte	Holds the last error codes if the serial input routine is used.

5.5.6 Interface

function SerialPort_InputReady: boolean;	Returns true if a character has been received on the serial port.
function SerialPort_Input: byte;	Return the received byte from the serial port. Return the receive error bits 1 (overrun) & 2 (framing) errors in the InErrorCode global variable. Warning: It waits if there is no char ready.
procedure SerialPort_Output(Ch: byte);	Write a character to the LCD with line/column management (CR & LF managed too); may be hooked to WRITE/WRITELN, see below.
procedure SerialPort_OutputString(const romable S: SerialString);	Write a string to the serial port. The string may be a string literal (ROM) or a string variable (RAM).

5.6 *The A2D unit*

The A2D unit is a wrapper to the Analog To Digital features.

It does not support interrupt-based acquisition.

Demo project: None.

5.6.1 Supported features

- Full initialization of the A/D system.
- 8 or 10 bits formats.
- Acquisition delay setting.

5.6.2 Conditional compilation

None. Any A2D mode may be adjusted at run time.

5.6.3 Constants

None.

5.6.4 Types

tA2D_Channels	Defines the format of the A/D channels mask (BYTE or WORD). The channels mask maps directly to ANSEL/ANSELH or equivalent registers or to ADCON1.PCFG for PIC18.
tA2D_Mode = (adm8bits, adm10bits);	Defines an acquisition result format, 8 or 10 bits. The result of the acquisition is always right justified in a byte (8 bits) or a word (10 bits).
tA2D_ACQT	Defines the acquisition time possibilities. For PIC18 it maps the TAD code, for others it is the nb of microseconds to wait after a channel change before to start the conversion (BYTE). For PIC18, they are in the form adtTAD_n where n is the nb of required TAD. The type may vary according the processor's possibilities.
tA2D_Clock	Defines the A/D clock possibilities. They may be of the form adcFOSCn where n is the Fosc divider, or adcFrc for RC mode.
tA2D_Vref_Mode	Defines the Vref mode possibilities. They may be of the form advPmMm, whre Pm is the Vref+ mode and Mm is the Vref- mode. Common mode is advVddVss, or advVdd for the simplest devices.

5.6.5 Variables

None.

5.6.6 Interface

A2D_Init(A2D_Channels: tA2D_Channels;
A2D_Acqt: tA2D_ACQT;
A2D_Clock: tA2D_Clock;
A2D_Mode: tA2D_Mode;
A2D_Vref_Mode: tA2D_Vref_Mode);
	Initialize the A2D system and turns on the A/D converter (ADON).

function A2D_Get(Channel: byte): word;	Perform an acquisition and conversion and return the result, right justified. The return format is always WORD, regardless of the initialized format (8 or 10 bits). If 8 bits is used, the result may be casted by the LO() pseudo function.

6 Compiler messages

6.1 Error messages

During compilation, PMP may produce error messages; these messages indicate where the error occurs. PMP tries to continue the compilation after errors and may flag other errors, in some circumstances errors are induced by the previous ones.

List of error messages:

Message	**Additional comment**
User error: <info>	Error by $UERROR directive
Illegal command line: <info>	
File create or write error: <info>	
Undefined identifier: <info>	
Duplicated identifier: <info>	
Type mismatch: <info>	The requested statement cannot be applied to the given identifier because of it's format or type.
Not implemented: <info>	This functionality is not yet implemented in the current version of PMP.
Unable to allocate unbanked ram area for variable: <info>	PMP needs unbanked variables for it's internal use. In this case, there is not enough free unbanked memory; try to simplify your code.
Unable to allocate ram area for variable: <info>	There is not enough free RAM memory to allocate the variable.
Unable to allocate EEPROM area for variable: <info>	There is not enough free EEPROM memory to allocate the variable.
Out of processor ram space: <info>	
Unresolved forward label: <info>	
Unresolved forward reference: <info>	
Divide by zero	
Illegal string operation	The given operator does not apply with strings.
Index out of bounds	
Bit number out of bounds	The bit number does not exist for the given variable.
Constant out of bounds	
Syntax error in include file: <info>	Bad MPAsm™ include file.
File not found: <info>	
Bad __BADRAM in include file: <info>	Bad MPAsm™ include file.
Bad delay value: µs	This duration cannot be generated in µs.
Exit not allowed here	
Break not allowed here	
Continue not allowed here	
Interrupt procedure cannot be called	
Range mismatch	
Declaration differs from previous one: <info>	
Interrupt procedure cannot have parameters	
Illegal function return type	
Illegal type declaration	
SFR declaration may be used only for one variable	Declare SFRs one by one.
Illegal type	
Array too large	Dimensions exceed the PMP maximum values.
Only one variable can be declared based absolute	Declare absolute variables one by one.
Record not allowed	A record is not allowed here.
Initialized EEPROM records not allowed	An EEPROM record cannot have initial values.
CONFIG syntax error	
CONFIG cannot be defined here	
Processor cannot be redefined at this point	
Pointers cannot be redefined at this point	
This directive is not allowed in units	
Illegal processor frequency	
Illegal variables start address	
Illegal variables top address	
Illegal EEPROM start address	
Illegal EEPROM top address	
Out of processor EEPROM space: <info>	
Bad identifier	
Illegal directive	
Unterminated conditional compilation block	
<info> expected	<info> is what was expected here.
Interrupt procedure cannot be declared in interface section	

Message	Additional comment
External procedure / function cannot be declared in interface section	
Forward procedure / function cannot be declared in interface section	
Unit not compiled: <info>	An unit declared in the uses section cannot be compiled.
Processor type mismatch: <info>	
Processor frequency mismatch: <info>	
Processor should be defined before	
Frequency should be defined before	
Illegal string relation	
Unit circular reference: <info>	
File circular reference: <info>	
Impossible baud rate at current processor speed	
Internal error	THIS IS A PMP COMPILER BLOW'UP: PLEASE REPORT ERROR TO THE AUTHOR.
This function is not allowed here (reserved to $IF conditional block)	
Cannot assign to constant	
Not implemented for this processor	
This device has no EEPROM area	Attempt to switch variables to EEPROM area or to use EEREAD or EEWRITE on a device that has no EEPROM.
Cannot redefine a reserved word	
Program / unit name mismatches file name	For PMP versions >= 1.1, program and unit names must match the file name.
Internal boolean stack overflow; expression is too complex	PMP uses a Boolean stack that is limited to one byte so this permits only 7 levels of parenthesis (explicit or implicit, one operator uses one level).
This IDLOC has no defined value	The referred IDLOC has not been defined before this point of code.
Illegal I/O channel	The I/O channel given in an ASSIGN statement shoud be INPUT, OUTPUT or ERROR only.
Illegal I/O channel assignment: expecting procedure(char), function:char or NIL	The given procedure for an ASSIGN(output, <proc>) or ASSIGN(error, <proc>) should have only one byte or char parameter . The given function for an ASSIGN(input, <func>) should have no parameter and a result type of CHAR or BYTE. Else an assignment of nil is expected.
Cannot redefine a globally defined symbol	An attempt is made to redefine ($DEFINE or $UNDEF) a conditional compilation symbol that is global (defined in project's properties or in the main program if in a unit).
MODx pseudo variable cannot be used here due to lack of previous DIV or optimization	
Reserved word; identifier expected	A reserved word is used where an identifier is expected.

6.2 *Warning messages*

During compilation, PMP may produce warning messages; these messages indicate where the warning occurs. A warning is not an error; it indicates that PMP has flagged a condition that may be optimized by the programmer.
Warnings may be enabled or disabled globally or by number (see compiler directives). The warning number is issued in the message.

List of warning messages:

#	Message	Additional comment
0	User message	Warning by $UWARNING directive
1	Boolean truncation	Boolean will be set with the 1 LSB only.
2	Byte truncation	Current size is greater than byte, only the 8 LSB will be stored.
3	Integer truncation	Current size is greater than integer, only the 16 LSB will be stored.
4	Word truncation	Current size is greater than word, only the 16 LSB will be stored.
5		Obsolete
6	String truncation	Destination max length is lower than the current size; only the destination max length characters will be stored.
7	Cannot generate accurate delay with this processor frequency	
8	Cannot achieve 1 µs delay with this processor frequency	
9	Cannot achieve 10 µs delay with this processor frequency	
10	Boolean occupies one bit so sizeof is not relevant	Sizeof is meaningless for Booleans (sizeof is the number of bytes).
11	Duplicated bit	The bit appears more than once in the list.
12	Converted to byte	Expression has been converted to BYTE wide value.
13	µs delay is limited	
14	Case items overlap	The given range overlaps another range.
15	Implemented as array of byte	Constant Boolean arrays are implemented as byte arrays (one byte per element).
16	Ram top address change not allowed here; ignored	
17	EEPROM top address change not allowed here; ignored	
18	Result is always zero	The constant expression optimizer as determined that this expression is always 0.
19	EEPROM variables cannot be local variables; ignored	
20	EEPROM variables cannot be arrays; ignored	
21	EEPROM variables cannot be absolute; ignored	
22	Unit not accurate, should be recompiled	obsolete
23	Internal routine call from ISR; possible reentrance conflict	The code uses PMP internal subroutine call that may be used in the main program; it may imply an unpredictable behavior.
24	Inaccurate baud rate	The requested value cannot match exactly due to processor limitations with the current frequency.
25	Small string buffer allocated: n bytes	Due to memory limitations, string buffer has been limited to n bytes.
26	The selected processor has a fixed frequency; ignored	
27	Fixed frequency; the project's default frequency has been overridden	
28	Deprecated or not applicable directive; ignored	
29	SFR is read only	Attempt to write to an SFR that is tagged as read only in MPLAB files.
30	SFR bit is read only	Attempt to write to an SFR bit that is tagged as read only in MPLAB files.
31	SFR bit is write only	Attempt to read an SFR bit that is tagged as write only in MPLAB files; result value is meaningless.
32	This SFR bit is undefined	This identifier is not valid bit name for the related SFR.
33	This bit is undefined for this SFR	This identifier is not a bit name of the related SFR.
34	Variable xxx is defined but never used	The given variable or parameter had been declared but never used in the program.
35	CHAR, BYTE or WORD cannot be negative; sub-expression is always false	
36	Low heap size	The remaining free memory allocated to heap gives less than 64 bytes.
37	SHORTINT truncation	Current size is greater than SHORTINT, only the 8 LSB will be stored.

7 PMP development issues

7.1 Known limitations

PMP was primarily designed for small PIC devices, and I have not checked it for all devices, so some problems may be encountered. Feel free to report any problem that will be fixed in future versions.

PMP currently uses the Microchip MPAsm™ and MPLink™ functionalities; it generates relocatable .asm code, with one section per procedure, function or internal subroutine, so the linker should fit all the code in available processor code pages. Code and identifier names limitations are MPAsm™ and MPLink™ limitations (for example the incredible 32 characters truncation for identifiers).

The processors.cfg file may help to customize some specific processors features.

Pointers and records are not yet fully supported. PMP supports only a subset of the standard Pascal features.

7.2 To do list

(Not in priority order) – Feel free to comment!

1. Optimize RAM usage for procedures and functions arguments or local variables.
2. Full implementation of records (more accepted types, nesting ...).
3. Implement dot notation for specifying external unit symbols. In progress.
4. Build a version that integrates in Microchip IDE (they do not seem to be very happy to give specifications). *In progress but in standby state: I had some good contacts with Microchip support, but for now it is not enough...*
5. Continue to tune-up the expression optimizer; there is always something to do…
6. Inline procedures and functions.
7. Procedure and function overloading?
8. Add a directive to optionally activate the strong checking of types as Pascal does. PMP is C-Like on this (what a shame!). In progress. NEW! Partially done since V1.4.10.
9. Evaluate GPUTILS support. In progress. NEW! Alpha support introduced in V1.4.7.
10. Evaluate using a optional stack for parameters passing and local variables for FSR-aware processors; PMP does not use any stack currently. This is memory consuming but really faster than using a stack.
11. And more…

7.3 Not to do list

1. PMP will never (well, who knows?) include an internal assembler, linker or simulator. I feel that MPLAB® suite is a great product (and free), so presently I don't want to spend time to make a product that will be less efficient, for sure.
2. OO programming (objects, classes ...). I think that OO programming complexity and the implied memory management is not a good idea for small micro controllers. Let's stay basic! (way of speaking, naturally!)

7.4 Limited support

PMP has its own Internet site: www.pmpcomp.fr; suggestions and bug reports may be posted on the forum or sent by email to: Philippe Paternotte philippe.paternotte@pmpcomp.fr.

7.5 About the author

My name is Philippe Paternotte, I was born in 1958 and I am french, living near Paris (feel free to report language errors).
I'm a Senior Software Engineer specialized in Factory Computing and more generally speaking in Factory Automation.
I'm a Pascal and Assembler programmer since the ol' 80's, author of many >100K lines applications and some >200K ones.
I'm using PICs since 2000.
I love fishing, Metallica and good wines.

7.6 Development tools and contributions

At the beginning PMP was written in Delphi 6.0, then the code has been translated to Delphi 2006.

It strongly uses classes and some JEDI Visual Components Library: http://jvcl.sourceforge.net/ or http://homepages.borland.com/jedi/jvcl.

Main lexical parser engine was built with a subset of TP Lex: http://www.musikwissenschaft.uni-mainz.de/~ag/tply.

PMP IDE in the windows version is build around the SynEdit package: http://synedit.sourceforge.net.

Some optimization techniques were found on the PIC mine of code web site: http://www.piclist.com/techref/microchip/routines.htm.

Floating Point routines are adapted from the PicFloat libraries from Myke Gore.

This documentation is written with Sun's Open Office 3.2.

7.7 Revision history

V1.6.0 - 2011-05-31 PPA:

Fixed: Always read volatile 16 variables LSB first and write them MSB first to manage PIC18 TMRx 16 bits mode.
Fixed: Unexpected syntax error when an argument of WRITE std procedure is a function call with parameters and with a char or string result type.
Added: In Extended Syntax mode, an optional BY clause to the FOR statement to specify the loop variable increment (MODULA/OBERON-like syntax).
Added: In Extended Syntax mode, an optional ELSIF clause to the IF statement (MODULA/OBERON-like syntax). Also added $ELSIF as an alias of $ELSEIF that stays for compatibility.
Added: In Extended Syntax mode, a "LOOP ... END" block equivalent to a "REPEAT ... UNTIL FALSE;" block (MODULA/OBERON-like syntax).
Added: In Extended Syntax mode, a "RETURN <Expression>" statement for functions that generates an optimized "RESULT := <Expression>; EXIT;" block.
Added: A "FOR <Loop Variable> IN <Enumerable type or variable> DO <Block>" iterator construction (now standard in DELPHI/FPC).
Fixed: Unexpected "undefined forward reference" error on a non-forward LABEL, due to an early modification side effect.
Changed: For PIC18, the default procedure interrupt level is now HIGH.
Added: For PIC18, a new attribute for interrupt procedures: FAST may be used to optimize interrupts when there's only one interrupt level. FAST mode is the default when there's only one interrupt procedure.
Fixed: With $INTERRUPTS UNIQUE the interrupt vector did not be generated if the interrupt procedure was in a unit.
Fixed: Bad generated code for STR() routine for bytes; bad conversion of some numbers; PIC18 processors.
Added: UPCASE built-in function.
Fixed: Bad generated code for MOVE procedure if the source parameter is a ROM identifier; all processors.
Fixed: Bad generated code for a by-value parameter if the passed value is in ROM (initialized ROM arrays); all processors.
Fixed: Bad dependency check between units when a unit is added to a uses list, avoiding rebuild in some circumstances and generating linker errors.
Fixed: Unexpected undefined symbol with local variables with a procedure/composite composite name larger than 32 characters (failure of the MPASM limit workaround).
Added: Support of ROM array parameter passing (CONST ROMABLE, see the new manual).
Fixed: Various formatting problems in ASM blocks.
Fixed: Bad side effect on STR() built-in procedure that generated unexpected errors.
Fixed: Bad dependency check that produced unnecessary unit rebuilds.
Changed: Symbols table management / rebuild has been optimized so that the compiler is a bit faster.
Fixed: In an ASM block double quoted strings was altered due to the Pascal parser that does not recognize them as strings.
Fixed: In an ASM block assembler hex format nnH was altered due to the Pascal parser that does not recognize them as hex numbers.
Updated: processors.cfg to include 16F630 & 16F676 specifics.
Fixed: Bad error message position in some circumstances; PIC16ENH & PIC18 processors.
Fixed: FP divide code generation was broken due to a bad optimization; all processors.

V1.5.4 - 2010-12-18 PPA:
Fixed: Side effect of new optimization: Bad generated code for $B- AND / OR expressions; all processors.
Fixed: Side effect of new optimization: Possible banking problem in MUL16/DIV16 MUL32/DIV32; all processors.
Fixed: Bad generated code if a function call is not the leftmost operand in an expression in some circumstances; all processors.
Fixed: Bad generated code if a function call is within a FOR loop in some circumstances; all processors.
Fixed: processors.cfg was not complete for PIC16F5? processors, generating bad banking instructions.
Fixed: Side effect of new optimization: bad parameter passing if only one parameter and if it may be evaluated as boolean (0/1); all processors.
Added: A "save as defaults" functionality in the project's options dialog to define options to be used for a new project.
Changed: IDE search/replace function preselects "selected text only" if something is selected in the current window.
Fixed: PIC12F629/675 false stack overflow warnings.
Added: $INIT COMPARATORS to initialize comparator-aware pins to all digital (comparators off) - CMCON in small devices.
Fixed: Removed unnecessary banksel instructions; some small devices (PIC10, PIC12).
Fixed: Compilation error in disabled conditional compilation blocks with a $VECTORS directive for another processor range.
Added: The version values mapping to ID locations option is now functional.

V1.5.3 - 2010-11-16 PPA:
Fixed: A ShortInt was not accepted as a FOR loop variable!!!
Fixed: Bad generated code for mixed signed/unsigned DIV in some circumstances; all processors.
Fixed: Bad generated code for mixed 8/16/32 bits operators - operand promotion in some circumstances; old bug, all processors.
Fixed: Big regression bug; if there's an IF statement that evaluates to FALSE at compile time, the whole current procedure code is not generated after this point.
Added: After too much side effects / regression errors, the enhancement of the validation system is in progress, with more new automatic checks based on Delphi code comparisons.

V1.5.2 - 2010-11-14 PPA:
Fixed: If $INTERRUPT UNIQUE, context saving was not always complete; all processors.
Fixed: New ASM enhanced features produces syntax errors with decimal notation .nnnn and d'nnnn'.
Fixed: Bad generated code due to a wrong pseudo stack usage / optimization when using slices. Visible impact on A2D unit; all processors.
Fixed: Bad generated code with CASE jump table in some circumstances. PIC18 processors.
Added: Enumerated types that had only two values are now generated as booleans to save memory.
Added: Now booleans may be declared absolute to another boolean; this applies to "boolean" enumerated variables too.
Fixed: Bad generated code for array read with a byte index in some circumstances; PIC18 processors.
Fixed: Side effect of new optimization: Warning nnn directive was not applied locally.
Fixed: Side effect of new optimization: Defining a String constant by concatenation was rejected.
Fixed: IDE file was not always focused after loading.

V1.5.1 - 2010-11-10 PPA:
Fixed: FP package was corrupted for PIC16.

V1.5.0 - 2010-11-08 PPA:
Modified: STATUS is preserved at startup so special bits may be tested by program.
Added: $PUSH/$POP directives for enhanced management of local compiler options.
Added: $SCRIPT directive to override normal linker script name.
Added: $OPTIMIZE_PARAMS directive to switch ON/OFF the procedure/functions parameters. WARNING: Possible compatibility issue with pure EXTERNAL procedures and functions.
Fixed: Bad generated code (old bug) in IN internal subroutines; all processors.
Fixed: IDE bad behavior with "Reopen" menu for recent files.
Fixed: Bad file loaded if the required source also exists in the "current" windows directory (std FileSearch behavior...).
Added: Better assembler block: now identifiers are automatically translated to internal format.
Added: Now VAR arrays may be in EEPROM, with optional initial values. EEPROM VAR arrays cannot be passed as CONST or VAR parameters.
Fixed: Syntax error after a label inside a repeat-until loop.
Modified: Better analysis of interrupt context save: called procedures/functions are now analyzed at full nesting deep.
Added: Open array parameters.

!For previous versions, see Changes.txt in the doc subdirectory.

V1.0.0 – 2006-01-14 PPA: Genesis and dark ages: IDE was a stone & burin version of EDLIN - Initial release of PMP, never distributed.

8 Index

9 User annotations

This page is left blank for user annotations.