Machine Translated by Google

PicPas 0.8.7

Compiler for Microcontrollers 8 bit PIC

USER MANUAL

By: Tito Hinostroza

Modified: 06/12/2018

Machine Translated by Google

1 Notes on PicPas

PicPas is an open source, cross-platform, cross-compiler for 8-bit PIC microcontrollers (the 10F, 12F, 16F and enhanced 16F series). The Serie 18F, not yet supported.

Works with the Pascal/Modula-2 programming language in one version adapted to handle devices with reduced memory. Only has implemented, some basic language features have been added some new ones, typical of bit management, which is very common in 8-bit microcontrollers.

PicPas includes an IDE and a simulator, both integrated into the compiler in the same application. These modules are designed to work together, not regardless.

PicPas is developed using the Object Pascal language, the Free compiler Pascal and the Lazarus development environment.

PicPas uses the common components that come in the standard distribution of Lazarus and some additional libraries. These are:

* SynFacilUtils: https://github.com/t-edson/SynFacilUtils
* MisUtils: https://github.com/t-edson/MisUtils
* MiConfig: https://github.com/t-edson/MiConfig
* PicUtils: https://github.com/t-edson/PicUtils
* Xpres: https://github.com/t-edson/t-Xpres
* UtilsGrilla: https://github.com/t-edson/UtilsGrilla
* ogEditGraf: https://github.com/t-edson/ogEditGraf

Machine Translated by Google

2 INTRODUCTION

2.1 Compiler Features

- Uses the Pascal language, in a simplified version, and with features of Modula-2.
- It is a fairly fast compiler. Most operations are done in memory and compiling 1000 lines of code should take no more than 100 milliseconds on a typical computer.
- Directly generate the *.HEX file, without the need for libraries, frameworks or additional programs.
- The generated code is quite optimized, at the level of the best commercial compilers.
- The compiler is cross-platform, as is the IDE. There are compiled versions for Windows, Linux and Mac.
- The compiler is integrated into the IDE. Not provided by separate.
- Supports the insertion of assembly code, directly into the source code.
- Allows you to define hardware characteristics (this is how the various supported microcontroller models are defined) through the use of directives.

2.2 Characteristics of IDE

- Supports multiple editing windows.
- Includes editors with syntax highlighting, code folding, and word and block highlighting, for the Pascal and ASM languages.
- Autocompletion is included, with code templates for structures, IF, REPEAT, WHILE, …
- It has no dependencies on external programs.
- Allows you to see the assembly code and the use of resources.
- Allows you to configure themes.
- Allows code navigation.
- It detects syntax errors in real time.
- It is highly configurable.
- Includes translations into languages: English, Spanish, German, French, Russian and Ukrainian.
- Includes a code debugger and real-time simulator.

2.3 Compiler limitations

- The language only handles the data types Bit, Boolean, Byte, Char, Word and DWord.
- Only some basic operations are implemented for Word and DWord type.

Machine Translated by Google

- **Recursion is not supported, due to resource limitations in the low and mid-range devices. •**
Floating point arithmetic is not included.
- **Support for arrangements or registrations is not included.**

Machine Translated by Google

2.4 Installation

The program was designed to not require installation. All you have to do is copy and unzip the folder that contains the program files.

The name of the executable file varies, depending on the platform:

• PicPas-win32.exe for Windows-x86 •
PicPas-win64.exe for Windows-x64 •
PicPas-linux for Linux •
PicPas-Mac.dmg for Mac

PicPas-win32 will also work on 64-bit Windows platforms.

Not all versions include executables for these three platforms (Windows, Linux and Mac). To generate an executable for a missing platform, you must first generate the executable from the source code, using the Lazarus programming environment https://www.lazarus-ide.org/.

To run the application correctly, you only need the executable and some folders and configuration files.

The folders used are:

/temp -> It is the folder where the editor's temporary files are created, when they are not assigned a particular name. /
units -> It is the folder where the PicPas libraries (units) should be.

/devices10 -> It is the folder where the libraries (units) that must be
They define all supported PIC models, from the low range (Baseline Core).
/devices16 -> It is the folder where the libraries (units) that
They define all supported PIC models, from the mid-range Core.
/devices17 -> It is the folder where the libraries (units) that define all the supported PIC models of the Enhanced Mid-range should be.

/syntax -> It is the folder where the syntax definition files for the editors are stored. /samples -> It is
the folder where some example programs are saved. /themes -> It is the folder where
the color themes for the app are saved.
SDI.

Additionally, to save the configuration options, the <name>.xml file is used, where “name” is the name of the executable file, according to the platform.

Machine Translated by Google

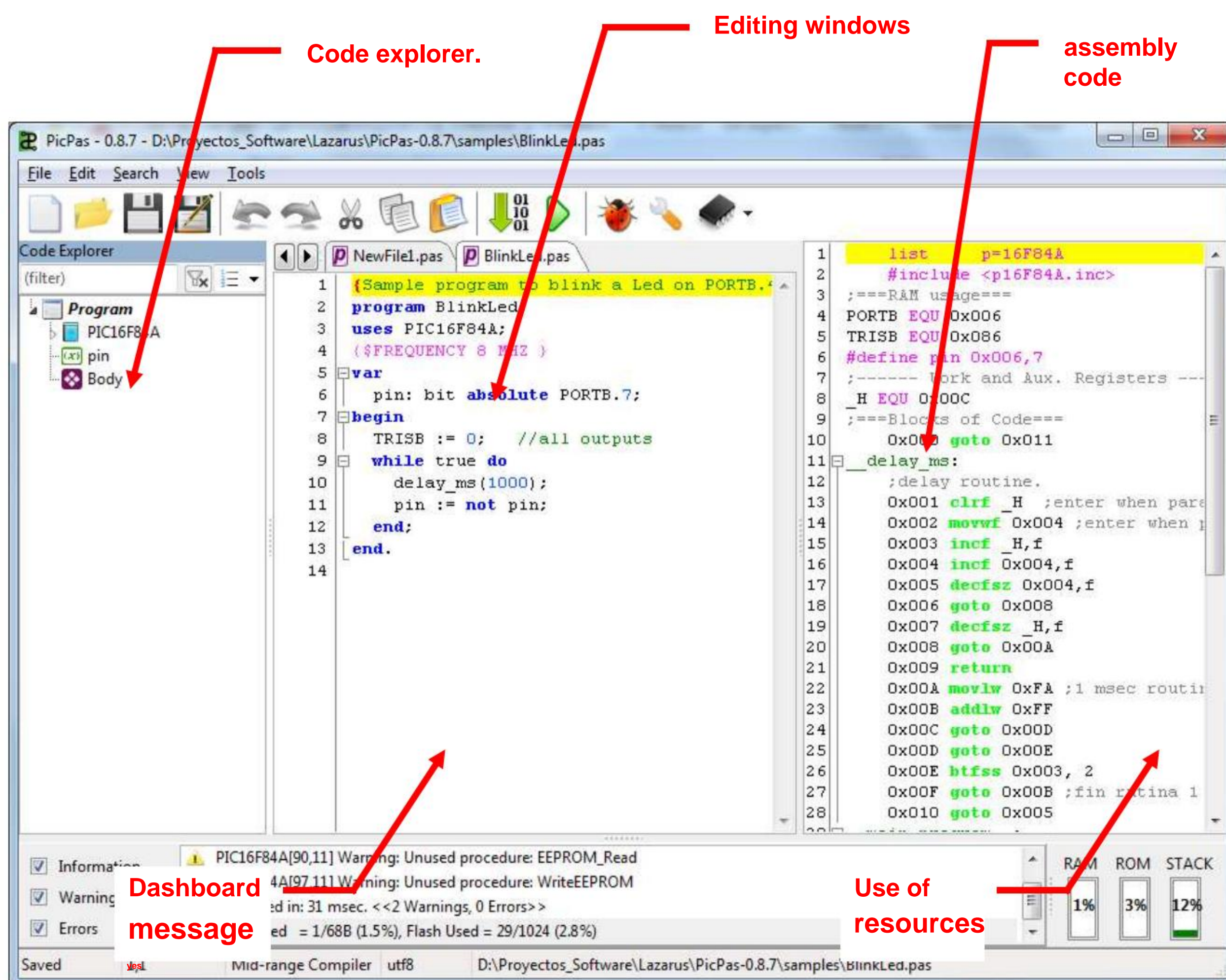
By default, source files have the extension *.pas.

Machine Translated by Google

3 THE INTERFACE

PicPas has a visual interface that is similar on various platforms. To this interface graphics, which allows us to create programs, edit them and compile them, is what We call it IDE (Integrated Development Environment).

The following image shows the parts of the main window:



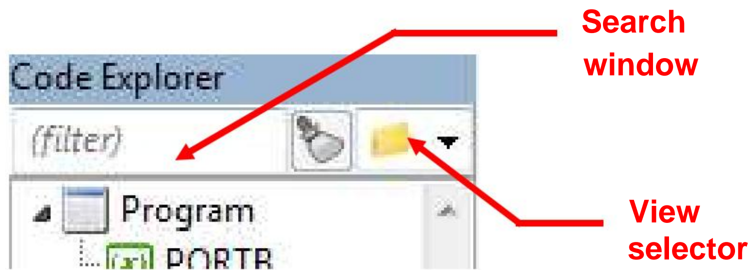
3.1 Code Explorer

It is located on the left side of the interface. This panel shows the program fountain in a tree structure.

The various elements of the program, such as constants, variables or procedures are represented graphically in this panel.

You can search within this panel, using the search box. top search:

Machine Translated by Google

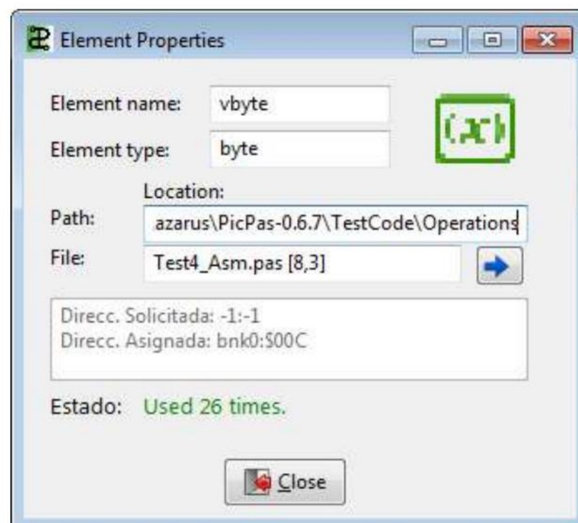


You can also select between two view modes using the control on the right side.

The view modes are:

• Grouped elements. •
Elements in the order of declaration (Not grouped).
• Disk Folders and Directories (Windows version only).

The code explorer also allows you to view important information, such as the number of times a variable is used or the physical address assigned to a variable, using the properties box, when an element is selected, in the code explorer:



You can also use the code explorer to find, in the code, the location where a procedure or variable is declared, using the context menu, or by double-clicking on the element.

3.2 Compiler Selection

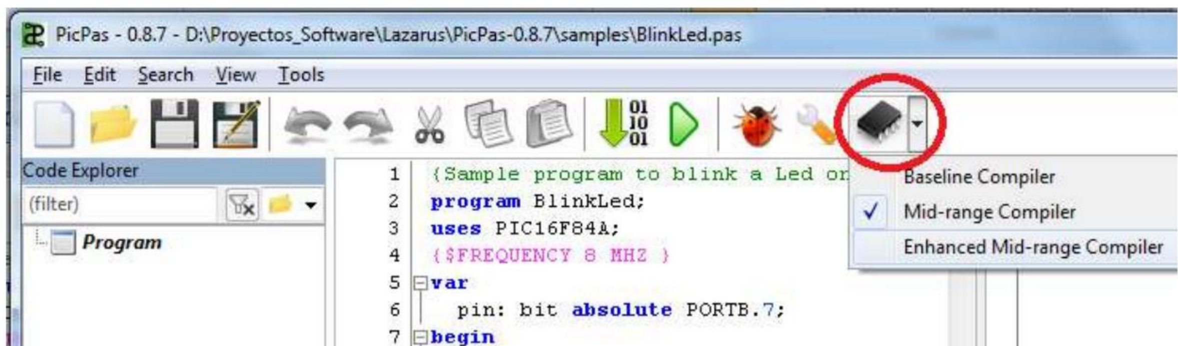
PicPas works internally with various compilers, each one for a specific family of PICs.

The compilers currently available in PicPas are:

Machine Translated by Google

CODE COMPILER	DESCRIPTION
Baseline Compiler	PIC10 Compiler for low-end microcontrollers PIC (Baseline) that have 12-bit instructions. In this group are the models PIC10F200, PIC10F222, PIC12F508, PIC16F54 and others.
Mid-range Compiler	PIC16 Compiler for the mid-range of PIC microcontrollers (Mid-range) that have 14-bit instructions. In this group are the models PIC12F675, PIC16F84, PIC16F877, PIC16F628 and others.
Enhanced Mid-range Compiler	PIC17 Compiler for enhanced midrange PIC (Enhanced Mid-range) microcontrollers that have 14-bit instructions. In this group are the models PIC12F1840, PIC16F1454, PIC16F1619 and others.

To select the working compiler, you must use the toolbar. tools:



At any time you can consult the current compiler by viewing the bar of State:

It is necessary to choose the correct compiler so that the program compiles correctly. appropriate form.

For example, the following program uses the PIC16F84A that belongs to the family "Mid-Range":

```
program BlinkLed;
use PIC16F84A;
```

Machine Translated by Google


```
{$FREQUENCY 8 MHZ }
var
   pin: bit absolute PORTB.7; begin pin :=
not pin;
      end.
```

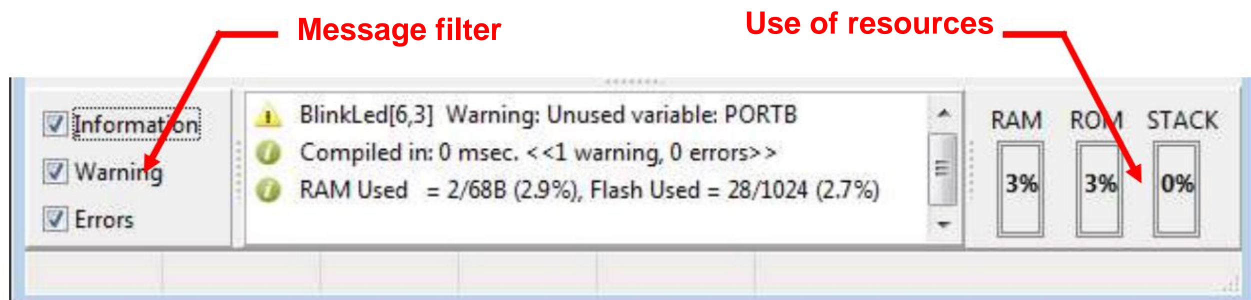
If you tried to compile this program by selecting the "Baseline" compiler, you would get an error message indicating that the PIC16F84A unit cannot be found, since the compiler will look for the PIC16F84A.pas unit in a path where only the units supported by the compiler are found. .

To see the units supported by each compiler, you can see the folders:

/devices10 -> "Baseline" compiler units. /
devices16 -> "Mid-range" compiler units. /
devices17 -> "Enhanced Mid-range" compiler units.

3.3 Message panel

The message panel is located at the bottom of the main window, and displays information about the build process.



These messages show three types of messages:

1. Information Messages.
2. Warning Messages.
3. Error messages.

Information messages are useful data generated by the compiler, such as the compilation time, or the amount of memory used.

Warning messages indicate some aspects in the source code that the compiler detects as dangerous, but which are not errors. However, they must be taken into account, to prevent logic errors or to improve the code. These errors can be, for example: "Variable not used".

Machine Translated by Google

Error messages are generated when the compiler detects syntax errors or lack of resources in the program. Errors stop the compilation process and must be corrected to get the program to compile.

Messages can be filtered using the controls on the left side of the page. message board.

The right side shows the microcontroller resource usage for which is being compiled.

There are 3 parameters that are shown:

1. **RAM memory usage.- Indicates how much of the device's total RAM memory is being used by the program. Either through the program's own variables or through internal variables that the compiler uses. The memory space in all RAM banks is taken into account.**
2. **Use of ROM memory (or Flash EEPROM). Indicates how much of the total program memory is being occupied. All available space is considered, across all memory pages.**
3. **Use of the stack (STACK). Indicates how many levels of the call stack are being used by the program. The use of the stack is due to the number of nested routines that the program has. For the low range it is 2 levels and 8 levels for the mid range.**

Machine Translated by Google

3.4 Edit Window

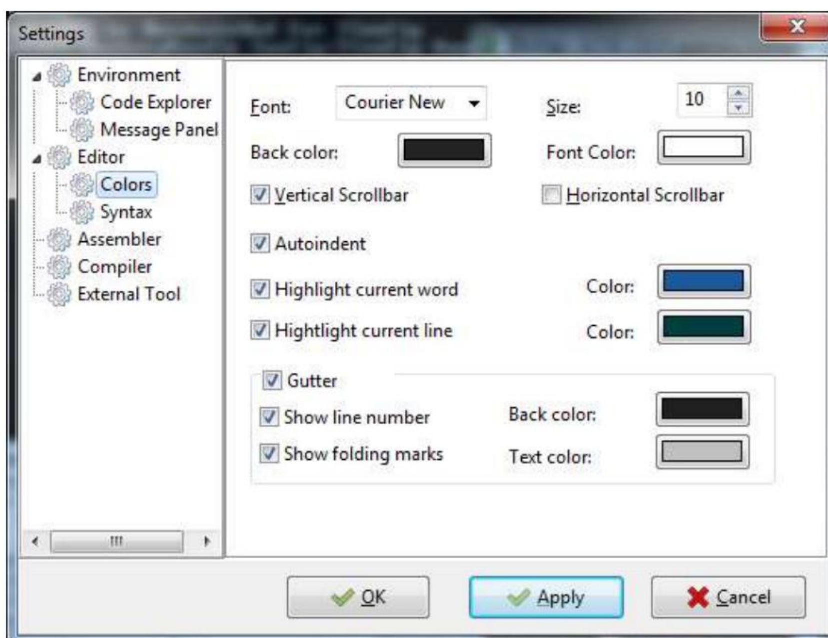
This window is where the source code or program is written. It is an editor with special functions to make writing code easier.

It is intended to work with Pascal files, but can edit other files as well. file types (such as *.C or *.ASM files), with limited options.

The editor features are:

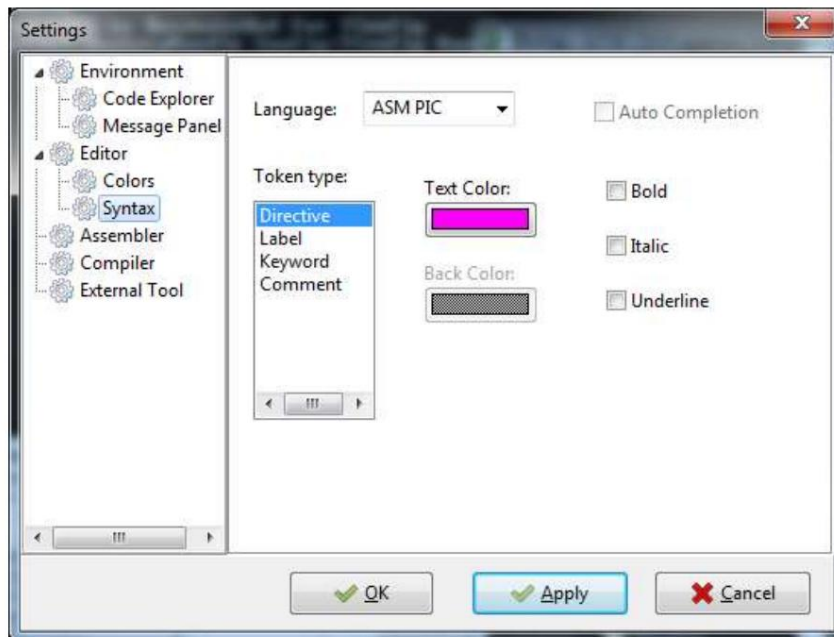
**• Line numbering. •
Undo, redo. •
Highlighting parentheses, braces or brackets.
• Highlighting of the current
word. • Highlighting of the
current line. • Syntax
highlighting. • Code
folding • Context menu and code
autocompletion. • Selection and editing
in column mode. • Editing with
multiple cursors. • Location of variables, constants and procedures.**

Some of the editor's features can be activated or deactivated from the configuration window:



It is also possible to configure the colors and attributes of the editor's lexical elements, using the configuration window:

Machine Translated by Google



The editing window supports multiple text windows, as tabs.

To move between tabs, you can use the combinations:

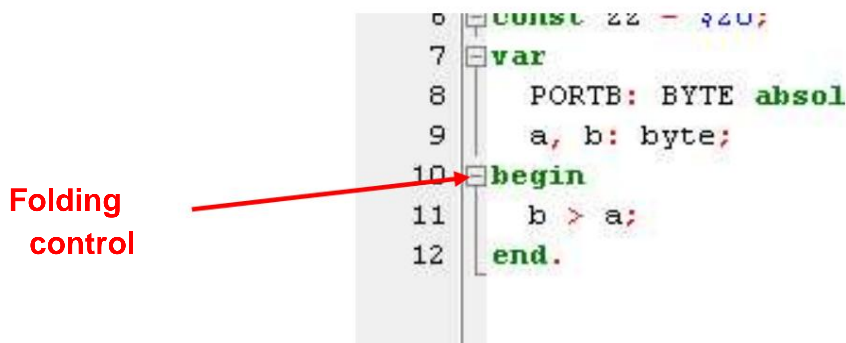
<Ctrl>+<Tab> to move to the next tab.
<Ctrl>+<Tab>+<Shift>, to go to the previous window.

To close the current editing window, you can use the tab's context menu or use the combination:

<Alt> + F4

3.4.1 Code folding

The code folding option is controlled through the right sidebar, where some boxes with the "+" or "-" symbol appear.



These boxes allow you to expand or collapse blocks of code.

However, it can also be folded and unfolded using keyboard shortcuts:

Machine Translated by Google

SHORTCUT	FUNCTION
<Alt>+<Shift>+ "+"	Displays the code at the current position.
<Alt>+<Shift>+ "-"	Folds the code to the current position.
<Alt>+<Shift>+ "0"	Displays the code of the entire editor.
<Alt>+<Shift>+ "1"	Folds the code throughout the editor.

3.4.2 Multiple Cursors

The IDE editors support the management of multiple cursors to edit at various points in the code.

```
 7     TRISB : BYTE absolute
 8     pin: bit absolute POI
 9   begin
10     TRISB := 0;     //all
11     while true do begin
12       delay_ms(1000);
13       pin := not pin;
14     end;
```

Normally the editor will only work with a cursor. To activate additional cursors, hold down the <Shift> and <Ctrl> keys and click somewhere else in the text. In the same way, more cursors can be added, which will perform the same editing action.

To exit multiple cursor mode, simply click the mouse at any position in the text.

Multiple cursors also allow editing of contiguous lines, in the same column.

```
 4  program BlinkLed;
 5  var
 6    PORTB : BYTE absolute $0
 7    TRISB : BYTE absolute $8
 8    pin: bit absolute PORTB.
 9  begin
10    TRISB := 0;    //all outp
```

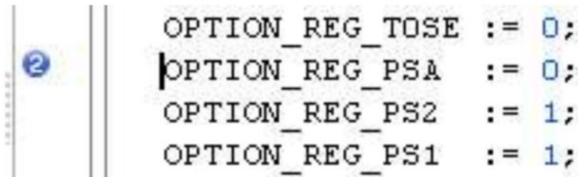
To activate this mode, just hold down the <Alt> and <Shift>, while we lower or raise the cursor with the arrow keys.

3.4.3 Text Markers

Machine Translated by Google

To quickly access special locations in an editor window, bookmarks can be defined.

Bookmarks are indicated by an icon, with the bookmark number, in the left margin of the editor:



Markers are defined by placing the cursor at the position of the text, which is you want to dial, and then use one of the following key combinations:

<Shift>+<Ctrl>+0
<Shift>+<Ctrl>+1
...
<Shift>+<Ctrl>+9

Each combination generates a marker. Only 10 markers can be defined.

Then if you want to access one of the bookmarks, just use the corresponding combination:

<Ctrl>+0
<Ctrl>+1
...
<Ctrl>+9

3.4.4 Synchronized Editing

Synchronized editing consists of editing the same identifier at various points in the editor, but without the need to manually edit each occurrence of the identifier.

This functionality is useful for refactoring source code.

If, for example, you want to edit the name of all identifiers in a selection range, you must select the text, starting with the first occurrence of the identifier:

Machine Translated by Google


```
7  procedure bien;
8  begin
9    pinLed := 1;
10   delay_ms(30);
11   pinLed := 0;
12   delay_ms(20);
13 end;
14 procedure Mal;
15 begin
16   pinLed := 1;
17   delay_ms(2000);
18   pinLed := 0;
19   asm SLEEP end
20 end;
```

Then, press the combination <Ctrl>+J, and various cursors will appear at the points where the identifier appears within the selection.

```
7  procedure bien;
8  begin
9    pinLed := 1;
10   delay_ms(30);
11   pinLed := 0;
12   delay_ms(20);
13 end;
14 procedure Mal;
15 begin
16   pinLed := 1;
17   delay_ms(2000);
18   pinLed := 0;
19   asm SLEEP end
20 end;
```

Allowing you to edit all the identifiers that are within the selection at the same time.

Synchronous editing can only be used with identifiers, including those are found within the comments. It cannot be applied to symbols or spaces.

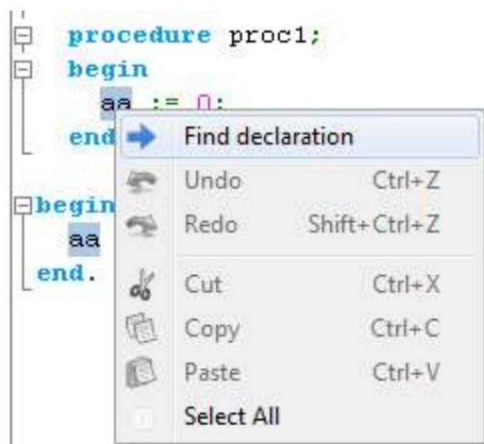
3.4.5 Declaration Search

PicPas offers the possibility of finding the part of the code, in which a variable, constant or procedure, is declared.

To do this, just place the cursor on the element we want to locate, and use the key combination <Alt>+<Up arrow>. At that moment, the cursor will move to the point where the declaration of the chosen element is made.

This functionality can also be accessed using the context menu of the editor and choosing the "Go to declaration" option:

Machine Translated by Google

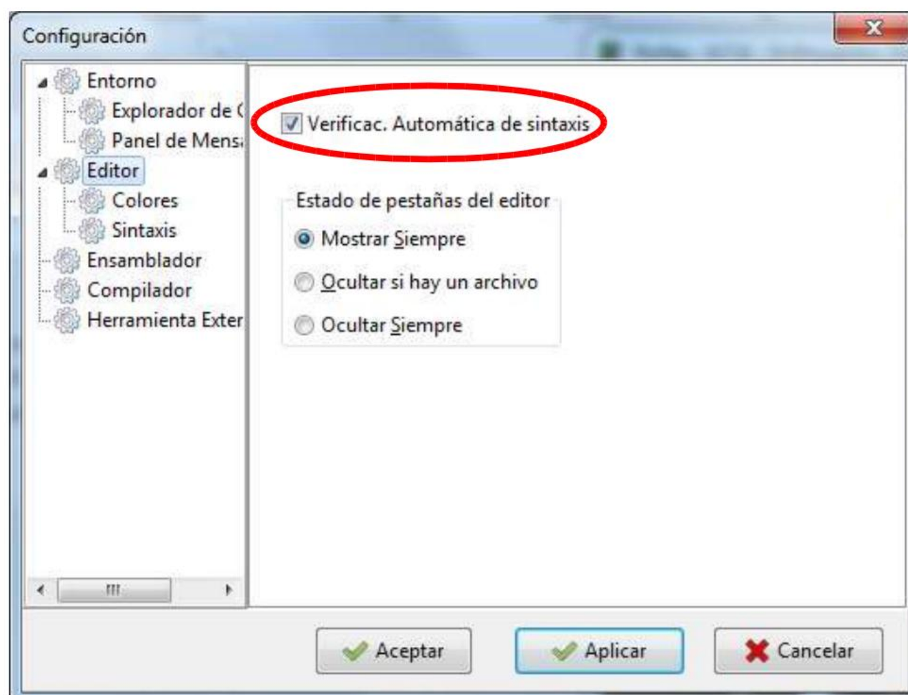


If the declaration is located in another file, that file will be opened in the editor to locate the declaration.

For the location function to work, the current file needs to be able to compile without errors.

3.4.6 Automatic syntax checking

To facilitate the automatic detection of syntax errors (without the need to compile the program or unit), the "Automatic Verification of Syntax" option is included. Syntax", in the configuration window:

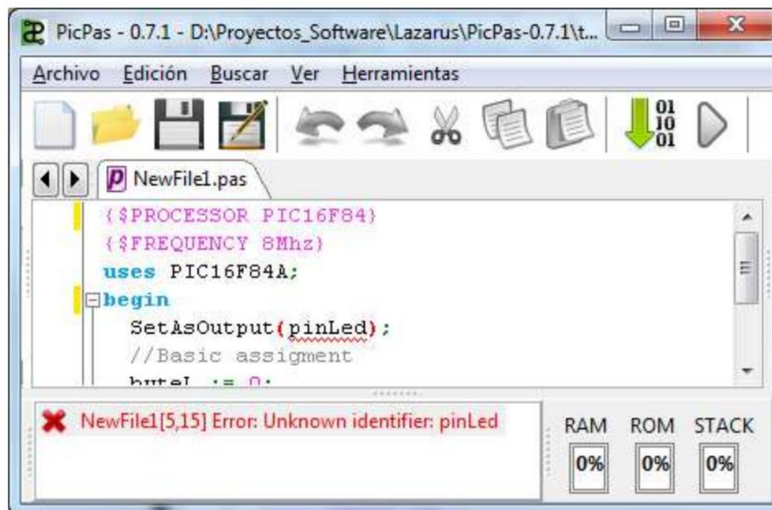


When you activate this option, a syntax check will be done, for a few moments after each modification made to the current file.

Machine Translated by Google

The error messages detected (and warnings) will appear in the message window as the source code is modified. This way you avoid having to constantly compile to find syntax errors.

The element (or elements) where the compiler assumes the error is occurring will also be marked in the editor:



Automatic error checking performs a process similar to compilation, but without linking, so that syntax errors are checked, but "Memory full" type errors, which are detected when compiling completely, will not be detected. the code.

If the code explorer window is visible, it will also be displayed. will update automatically, after automatic syntax checking.

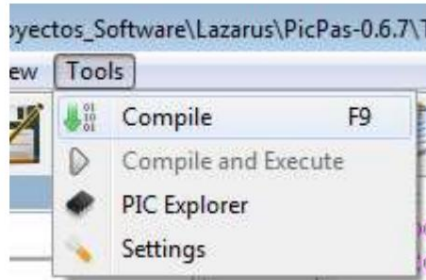
Machine Translated by Google

3.5 Compilation

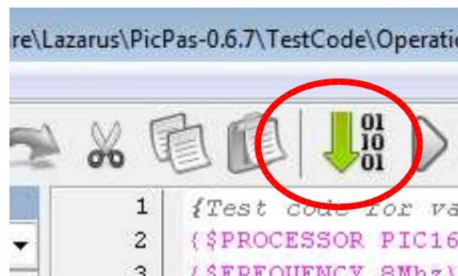
The entire compilation, assembly, and linking process is carried out by PicPas internally, in a single step and without requiring any additional software.

The product of this entire process is a file called: <name>.hex, where <name> is the name of the program that is being compiled.

The build can be started from the main menu:

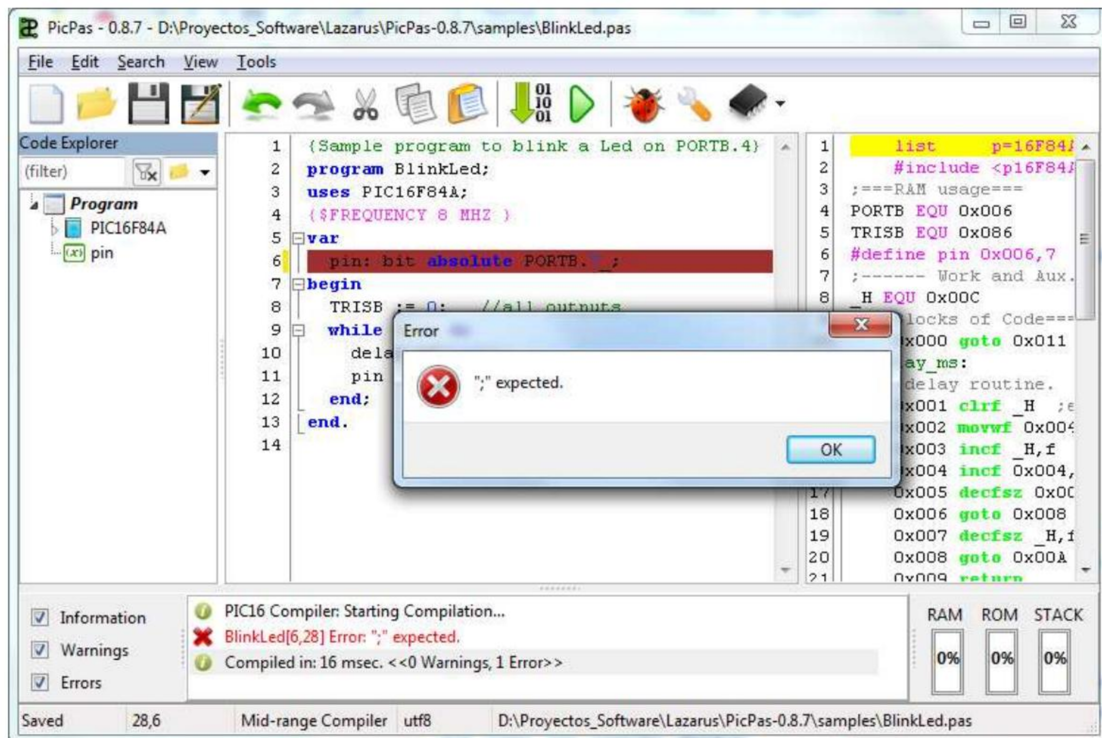


With the shortcut F9, or with the toolbar button:



If an error occurs in the compilation, a message is displayed and the process is stopped. In that case, the cursor is left on the file, where the mistake:

Machine Translated by Google



It is possible to disable the dialog box with the error message from the configuration window.

The position where the compiler detects the error does not always coincide with the actual position of the error, but should be taken as a reference or clue, rather than as an absolute position.

3.5.1 Partial Compilation

Partial compilation consists of performing only the first stage of compilation, which does not include linking, generation of the *.hex file, or updating the assembler window.

This form of compilation is used in some situations (See next section) in which only a quick syntax check is required, or when the generation of *.hex files is not applied, as is the case with units.

The units do not generate *.hex files because they are not complete programs, but rather they are libraries that help in the creation of programs.

3.5.2 Automatic Compilation

The compilation process is not only done when the compile option is chosen. Partial compilation1 can also occur , automatically, in the following situations:

[1] This partial compilation consists of

Machine Translated by Google

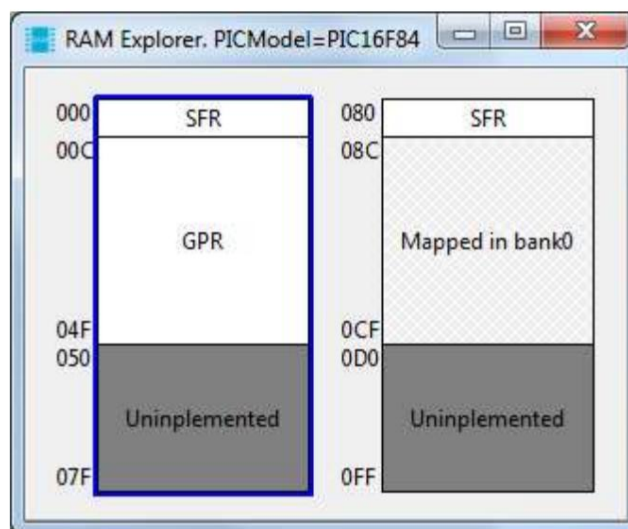
- In some cases of completion, when the autocomplete window is opened manually.
- When writing normally in the editor, when the autocomplete option is activated.
- When the "Automatic syntax verification" option is activated, forcing the code to be partially compiled to detect errors.

Generally, the compilation process is done in a few milliseconds, and the partial compilation process is transparent, but if the program is very extensive, and there is some delay when writing in the editor, the mentioned options can be deactivated. .

3.6 RAM Explorer

The RAM explorer displays a map of the RAM memory of the current microcontroller.

It is accessed through the "Tools" menu or through the F12 key. Has the following appearance:



The microcontroller, whose map is shown, is the one defined when compiling a program.

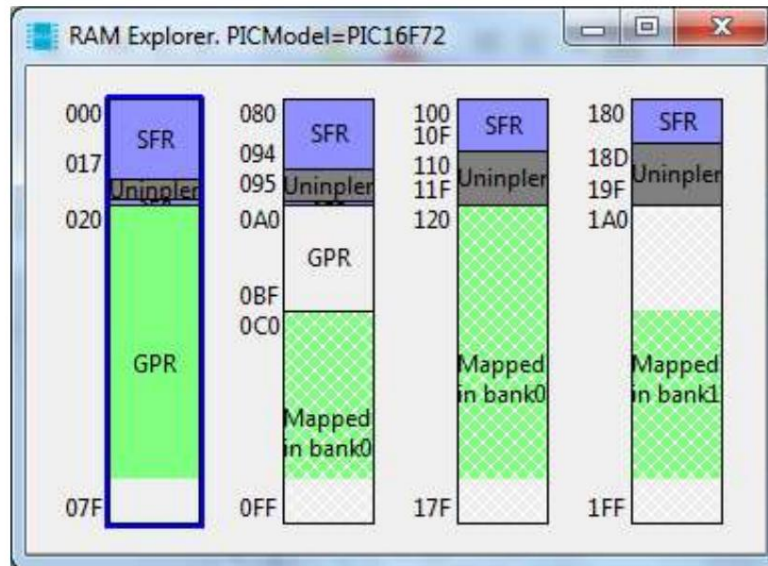
Each bank of RAM is shown in a vertical strip, showing the areas corresponding to the SFR, GPR and Not implemented registers.

The area of each section is proportional to the number of bytes in each section.

The memory areas that are mapped to another bank are displayed, with the text "Mapped to…"

The following figure shows the RAM map of a 4-bank microcontroller:

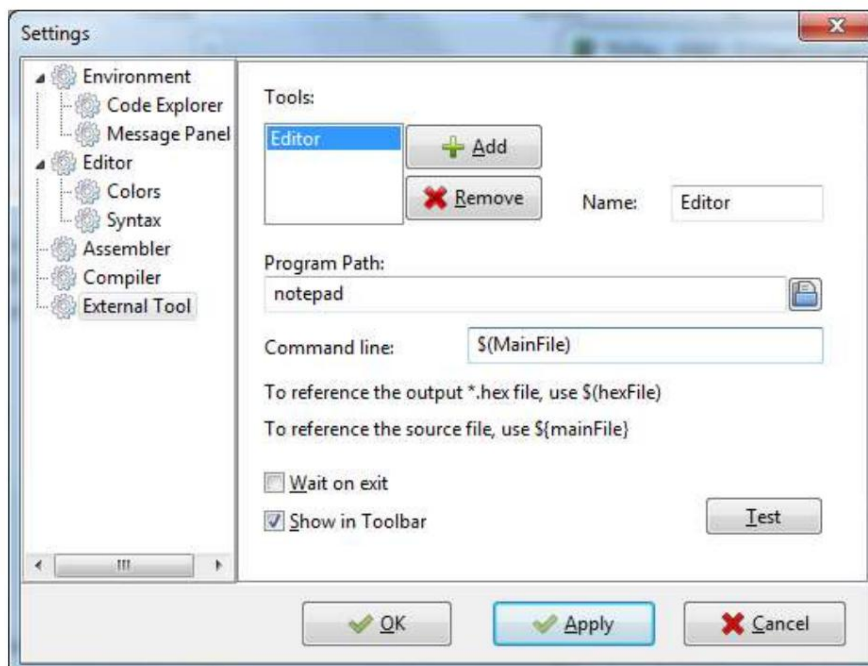
Machine Translated by Google



The RAM areas, which are being used by program variables, in which GPR zones appear as green areas.

3.7 External Tools

You can configure the execution of external programs, through the configuration screen. "External Tool" configuration:



The "Add" and "Delete" buttons allow you to create and delete new accesses for the execution of external programs (External Tools).

Machine Translated by Google

Each external tool is configured with the controls on the right:

Name -> It is the name with which the tool will be identified. this name It will also appear in the "Tools" menu when it is configured.

Program path -> It is the name of the file to be executed, including the full path. For example: d:\prorgamas\prog1.exe

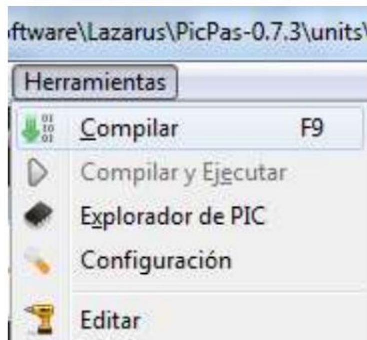
Command line -> These are the parameters that will be passed to you at the time of run the program. For example: /c dir

Here it is possible to use some variables as part of the string. These variables are:

$(hexFile) represents the output *.hex file, including the full path. $(mainFile) represents the current source file (the one being edited), including the full path. $(mainPath) represents the path of the current source file (the one being edited), not including the trailing directory separator. $(picModel) represents the PIC model, which is being used in the program current. It could be something like "PIC16F84".

Wait to finish -> Indicates that PicPas execution stops until the external program finishes executing.

Show in Tool Bar. -> Create access to the program, from the "Tools" menu and from the Toolbar:



Many external tools can be displayed but only the 5 can be shown first accesses, in the menu and in the toolbar.

The assigned icon is fixed and cannot be changed.

3.8 Theme Configuration

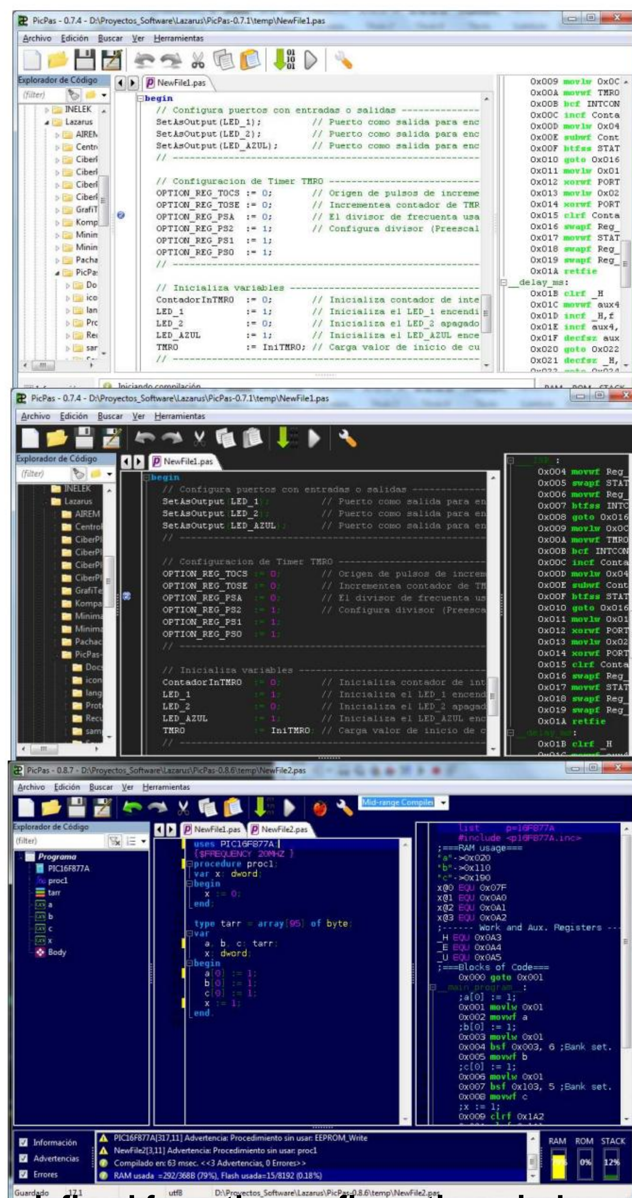
Machine Translated by Google

The PicPas IDE is highly configurable, being able to define the color and characteristics of the editing windows (Pascal and ASM), the file explorer, the message window, the menus, etc.

Color settings and some additional visual aspects can be saved in a single file, so that those settings can be used at any other time.

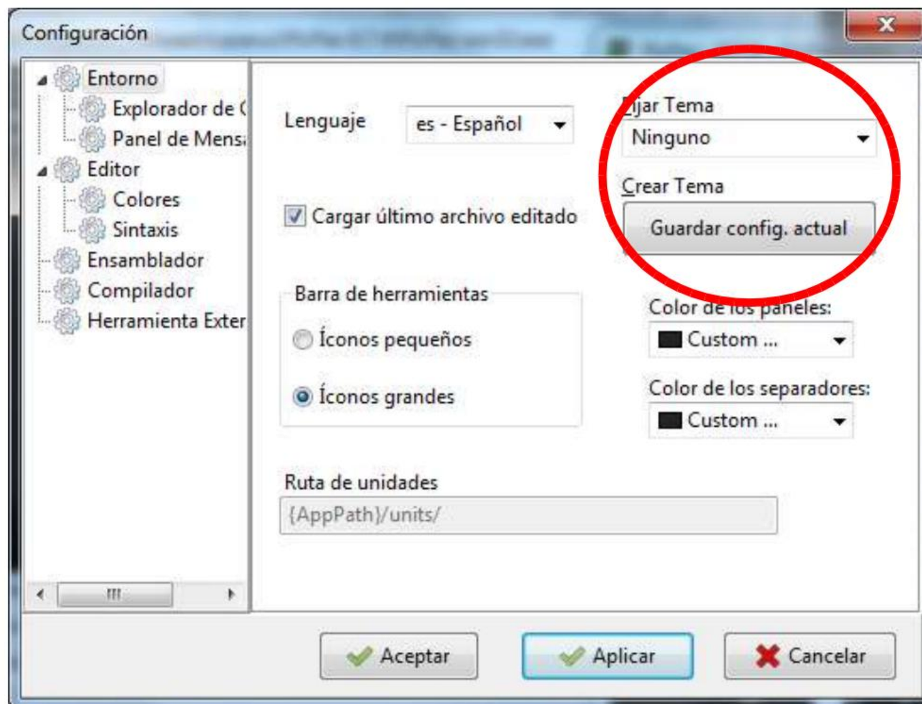
That particular configuration is called "themes," and it allows you to quickly change the appearance of the IDE.

The following figure shows two aspects of the same IDE, with different themes:



Themes are defined from the configuration window

Machine Translated by Google



To set the IDE with a new theme, you must choose one from the "Set Theme" drop-down list, which is usually set to "None."

When you press "Apply" or "OK", the settings for the selected theme will be loaded and the current settings will be lost.

It is important to ensure that if you are going to make custom settings, you must have the dropdown list set to "None", otherwise these changes will be ignored.

PicPas comes with some predefined themes, but you can create whatever you want. To create a new theme, configure the IDE as desired, and then press the "Save Settings" button. current".

Providing a name will save the current configuration as a theme, in the /themes folder, reserved to store all PicPas themes.

Among the parameters that are saved in a theme are:

- **The color of the panels and the dividers of the IDE panels.**
- **Text colors, and highlighting of the Message Panel.**
- **Code explorer text and background colors.**
- **Pascal and ASM syntax element colors.**

Machine Translated by Google

4 DEBUGER/SIMULATOR

PicPas includes a code simulation and debugging tool assembler, within the same application.

The code to simulate is obtained from the same PicPas compiler. This implies that Only programs compiled by PicPas can be simulated, not from other sources.

The simulator has the following features:

- Allows execution of programs in real time.
- Shows the elapsed time and the number of cycles executed.
- Can graphically display the logic value of the microcontroller pins.
- It only simulates the execution of instructions and RAM and FLASH memory.
- Does not include EEPROM simulation, counters, interruptions, comparators, ADC, and other peripherals.
- Allows you to connect electronic components to the input and output pins of the microcontroller.
- Performs electrical simulation by node analysis, which includes calculation of voltages and impedances.

This debugger has the following features

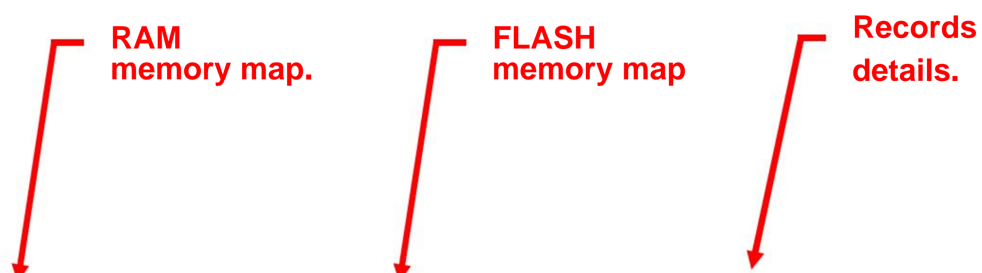
- Includes RAM and FLASH memory maps.
- Assembly code viewer -
Execution control, “Reset” and Pause.
- Step-by-step execution, by instructions and subroutines.
- Possibility to go back in the execution of the program.
- Inspection window for records, configured by your name or your address.
- Details inspection window for W and STATUS records.
- Stack overflow detection.

4.1 Starting the debugger/simulator

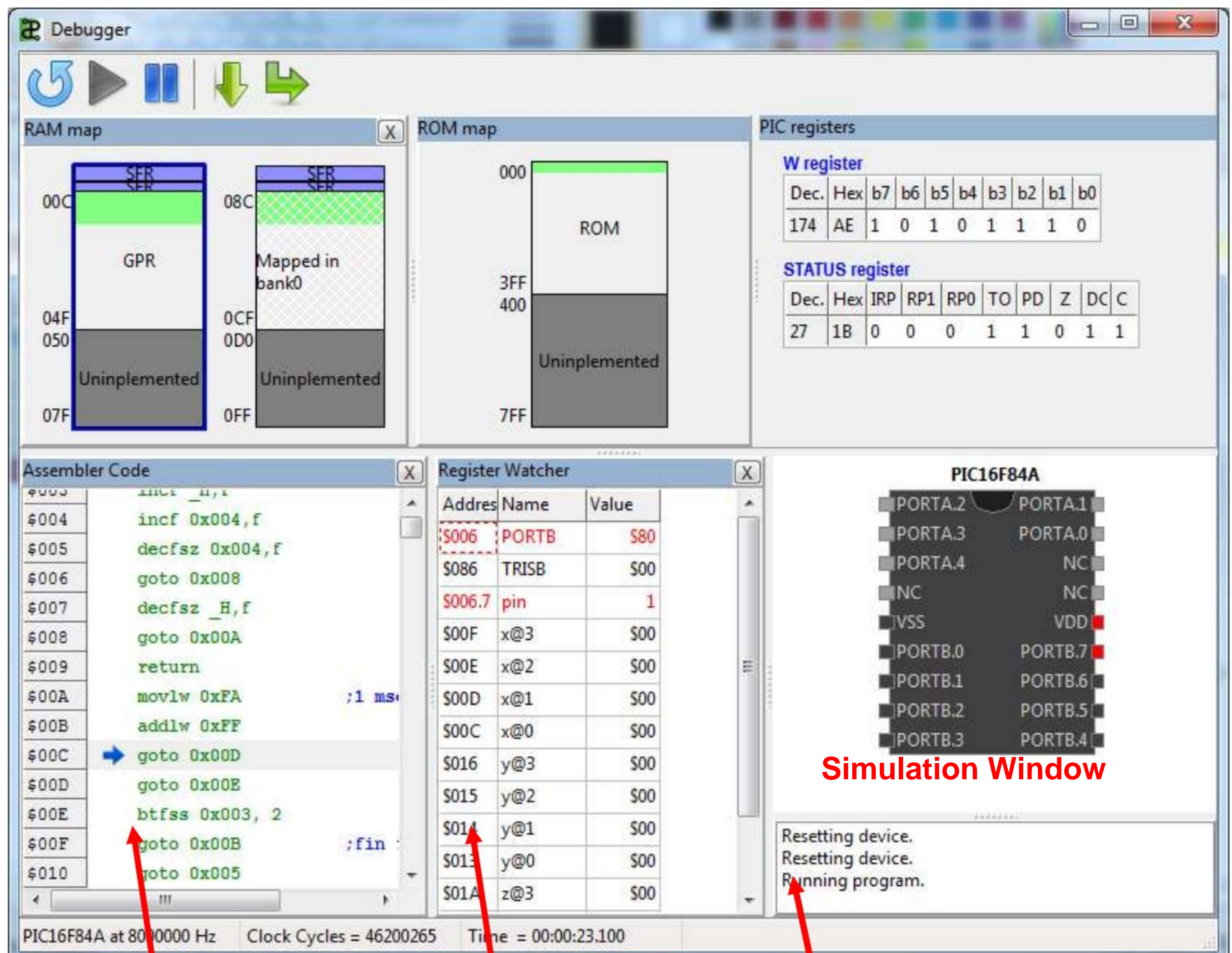
Before starting debugging, you must have the program ready to be debugged. The program to debug is the one that is compiled from the main PicPas window.

Once compiled, without errors, you can start debugging.

The ASM debugger/simulator resides in the same window. Can access it from the main menu Tools>Debugger, or with the shortcut Ctrl+F12.



Machine Translated by Google



4.2 Memory maps

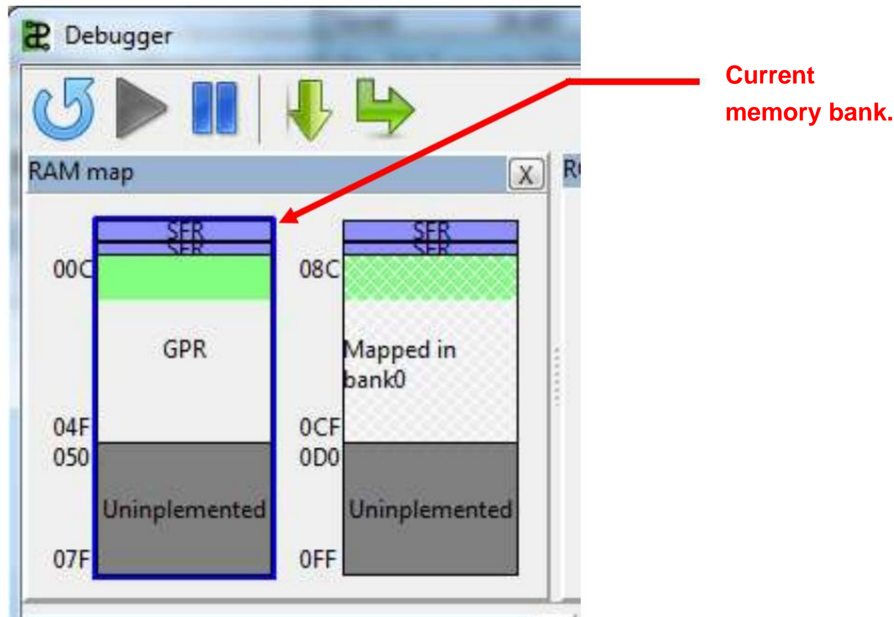
RAM Map is a reduced version of RAM Explorer described in section 3.6.

FLASH memory map is similar to RAM memory map, but applicable to ROM or FLASH memory.

Both memory maps are updated in step-by-step execution, or in real-time simulation.

The RAM memory map provides, as additional information, the bank of memory currently targeted by the microcontroller. This is highlighted with a thicker blue line:

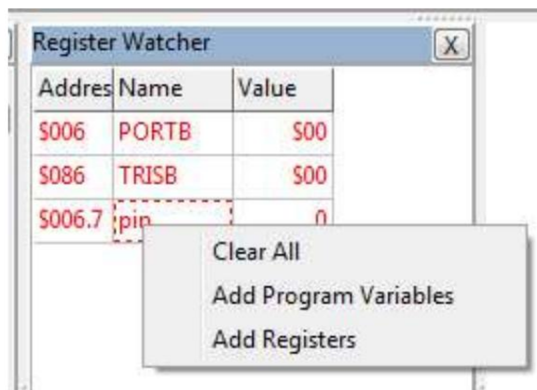
Machine Translated by Google



4.3 Records Inspector

The Records Inspector allows you to see the value of the listed records:

By default they show the addresses of the variables used in the program, but the additional records are not shown. To add the additional records, It can be done manually or using the context menu:



Machine Translated by Google

With this menu you can also clear the list and add program variables again.

Additional variables or additional memory locations can be added by moving the cursor after the last row and directly putting the register name or its address into a new row:

Register Watcher

Addres	Name	Value
$006	PORTB	$00
$086	TRISB	$00
$006.7	pin	0
$004		

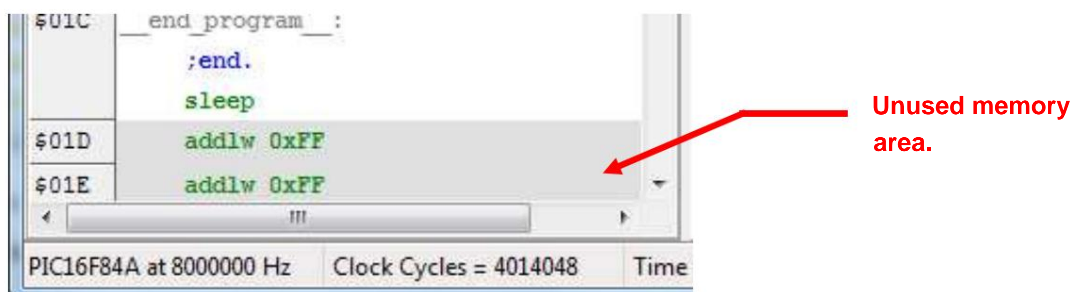
The addresses of the variables must always be written in hexadecimal with three digits and preceded by the "$" symbol. To add BIT or BOOLEAN type variables by their address, you must indicate their physical address and the bit number, such as: "$020.1".

The register names used are those generated by the compiler, according to the program.

• In the case of single-byte variables (BYTE or CHAR), the same name as the variable is used. • In the case of single-bit variables (BIT or BOOLEAN), the name of the variable is also used. • In the case of variables of more than one byte, the name of the variable is used, adding the subfixes @0, @1, @2, … for each byte of the variable.

4.4 Assembler Panel

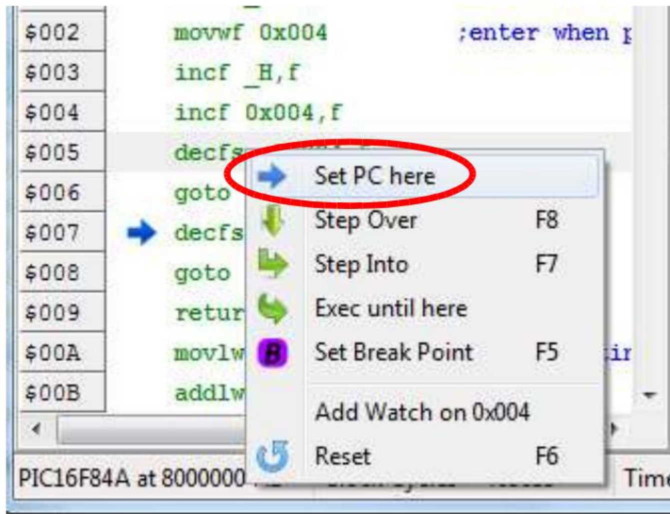
In the assembler panel, each row represents an address of the device's FLASH memory. The comments that appear can be activated or deactivated from the configuration window, in the "Assembler" section.



Machine Translated by Google

The unused part of the FLASH memory is drawn in Gray in the control panel. assembler. The program should never execute these instructions.

The current instruction to be executed can be changed at any time, breaking the normal flow of execution. To do this, you must use the context menu and choose the "Set PC here" option.



Consider that jumping or going back in the execution of a program does not change the status of the other registers, but is only an aid for debugging the code.

4.5 Execution control

When opening the debug window, the simulator is stopped by default (paused) after a restart has been generated (Reset).

To control the execution, the following keys can be used:

F5 -> Places a breakpoint at the current position, in the assembly code.

F6 -> Restart the device.
F7 -> Execute step by step by instruction.
F8 -> Execute step by step per subroutine.
F9 -> Run the program in real time.

Step-by-step execution allows you to execute instruction by instruction, to see the status of the records at each step. It is used for debugging.

The instruction that will be executed when you press F7 or F8 is always marked with a blue arrow, in the Assembler Panel:

Machine Translated by Google


```
Assembler Code
$015      movwf _H
$016      movlw 0xE8
$017  ->  call 0x002
$018      ;pin := not pin;
          movlw 0x80
$019      xorwf PORTB,f
$01A      ;end;                  ;Bank set.
          bsf 0x003, 5
$01B      goto 0x013
$01C  __end_program__:
          ;end.
          sleep
$01D      addlw 0xFF
$01E      addlw 0xFF
PIC16F84A at 8000000 Hz   Clock Cycles = 4014048   Time
```

If the statement to be executed is a subroutine call (CALL statement), pressing F7 will skip to the statement and execute each statement in the subroutine. On the other hand, when you press F8, in a CALL instruction, the entire subroutine will be executed in a single step.

4.6 Execution information

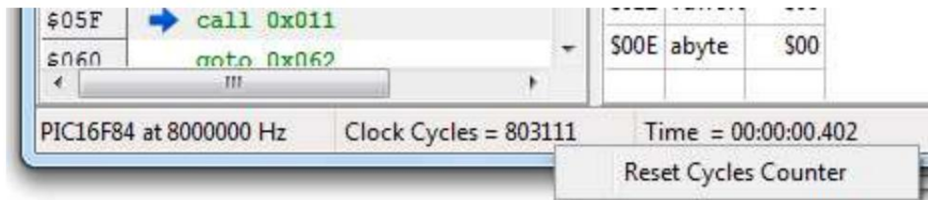
As the program runs, you can see the number of cycles consumed, and the elapsed execution time in the Status Bar.

```
$05F  ->  call 0x011            $00E  abyte   $00
$060      goto 0x062
PIC16F84 at 8000000 Hz   Clock Cycles = 803111   Time = 00:00:00.402
```

It is possible to count the number of cycles that a subroutine consumes, using the cycle counter. To do this, it will be enough to reset the cycle counter, before executing the subroutine, then execute the subroutine and see the elapsed cycles.

To reset the cycle counter, you must use the context menu in the status bar (right mouse button):

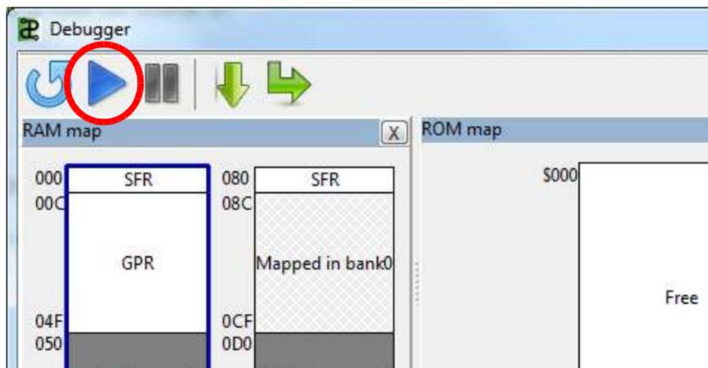
Machine Translated by Google



Machine Translated by Google

4.7 Real-time execution

Pressing F9, or using the button, starts the program execution process in real time.



Real-time simulation implies that the same number of instructions per second will be executed as those that would be executed on the real device, at the same clock speed.

So, for example, if a PIC16F84 at 4MHz is being simulated, the simulator It will execute a million instructions per second.

As the instructions are executed, the panels, registers or RAM and FLASH memory maps are updated. This refresh is done with a frequency of 5 frames per second, therefore higher speed changes may not be detected.

The diagram in the Simulation Window will also be updated:

By default, only the microcontroller is shown, showing the details of the pins. These are colored according to the state of their logic and impedance levels:

Machine Translated by Google

COLOR	STATE
Black	Low impedance, zero voltage.
Red	Low impedance, voltage at Vcc
Grey	High impedance.

The program execution will stop automatically, if:

- **It is stopped manually by pressing the “Pause” button.**
- **It is stopped manually by pressing the “Reset” button.**
- **A breakpoint is detected.**
- **A SLEEP instruction is executed.**
- **An attempt is made to execute an area of memory that has not been written by the program.**
- **An attempt is made to execute address $000. As in the case of a Reset.**

Machine Translated by Google

5 LANGUAGE REFERENCE

5.1 Reserved Words

Within the Pascal language, which PicPas implements, there is a group of words which are reserved by the compiler and cannot be used for any other purpose, other than the one assigned to them by the compiler. These words are:

ABSOLUTE AND ASM
BEGIN BIT BOOLEAN BYTE
CONS DIV DO
END EXIT ELSE ELSIF
FALSE FOR IF
IMPLEMENTATION INTERFACE INTERRUPT
MOD NOT OR PROGRAM
RESULT REPEAT
THEN TO TRUE TYPE
UNIT UNTIL VAR
WHILE WORD XOR

5.2 Comments

Comments can be a single line, using the characters //:

```
//This is a comment
```

Or multiple lines, using the { and } delimiters:

```
{This is a multi-line comment}
```

Optionally, delimiters (* and *) can be used for comments.
several lines.

```
(*This is also a multi-line comment*)
```

Nesting of comments is not supported. This means that the comments of a single line will always begin with // and will always extend to the end of the line, no matter what characters are found.

In the same way the { symbol will open a comment that will be closed only with the } sign, no matter how many { characters are in the way.

5.3 Numbers

Machine Translated by Google

The numbers in PicPas are expressed in the usual way:

```
0
123
10000
```

In the current version of PicPas, decimal or negative numbers are not supported. Therefore, the following numbers are considered erroneous:

```
-1
2.5
1e20
```

Numbers can also be expressed in hexadecimal format:

```
$15
$FFFF
```

Or in binary format:

```
%10010011
%01
```

5.4 Identifiers

Identifiers are words used to name variables. constants or other elements of a program.

In Pascal, identifiers must have at least one letter and are formed applying the following rules:

- The first character must always be a letter, whether uppercase or lowercase or the underscore character "_".
- The following characters, if any, can be any letter, number or the underscore character "_". • Identifiers cannot be reserved words.

Special characters, such as space, signs, or accent characters, are not They can be part of an identifier.

The following identifiers are valid:

```
to
x1
long_name
variable20
a123b_
```

The following identifiers are not allowed

```
1name          //Start with number
```

Machine Translated by Google


```
the_child         //Contains an invalid character
my name //Contains space
var               //It is a reserved word
```

Within Pascal, identifiers are not case sensitive, this means that, for Pascal, the identifiers: “Pascal”, “PASCAL” or “Pascal”, are exactly the same identifier.

5.5 Structure of a program

A typical Pascal program has the following structure:

Program Identification
Unit declaration
Declarations of: - Constants - Variables - Guys - Procedures
Main body of the program

Of these 4 sections, only the last one (the main body of the program) is mandatory.

Thus, the simplest (and most compileable) program that can be written would be, something like this:

```
begin
   //Main body of the program end;
```

A more didactic example, in which some statements are shown, could be this:

```
//Hardware definition
{$PROCESSOR PIC16F628A}
{$FREQUENCY 20MHZ }

program BlinkLed;
// Const constant declaration

   VALUE = $86;

//Type declaration type number
= byte;
```

Machine Translated by Google


```
//Declaration of variables
var
   var1 : number;
   var2 : number;

//Main body of the program
begin
   var1 := VALUE;
end.
```

The body of the program always goes at the end, between the reserved words BEGIN …END, and always ends with a period.

The main program is a set of instructions. The most common instructions are assignments:

```
x := 1;
variables := x + y;
```

There may also be instructions that are procedure calls or function:

```
Do something;

ExecuteFunction(x,y);
```

Or, the instructions can be conditional or iteration structures:

```
while i>0 do do_something_with_i; end;

if i>0 then do_something_with_i; end;
```

Conditional structures are described in section 5.12.

All statements must end with the semicolon (;) delimiter.

5.6 Operators

PicPas supports a subset of Pascal operators. These are:

ARITHMETIC OPERATORS

OPERATOR	NAME
+	Addition
-	Subtraction
-	Sign
*	Multiplication

Machine Translated by Google

Div	Integer division
Mod	Residue

The multiplication, division and remainder operators are only implemented for some types in the current version of the compiler.

All arithmetic operators work on integer data, because in the current version of PicPas, it does not support floating point numbers.

LOGICAL OPERATORS

OPERATOR	NAME
NOT	Logical NOT operation
AND	Logical AND operation
OR	Logical OR operation
XOR	XOR logical operation

Logical operators work with Boolean variables, according to the following tables:

X	Y	X AND Y
True	True	True
True	False	False
False	True	False
False	False	False

X	Y	X OR Y
True	True	True
True	False	True
False	True	True
False	False	False

X	Y	X XOR Y
True	True	False
True	False	True
False	True	True
False	False	False

X	NOT X
True	False
False	True

And they work the same way with bit variables, with the exception of using 1 instead of True and 0 instead of False.

BIT OPERATORS

Machine Translated by Google

OPERATOR	NAME
NOT	Logical NOT operation
AND	Logical AND operation
OR	Logical OR operation
XOR	XOR logical operation
>>	Logical shift to the right
<<	Logical shift left

The same logical operators allow operations to be carried out at the bit level, in numerical variables.

For example, the following code inverts the value of all bits on port B:

```
PORTB := PORTB xor $FF;
```

COMPARISON OPERATORS

OPERATOR	NAME
>	Greater than
>=	Greater than or equal
<	Minor
<=	Less than or equal to
=	Equal
<>	Different

Comparison operators always return a Boolean expression.

ASSIGNMENT OPERATORS

OPERATOR	NAME
:=	Assignment
+=	Assignment with sum

The assignment operation with addition is used as follows:

```
variable += value;
```

Which is equivalent to:

```
variable := variable + value;
```

Machine Translated by Google

The advantage of using the assignment operator with addition is that the statement is compiles more compactly than in the long version.

Machine Translated by Google

5.7 Operator precedence

The operators are evaluated according to the following precedence:

OPERATORS	PRECEDENCE
NOT, sign “-“	6
*, DIV, MOD, AND	5
+, -, OR, XOR	4
=, <>, <, <=, >, >=	3
:=, +=	2

The operators that have the highest precedence are the ones that are evaluated first in an expression. So, for example, in the following expression:

1 + 2 * 3

Operation 2 will be evaluated first * 3, because the multiplication operator “*” has higher precedence than the addition operator “+”.

To change the order of evaluation, expressions can be grouped, using parentheses:

(1 + 2) * 3

5.8 Data types

PicPas supports the following data types:

GUY	SIZE	RANGE
Bit	1 bit	0 or 1
Boolean	1 bit	TRUE or FALSE
Char	1 byte	chr(0) .. chr(255)
Byte	1 byte	0..255
words	2 bytes	0..65535
DWord	4 bytes	0.. 4294967295

To save memory, the Bit and Boolean types occupy only one bit.

Signed variables are not handled, nor is a number floating point, in the version current of the compiler.

DWord variables have limited support in the current version of PicPas. Only assignment, comparison (= and <>), addition and assignment with addition.

Machine Translated by Google

5.8.1 Char Type

The “char” variables store simple characters, such as the letter 'A', or the number '0'. It is the only data type that handles alphanumeric characters.

The assignment of values is done in the following way:

```
var
   c:char;
begin
   c := 'a'; end.
```

The characters must be enclosed in single quotes; double quotes cannot be used.

You can also express literals of type Char, using the Chr() function:

```
   c := chr(65); //Equivalent to ac := 'A';
```

What is indicated to the chr() function, as a parameter, is the ASCII code of the character you want to obtain.

You can also use the “#” character and the ASCII code directly:

```
   c := #49; //Equivalent to ac := '1';
```

The operations allowed on the char type are:

Assignment: var_char :=
'a'; Comparison: if (var_char = 'a') …
Conversion to byte: n := ord(var_char);

5.8.2 Bit Type

The Bit type is a type that does not belong to the standard Pascal. It behaves similar to the Boolean type, and like it, it occupies only one bit of memory.

They are declared in the usual way:

```
var
   var_bit: bit;
```

Machine Translated by Google

Supports the same logical operators as boolean types (AND, OR, XOR, NOT):

```
x := var_bit_1 AND var_bit_2; x :=
var_bool_1 AND var_bool_2 x := NOT var_bit
x := NOT var_bool
```

However, the way they are initialized is different:

```
var_bit := 1;
var_boolean := TRUE;
```

Variables or bit expressions cannot be used as conditions directly. That is, if a verification of the form is attempted:

```
if var_bit then ...
```

An error will be raised, because a boolean expression is expected within the IF conditional.

To overcome this problem, a type comparison must be done:

```
if var_bit = 0 then ...
```

Without expecting loss of efficiency, since the compiler optimizes the operations with Bit variables at the same level of efficiency as with boolean variables.

You also cannot directly assign values between Bit and Boolean, because semantically, they are considered different data types.

5.8.3 Word and DWord type

Word variables take up two bytes in memory. Usually these two bytes They are consecutive. Let's consider the following statement:

```
var num: word;
```

This statement will place two bytes in consecutive positions:

0x20 -> Low Byte
0x21 -> High Byte

The case of DWord variables is similar:

Machine Translated by Google


```
var num: dword;
```

0x20 -> Byte 0
0x21 -> Byte
1 0x22 -> Byte
2 0x23 -> Byte 3

In some critical cases, when you are at the end of a bank of RAM, the compiler can optimize the use of RAM, making the two bytes of a Word variable occupy positions in different banks.

The behavior is similar in the case of DWord variables, except that they occupy 4 bytes in memory.

5.8.4 Type conversion

Pascal is strict about types. Assignments and operations should be done only between allowed types. For example, you cannot assign a "char" to a byte or a boolean.

To convert between types, the conversion functions must be used:

Ord(<char>)
Chr(<byte>)
Bit(<expression>)
Byte(<expression>)
Word(<<expression>)

For example, to convert a Char to its ASCII code, Ord() is used.

All variables and constants within PicPas have an assigned type, at the time of their declaration or their appearance in the code.

PicPas does not do implicit type conversions. All operations must be defined on the type of data used. There is strictly no such thing as "compatible types". If type compatibility is allowed, it is because such an operation is implemented within the compiler.

So, if you have the code:

```
var large_num: word;
...
  large_num := 200;
```

Machine Translated by Google

This operation would be type incompatible, because a byte (the number 200 is encoded as a byte) is being assigned to a word, so legally we should do:

```
var large_num: word;
...
   large_num := word(200);
```

But PicPas allows the first assignment, because internally, the <word> := <byte> operation has been implemented.

As well as this operation, there are other "compatible" operations that are implemented. But in general, the rigidity between types, which characterizes Pascal, is maintained.

5.8.5 Type properties

A non-standard feature of Pascal, implemented within PicPas, is the be able to use properties, in the form of fields, to basic types such as byte or word.

For example, to access the lowest bit of a Byte type variable, we can do:

```
var
   a_byte: byte;
...
   a_byte.bit0 := 1;
```

The "bit0" field is already internally defined for all byte variables and constants.

In the case of constants, we proceed in a similar way:

```
const
   a_byte = $10;
...
   a_byte.bit0 := 1;
   a_byte.bit0 := (16).bit1;
```

Note that, for numbers, they must be enclosed in parentheses, to avoid confusing the number with a decimal number, due to the effect of the point.

The following properties have been defined for the Byte type:

Field	Description
bit0	Returns bit 0 of the byte
bit1	Returns bit 1 of the byte

Machine Translated by Google

bit2	Returns bit 2 of the byte
bit3	Returns bit 3 of the byte
bit4	Returns bit 4 of the byte
bit5	Returns bit 5 of the byte
bit6	Returns bit 6 of the byte
bit7	Returns bit 7 of the byte

You can also use the simplified notation:

```
a_byte.0 := 0;
a_byte.7 := 1;
```

But this notation is maintained only for compatibility. The recommended way, is to use bit0, bit1, etc.

For the Word type, two properties are defined:

Field	Description
Low	Returns the low byte
High	Returns the high byte

So, for example, to copy the low byte into the high byte of a word, you can do:

```
un_word.High := un_word.Low;
```

Also, the access can be nested, so that, to clear the highest bit weight of a word, you can do:

```
un_word.High.bit7 := 0;
```

For the DWord type, four properties are defined:

Field	Description
Low	Returns byte 0
High	Returns byte 1
Extra	Returns byte 2
Ultra	Returns byte 3
LowWord	Returns the Word under
HighWord	Returns the tall Word

Machine Translated by Google

So, for example, to reverse the words of a Dword, you could do:

```
tmp := a_dword.HighWord;
un_dword.HighWord := un_dword.LowWord;
un_dword.LowWord := tmp;
```

From the compiler's point of view, the properties of variables behave as if they were common variables, therefore, they can be worked on in the same way as working with a variable.

In the same way, the properties of constants behave like other constants, and do not generate compiled code.

Machine Translated by Google

5.9 Variables

Variables are language elements that allow values to be stored, during the course of program execution.

Variables are defined with an identifier in the “VAR” section of the statements:

```
var
  var1 : byte; var2 :
  bit;
  long_name_variable: boolean;
```

All variables must be declared indicating their type.

There cannot be two variables with the same name, at the same level of the program.

The compiler allocates space for the variables in free addresses. memory, defined by the compiler's criteria.

5.9.1 Absolute Variables

To define a specific memory address for a variable, use the reserved word ABSOLUTE:

```
var
  PORTB: BYTE absolute $06; pin1 :
  boolean absolute PORTB.1; pin0 : bit absolute
  $06.0;
```

The alias “@” can be used as a replacement for the word ABSOLUTE:

```
var
  PORTB: BYTE @ $06;
```

Addresses can be expressed as numbers in decimal, binary or hexadecimal format:

```
  varAbs1: byte absolute 32; varAbs2:
  byte absolute %010001; varAbs3: byte
  absolute $20;
```

The bit and boolean variables need to specify, in addition to the address, the bit number.

Machine Translated by Google

Type properties (See section) can be used to specify the reference of an absolute variable:

```
program name;
var
   un_word: word;
   un_bit: bit absolute un_word.Low.bit0;
```

Two variables occupying the same absolute position behave as a single variable. What is written in one will be reflected in the other.

Absolute variables also take up memory space when they point to the user's memory area. The compiler will not place new variables in positions occupied by absolute variables.

The following code shows memory usage cases:

```
{$PROCESSOR PIC16F84}
var
   stat: byte absolute $03; var1: byte
   absolute $20;
```

The "stat" variable does not use user RAM, because it is located in a position, which corresponds to the PIC16F84 system registers.

However, the variable "var1" is located in the user's RAM, and therefore, that memory position will be assigned and reserved. Considering it as "used" memory.

Word type variables occupy two bytes. If specified as Absolute, They will always be located as consecutive positions. For example, the following code:

```
var
   a: byte; b:
   word absolute a;
```

It will cause the lowest byte of "b" to be located at the same physical address as "a", and the highest byte will be located at the next address. This can generate a compilation error if the variable "a" has been located right at the end of a bank of RAM.

5.9.2 REGISTER variables

A particular feature of PicPas is that it has a special type of storage for variables: REGISTER storage allows you to use variables that use internal registers of the microcontroller as storage.

The advantage of a REGISTER variable is that it does not take up memory space and its access is faster.

The disadvantage is that since the PIC has only one register, its contents are lost after executing some operation that uses that register. Which in the case of the PIC, are the majority of instructions.

In the current version of PicPas, this type of storage is only allowed for procedural parameters. So it is possible to define a procedure of the form:

```
procedure SetValueQuickly( fleeting register: byte); begin

   //Be careful what you put here
   PORTB := fleeting;
end;
```

This procedure has its "fleeting" parameter, of type REGISTER, which means that it will receive the value directly in register W. Therefore, and since W is a volatile register, the first thing to do is use "fleeting", before it is lost.
If for example this were done:

```
procedure SetValueQuickly( fleeting register: byte); begin

   PORTA := $FF; //Use the W register
    PORTB = fleeting; //"fleeting" has the value of the last instruction (maybe
$FF)
end;
```

There would be an error, because the "PORTA := $FF" instruction makes use of the W register, and therefore the value of "fleeting" is lost.

This might seem like a disadvantage, however, using the parameters REGISTER allow:

• Assign them more quickly. •
Read them more quickly. •
Save code in the procedure call.

IMPORTANT: Only one REGISTER type parameter can be declared, and it must always be the last one.

The REGISTER parameters are only implemented with the byte, char and word types, in the current version. However, when used with Word, the instruction savings is less, because you only have one work register in the PIC, and an additional memory location will still be used.

5.10 Constants

Machine Translated by Google

Constants are like variables that do not change their value throughout the program execution. PisPas implements the use of constants, in the common way in Pascal:

```
const
  VALUE = 5;                    //decimal
  VarDir = $21;                 //hexadecimal
  Macara = %0101;               //binary
  AT = '@';                     //character
```

Constants are defined with the "=" operator, and can be defined in decimal, hexadecimal or binary format.

Constants, like variables, also have a type. This guy, without However, it is not usually specified, but is assumed by the compiler.

For example, in the following definition:

```
const
  HIGH = true;
```

The compiler will create the constant "HIGH" as a boolean type, and it will have all the characteristics and restrictions of the boolean type.

Numerical literals are assigned the smallest possible type, which can store them. So we have to:

- Literal 123 is read as a constant of type Byte.
- Literal 256 is read as a Word type constant.
- The literal 65536 will generate an error because PicPas does not have a type numeric to store at this value.

The most basic numeric type is the byte, although the values "0" and "1" can be assumed to be of bit type, the byte type will always be assumed for numbers less than 256.

To force the type assigned to constants, you can use some of the type conversion functions, so you could declare word type constants, wearing:

```
const
  LARGE_NUM = word(1);
```

And in this way, a WORD constant will be created, with the value 1. If the conversion function, PicPas will assume the byte type.

In the same way, the conversion functions bit(), or chr() can be used.

Machine Translated by Google

They can also be defined as expressions from other constants.

```
Const
    LongValue = 10000 + Value; //Formula
```

The supported operations are the same as the variables, according to their type.

5.11 Types

Within PicPas, it is possible to declare custom names for basic data types.

The following example shows how the type "number" is created, as a equivalent of the word type, and the character type, as an equivalent of the "char" type:

```
//Define types type

number = word;
    character = char;

//Create variables of those types
var
    num: number;
    char: character;
```

The definition of types also allows creating arrays.

5.11.1 Arrangements

PicPas, in its current version, supports a basic definition of arrays, for the byte, char and word types.

Arrays must be declared using an array type:

type <array_type> = array[<number of elements>] of <element type>;

The number of elements is an integer from 1 to 127. The maximum size of an array will depend on the memory available on the device, and should not exceed the size of a bank of RAM.

The following example declares a byte array of 10 elements:

```
type
    abytes = array[10] of byte;
var
    a: abytes;
```

Machine Translated by Google

The first element of an array is the value a[0], and the last is a[9]. To access to the size of an array, the "a.length" property would be used. Creating arrangements is not allowed with initial index different from zero.

The following code shows how arrays are used on supported types:

```
type //Array types
    Cross out = array[3] of char;
    Tabyte = array[3] of byte;
    Taword = array[3] of word;
var //array variables
    achar: Cross out;
    abyte: Tabyte;
    aword: Taword;
begin
    achar[0] := 'a';
    abyte[1] := 1;
    aword[2] := word(5000);
end.
```

There are a group of methods that can be applied to an arrangement. These will listed in the following table:

METHOD	DESCRIPTION	EXAMPLE
length	Returns the length of the array.	x := array.length;
low	Returns the lowest index of the array. For now it is always 0.	x := array.low;
high	Returns the largest index of the array.	x := array.high

Array handling is still limited in the current version of PicPas. A limitation is that you cannot use arrays directly as part of expressions conditionals.

Machine Translated by Google

5.12Structures

PicPas supports the most common structures of the Pascal language:

5.12.1 Conditional IF ... THEN

The IF structure has the following forms:

```
If <boolean expression> then
   <instructions>
end;

If <boolean expression> then
   <instructions>
else
   <instructions>
end;
```

Note that the syntax of the IF conditional is not similar to Pascal's IF. This variation corresponds to one of the improvements in the language that PicPas implements.

The IF conditional must end with the reserved word END, even if it only has one instruction. The following are usage examples:

```
if x=1 then x := 0; end;

if x = 0 then
   y := 1;
   x := y + 1;
else
   y := 0;
end;

if x = 0 then
   y := 1;
   x := y + 1;
elsif x = 1 then
   x := 0;
else
   y := 0;
end;
```

To make additional comparisons, in the same IF, just use the word reserved ELSIF, and then include the condition, as if it were a simple IF.

PicPas can work with the Pascal IF syntax. To do this, you must include first the directive: {$Mode Pascal}

Machine Translated by Google

5.12.2 REPEAT loop

The REPEAT loop has the same shape as in Pascal.

```
Repeat
   <statement or block> until
<boolean expression>;
```

An example of a REPEAT loop would be:

```
   repeat
      inc(a);
   until a>5;
```

5.12.3 WHILE loop

The WHILE loop has the following syntax.

```
While <boolean expression> do <statement
   or block>; end;
```

Note that the syntax is similar to Pascal's WHILE, but the body of the loop always includes the END delimiter, and the BEGIN should not be included.

An example of a WHILE loop would be:

```
while a<5 do
   a := a + 1; end;
```

PicPas can work with WHILE's Pascal syntax. To do this, you must first include the directive: {$Mode Pascal}

5.12.4 FOR loop

The FOR loop has the following syntax:

FOR <numeric variable> := <start value> TO <end value2> DO

END;

Note that the syntax is similar to Pascal's FOR, but the body of the loop is always includes the END delimiter, and should not include the BEGIN.

2 In the current version of PicPas, the final value cannot be an expression but only a constant or variable.

Machine Translated by Google

The numeric variable must be of type BYTE.

An example of a FOR loop would be:

```
for i:=1 to 3 do
    do something;
end;
```

PicPas can work with the FOR Pascal syntax. To do this, you must first include the directive: {$Mode Pascal}

5.13System functions

There is a group of predefined procedures and functions that are part of of the language, without the need to use any unit. These are:

FUNCTION	DESCRIPTION
delay_ms()	Allows you to generate a time delay, the number of milliseconds indicated. It can be indicated from 0 to 65536 milliseconds. To use this function, the clock frequency of the current microcontroller must be at certain allowed values, which are: 1MHz, 2MHz, 4MHz, 8MHz, 10MHz, 12MHz, 16MHz, and 20Mhz.

Inc()	Increments the value of a variable, of type Byte, Char, Word or DWord.
	Example:
	Inc(n); //If n was 0, now it is 1
	If n reaches the upper limit of the value, it will automatically go to the lower limit of the variable value. Thus, if n is Byte, and is equal to 255, apply Inc(n), it will be worth 0.

Dec()	Decrements the value of a variable, of type Byte, Char, Word or DWord.
	Example:
	Dec(n); //If n was 1, now it is 0
	If n reaches the lower limit of the value, it will automatically go to the upper limit of the variable value. Thus, if n is Word, and is equal to 0, when Dec(n) is applied, it will become equal to 65535.

SetBank()	Forces switching to the indicated RAM bank. For example, the instruction: SetBank(1); Generates instructions to switch to bank 1, regardless of the position of the current bank. If an invalid bank is provided, only the lowest-weight bits are read.

Machine Translated by Google

Exit()	**Exits a function or terminates the execution of the main program. If called from within a procedure, exits the procedure. If called from within a function, it exits the function, with the possibility of returning a value.**
Ord()	**Converts a character to a byte value, returning its ASCII code. Example: n := ord('A'); //returns 65**
Chr()	**Converts a byte to a character, using ASCII code. Example: c := chr(65); //returns the character 'A'**

Bit()	**Converts an expression into bit data.
	Example:
	varbit := bit(100); //write 1
	varbit := bit(0); //write 0
	Any value other than zero will return the value bit 1.**

Boolean()	**Converts an expression to data of type boolean.
	Example:
	varbool := boolean(100); //write 1
	varbool := boolean(0); //write 0
	Any value other than zero will return the value bit 1.**

Byte()	**Converts an expression to a Byte.
	Example:
	varbyte := byte(varbit); //returns 0 or 1
	varbyte := byte('A'); //returns 65
	varbyte := byte($0102); //return $02
	When applied to char values, Byte() returns ASCII code.
	When applying variables, Word or Dword, returns the low byte.**

Word()	**Convert an expression into a Word.
	Example:
	varword := word(varbit); //returns 0 or 1
	varword := word(1); //returns 1 like Word
	varword := word('A'); //returns 65 like Word
	varword := word($01020304); //returns $0304
	When you apply variables, Dword, returns the base Word.**

DWord()	**Converts an expression to a DWord.
	Example:
	vardword := dword(varbit); //returns 0 or 1
	vardword := dword(1); //return 1 as Dword
	vardword := dword('A'); //return 65 as Dword
	vardword := dword($0102); //return $0102 as Dword**

SetAsInput()	**Sets a port of byte or bit size as the input port. Examples:**

Machine Translated by Google

	SetAsInput(PORTA) SetAsInput(pin1) //"pin1" must have been defined
SetAsOutput( )	Configures a port of byte or bit size as the output port. Examples: SetAsOutput(PORTA) SetAsOutput(pin1) //"pin1" must have been defined

5.14Procedures

In PicPas you can declare procedures, in the usual way in which they are they do in Pascal.

```
procedure proc;
var
    x: byte; //local variable begin x := 1;
end;
```

Procedures can in turn have local variables. These variables will not be accessible from outside the procedure.

Parameters can also be used within procedures.

```
procedure proc1(par1: byte); begin x :=
even1;
    end;


procedure proc2(par1, par2: byte); begin x :=
even1
    + even2; end;
```

Procedures can, in turn, call other procedures:

```
procedure proc1(par1: byte); begin x :=
even1;
    end;


procedure proc2(par1, par2: byte); begin x :=
even1
    + even2; end;
```

All procedure parameters are passed by value. In the current version, PicPas does not support parameters by reference.

Machine Translated by Google

5.15 Functions

Functions are declared in a similar way to procedures, but indicating the type of data they return.

```
procedure Next(par1: byte): byte; begin exit(even1+1);
end;
```

This function will return the value of the parameter, increased by 1. The call to a function is made in the common way as it is done in Pascal:

```
x := next(1);
```

The system function “exit” allows you to indicate the value that is returned as a result of the function, and ends the execution of the function. You can also use “exit” to exit a procedure, but in this case the parameter should not be indicated.

To optimize simple functions, it is recommended to use REGISTER parameters (Section 5.9.2), to achieve more optimal code. The previous function, then we could write it like this:

```
procedure Next(register par1: byte): byte; begin exit(even1+1);
end;
```

Thus the generated code is 2 instructions shorter and two cycles faster.

PicPas supports procedure overloading, that is, several procedures can be declared with the same name, as long as they have parameters of different types.

For example, in the following code, two functions are declared but they They handle different types of parameters:

```
//Overloaded functions procedure
fun5(value1: byte): byte; begin exit(value1+1);
end;



procedure fun5(value1: word): word; begin

exit(value1+2); end;
```

Machine Translated by Google

Although these functions have the same name, for the compiler they are completely different, and one or the other is chosen, depending on the type of parameter used.

All function parameters are passed by value. In the current version, PicPas does not support parameters by reference.

5.16Interruption Procedure

Interrupt procedures are used to implement interrupt handling with PIC microcontrollers.

To define a procedure as an interrupt service routine (ISR), you must include the INTERRUPT reserved word:

```
procedure mi_isr; interrupt; begin

    ...
end;
```

Declaring a procedure as ISR differentiates it from a common procedure in that:

1. It always compiles, without the need to be called from the code source3 .
2. They are always compiled at program memory address 0x0004.
3. They end with the RETFIE statement, instead of RETURN or RETLW.
4. No additional bank switching instructions are generated at the beginning of the procedure4 . It is the responsibility of the programmer to correctly manage the banks within the routine.

Interrupt procedures do not save the value of the registers, nor They control the flags. This must be done manually.

There should only be one interruption procedure. If more than one is declared, no syntax error will be generated, but only one of them will be recognized.

---

3 By optimizing the PicPas code, it does not compile procedures that are not used in the program.
[4] There is a configuration option that allows bank startup routines to be generated at the beginning of procedures.

5.17 Assembly code

It is possible to directly include assembly code within a program, almost anywhere. To do this, an ASM…END block is used.

```
procedure DoesSomething;
begin
   x := 10;
   asm
      MOVLW 2
      ADDWF $20,F
   end
end;
```

ASM blocks can be written on one or more lines. For example, him following code is completely valid:

```
procedure DoesSomething;
begin
   x := 10;
   asm sleep end end;
```

Note that you should not put ";" after the END delimiter, because it is not treated as a statement, but rather as a comment or directive. Therefore, it is possible to include ASM blocks, even within an instruction:

```
procedure DoesSomething;
begin
   x := 10 asm CLRF $OC end; end;
```

Several ASM blocks can be included, within the same procedure or in the main program.

The mnemonics used for the instructions are standard, but the numeric and character formats are those used by Pascal. This means that the numerical values are expressed as follows:

Prefixing "$" for hexadecimal values: CLRF $20
Prefixing "%" for binary values: CLRF %1101
Using quotes for character codes: MOVLW 'A'

Comments are accepted, within the assembly code, and so are tags. For comments, use the ";" character:

```
procedure DoesSomething;
begin
   x := 10;
   asm
```

Machine Translated by Google


```
        ;one line comment
       MOVLW 2 ;comment at end of line
       ADDWF $20,F
    end
end;
```

You can also manage change in RAM banks, from within the ASM blocks:

```
program exampleASM;
begin
    asm
       bsf 3, 0 ;change to bank 1
       clrf $20 bcf
       3, 0 ;return to bank 0
    end
end.
```

Changing banks in RAM can cause an error in the compilation, or in the generated program.

5.17.1 Labels and jumps

To specify jumps, within ASM code, you can use the pseudo-variable "$", which indicates the current memory address. Thus, the following code represents a infinite loop:

```
asm
    ;infinite loop
    goto $end
```

What this instruction does is encode a jump to the same instruction leap.

You can include displacements to the current address, using operators add and subtract. The following examples show forward and backward jumps:

```
asm
    goto $+1 ;jump one position forward goto $-2 ;jump two
    positions back
end
```

Labels can also be used to indicate the destination address of the jump:

```
asm
    ;infinite loop
```

Machine Translated by Google


```
label:
   DECFSZ $21,f
   GOTO end tag
```

Tags are always defined as an identifier, followed by a colon “:”. It is allowed to put instructions after the labels.

Tags can also be used to create subroutines:

```
   asm
      GOTO start

sub_nothing: ;does nothing
      RETURN
   start:
      DECFSZ $21,f
      CALL sub_nothing
   end
```

5.17.2 Reference to variables

Within an ASM block, it is possible to access variables, considering that:

• Variables of type Byte, Char, Boolean, bit and word can be accessed.
• Global and local variables can be accessed. •
Access to variable names follows the same scope rules as Pascal5 code . •
Variables can be
read or written. • A variable used, from
within an ASM block, will behave as follows:
same way as if it were used from the Pascal6 program .

The following code shows how the program's global variables are cleared:

```
var
  byte1: byte;
  char1: char;
  bit1: bit; bol1:
  boolean; begin //
start at
    low level
   asm
      CLRF byte1
      CLRF car1
      BCF bit1
      BCF bol1
   end
```

5 This means that in name resolution, first a name is searched within the current block, which can be the local variables, and then it will be searched in the previous blocks, which can be the main program, or in the units.

6 This implies that the compiler will apply the same optimization criteria.

Machine Translated by Google


```
end.
```

Note that access to bit variables is done by specifying only the name of the variable, without the need to specify the bit.

Access to Word variables can be done by specifying '.High' or '.Low' to indicate the byte you want to use:

```
var
  w: word;
begin
  asm
    MOVLW 0
    MOVWF w.high
    MOVLW 1
    MOVWF w.low
  end
end.
```

Access to local variables or parameters of a procedure is done using the same name used since Pascal:

```
  procedure Clear(n: byte; char: char) : word; begin

    ASM
      CLRF n          ;sets to zero
      CLRF car ;resets
    END
  end;
```

REGISTER parameters can be accessed from ASM blocks, considering that they are stored in the W register. You should not try to access REGISTER parameters by name.

5.17.3 Reference to constants

It is also possible to access program constants from within a ASM block. The considerations are:

- Constants of type Byte, Char, and word can be accessed.
- Global and local constants can be accessed. •
Access to the names of the constants follows the same rules as range than Pascal7 code .
- A constant used, from within an ASM block, will behave as follows: same way as if it were used from the Pascal8 program .

[7] This means that, in name resolution, first a name is searched within the current block, which can be the local variables, and then it will be searched in the previous blocks, which can be the main program, or in the units.

8 This implies that the compiler will apply the same optimization criteria.

Machine Translated by Google

The following code is an example of using program constants within ASM blocks:

```
const
  CBYTE = 3;

  //constant access
  asm
    MOVLW CBYTE
    MOVWF vbyte
  end

  asm
    BSF vbyte, CBYTE end
```

To access the bytes of Word type constants, you can use the form <constant>.HIGH and <constant>.LOW, in the same way as you do with variables.

5.17.4 Reference to Procedures

Procedures can be accessed from within blocks in assembler, using the same name that the procedure has in Pascal:

```
procedure proc1;
//Procedure in Pascal begin


end;

begin
  asm
    call proc1 end
  end.
```

However, you cannot pass parameters when the procedure needs them.

Only in the case where the procedure uses a single REGISTER parameter of type byte, the register W can be used to pass the parameter:

```
procedure proc1(register x: byte); begin end;



begin
  asm
    MOVLW 5 ;will be passed as parameter
    CALL proc1
```

Machine Translated by Google


```
    end
end.
```

When you have several procedures with the same name, the reference from ASM will always use the last declaration:

```
procedure proc1(x: byte); begin

end;

procedure proc1; begin

end;

begin
    asm
        call proc1 ; will call the second statement of proc1 end

end.
```

5.17.5 Returning results for functions

It is possible to set the result of a function, directly from the assembler. To do this, you must take into account the records that the functions use to return their results.

Depending on the type returned by a function, the following registers or bits must be used:

KIND OF FACT	PLACE WHERE IT IS RETURNED RESULT
Bit or Boolean	Z bit of the STATUS register.
Byte or Char	Accumulator
words	W High byte in internal register H and low byte in accumulator

The H register is an internal register that the compiler creates when using operations that involve the Word type.

A simple example could be returning the result of a Byte function that increments the value of a parameter:

```
    next procedure (even: byte): byte; begin

        asm
            INCF even, w
```

Machine Translated by Google


```
    end
end;
```

The following example shows how to set the result of a Word function:

```
//Function for reading test and returning values procedure SetTo8040: word; //
Returns $8040 begin

    asm
        ;The result of a word is returned in _H (high part) and W (high part)
low)
        MOVLW $80
        MOVWF_H
        MOVLW $40
    end
end;
```

Note that to access the H register, the name "_H" is used, which is a reserved name, in ASM blocks.

5.17.6 The ORG Directive

The ORG directive allows you to specify the current address, where the following instructions will be compiled.

The following code shows how the compiler is forced to compile the instruction "vbit := 1" at the beginning of the program memory.

```
asm
    org 0x00
end
vbit := 1;
```

Changing the memory address of the program can cause an error in compilation, or in the generated program.

You can also use the "$" character to indicate the current memory address, so that the next block will move the program counter back 2 positions:

```
asm
    org $-2
end
vbit := 1;
```

The result of this code will be to overwrite the last two instructions, with the code generated by "vbit := 1".

5.17.7 Programming only in Assembler

It is possible to create programs developed entirely in assembler, with little or no Pascal.

In this case, almost total control is left to the code, and PicPas is used only as a framework and can be used to leverage SDI for development.

The following image shows the "Hello World" program to blink an LED, using only assembler code and a Pascal variable, used as a counter:

PicPas - 0.8.1 - D:\Proyectos_Software\Lazarus\PicPas-0.8.1\temp\NewFile1.pas

Archivo Edición Buscar Ver Herramientas

NewFile1.pas

```
{$FREQUENCY 4 MHZ }
{$PROCESSOR PIC16F84A}
uses PIC16F84A;
var contador: byte;
begin
asm
;--- EJEMPLO DE PROGRAMA EN ASM ---
        BSF   STATUS, STATUS_RP0
        CLRF  PORTB
        BCF   STATUS, STATUS_RP0
;------- BUCLE PRINCIPAL ---------
INICIO:
        BSF   PORTB, 0 ;enciende led
        CALL  ESPERA   ;retardo
        BCF   PORTB, 0 ;apagaled
        CALL  ESPERA   ;retardo
        GOTO  INICIO
;-------- SUBRUTINAS --------------
ESPERA:
        CLRF CONTADOR
LAZO1:  DECFSZ CONTADOR, F
        GOTO LAZO1
        RETURN
end
end.
```

```
    list      p=16F84A
    #include <p16F84A.inc>
;===RAM usage===
STATUS EQU 0x003
#define STATUS_RP0 0x003,5
PORTA EQU 0x005
PORTB EQU 0x006
INTCON EQU 0x00B
OPTION_REG EQU 0x081
EECON1 EQU 0x088
contador EQU 0x00C
;------ Work and Aux. Registers ---
;===Blocks of Code===
    0x000 goto 0x001
__main_program__:
    0x001 bsf STATUS, 3
    0x002 clrf PORTB
    0x003 bcf STATUS, 3
    0x004 bsf PORTB, 0
    0x005 call 0x009
    0x006 bcf PORTB, 0
    0x007 call 0x009
    0x008 goto 0x004
    0x009 clrf contador
    0x00A decfsz contador,f
    0x00B goto 0x00A
    0x00C return
__end_program__:
```

Información / Advertencias / Errores

Iniciando compilación...
Compilado en: 16 msec. <<0 Advertencias, 0 Errores>>
RAM usada =1/136B (0.74%), Flash usada=14/1024 (1.4%)

RAM 1% ROM 1% STACK 0%

Guardado 30,15 utf8 D:\Proyectos_Software\Lazarus\PicPas-0.8.1\temp\NewFile1.pas

5.18Units

5.18.1 Use of units

Machine Translated by Google

The units to be used are included within the USES section of a program or from another unit. They are used as a library, as in the following example:

```
use PIC10F200; begin
while
   true do
      GPIO_GP0 := not GPIO_GP0;
      delay_ms(500); end;

end.
```

This program includes the PIC10F200 unit, which includes definitions that allow you to work with this microcontroller. Among the definitions are the hardware characteristics (RAM and ROM memory size, maximum CPU speed, implemented memory areas, etc.) and access variables to special registers such as STATUS, or GPIO.

It is always necessary to define the microcontroller; so that the compiler knows the characteristics of the target hardware, such as the size of RAM and ROM, or the number of banks or pages.

If the microcontroller is not specified, a generic device will be assumed, with the 'DEFAULT' model with characteristics that depend on the selected family:

• Low range: DEFAULT is equivalent to a PIC10F200. •
Mid-range: DEFAULT is equivalent to a PIC16F84.

5.18.2 Unit creation

PicPas supports the use of units, to a limited extent.

In PicPas units, constants, variables and procedures can be defined.

The units of PicPas are similar to those of Turbo Pascal:

```
unit PIC16F84A;
interface
var
   TMR0          :byte absolute $01; :byte
   PCL           absolute $02;
   STATUS :byte absolute $03;

procedure Proc1(x: char);

Implementation

procedure Proc1(x: char); begin x :=
0; end;
```

Machine Translated by Google


```
end.
```

The drive name must match the file name for the drive to be considered valid.

A unit can be compiled individually, but a *.hex file will not be generated, but rather will serve more as a check that the unit does not contain syntax errors.

A unit is compiled in the same way as a program.

Machine Translated by Google

5.19Pointers

PicPas includes basic support for pointer handling, within the language.

Pointers in PicPas have the following characteristics:

• They can store addresses ranging from $00 to $FF, which means that they cover RAM banks 0 and 1, but they cannot access variables from bank 2 and 3, if they exist in the microcontroller used.
• Only pointers to the byte and word type can be created. • Every pointer must have a pointed type. Cannot create pointers without guy.

5.19.1 Definition of pointers

Pointers are defined by first creating a type:

```
type
   ptrByte: ^Byte;
   ptrByte: ^Word;
var
   pbyte: ptrByte;
   pword: ptrWord;
```

Pointers can be assigned as byte variables, but they are commonly assigned with addresses of other variables. It is necessary that a pointer be first assigned before working with it:

```
type
   ptrByte: ^Byte;
var
   pbyte: ptrByte; : byte;
   m
begin
   @m; pbyte^ :=            //Assign pbyte address :=
   $ff; //Write value
   //Now "m" is $ff
end.
```

The @ operator returns the physical address of a variable. In the current version, it can only go from $00 to $FF, so it will only be able to cover variables from banks 0 a

To access the value pointed to by a pointer, the notation is used:

pointer_variable^

Which behaves like a byte variable.

Machine Translated by Google

5.19.2 Pointer Arithmetic

Pointers support some basic operations. These are:

OPERATION	EXAMPLE
Assignment	p1 := p2;
Comparison	if p1 = p2 then ...
Increase	Inc(p);
Decrement	Dec(p);
Addition	p1 + p2 + 1
Subtraction	p1 - 5

Additions and subtractions must be between pointers of the same type, or they must be between pointers and byte values.

When pointers with bytes are added or subtracted, the result of the expression is of the same type as the pointer. For example, if in the following expression:

```
$10 + p
```

“p” is of type “pointer to byte”, so the value of the expression “$10 + p” is also of type “pointer to byte”.

Machine Translated by Google

6 DIRECTIVES

Directives are a set of instructions, which are interpreted by the compiler, at the time the program is being compiled.

The compiler directives, included in PicPas. These are defined as comments, but they always begin with the characters "{$":

```
{$PROCESSOR PIC16F84} //Here is a directive program name; begin
x := {$value};

                    //Here too
end.
```

They can be included anywhere in the code, but for functionality, some should always be written at the beginning.

Directives are always executed at compile time, and their purpose
The main thing is to configure and give the final form to the code, which will finally be compiled.

6.1 Use of directives

Directives are typically used to define values that will later be used in the source code:

```
{$DEFINE output_pin=PORTB.0} uses
PIC16F84A; begin

SetAsOutput({$pin_output}); {$output_pin} :=
  1;
end.
```

And they are also used to determine the conditional and selective compilation of Pascal code:

```
{$IFDEF output pin}
  SetAsOutput({$pinOutput});
  {$pinOut} := 1; {$ELSE}

  SetAsOutput(PORTB.1);
  PORTB.1 := 1;
{$ENDIF}
```

In general the directives allow:

1. Set values for certain system variables.
2. Define symbols and macros.
3. Define conditional compilation blocks.

Machine Translated by Google

4. Generate messages in the build

Access to system variables allows you to configure system parameters. microcontroller before compilation.

Because of its ability to define macros and conditional compilation, it would seem that directives are treated with a pre-processor (as is done in C compilers), but this is not the case. It is the same PicPas compiler, which processes the directives in the same pass in which the compilation is done.

6.2 Directive processing

Directives are executed in the build phase. Therefore, the order in which they are processed is the same order in which the source code is compiled.

This means that directives are processed from the beginning of the main program file, through the contents of all units, recursively.

For example, if we have the following program:

```
{$DEFINE macro1}
program name; use
my_drive; begin
value :=
   0; end.
```

The compiler will linearly explore the program from the beginning, first executing the {$DEFINE macro1} directive and then continuing to explore the code until it finds the USES statement. At that moment it will begin to explore the unit “my_unit”, inside which “macro1” can be accessed, because it has already been recognized by the compiler.

Likewise, after the USES statement, in the main program, you can access all the macros or variables that have been declared within “my_unit”.

Also consider that directives found within procedures, not used or called, will also be processed, because the compiler always explores all procedures.

The following code illustrates this case:

```
program name;

procedure proc_not_used; begin

   //This macro will always be created
   {$DEFINE macro=value}
```

Machine Translated by Google


```
end;

begin

end.
```

Machine Translated by Google

6.3 The language of directives

Directives can be understood as a simple programming language (with its assignment and conditional statements), language independent Pascal and which is interpreted at compile time.

The main characteristics of the directive language are:

- It is not sensitive to the text box. Upper or lower case can be used indistinctly.
- The instructions always occupy a single line, and are always found delimited by {$…}
- It is not a typed language. Variables can change type and interact between them.
- It is not necessary to declare variables before using them.
- There are only two types of data: numbers and strings.

It can be said that the language of the directives is quite similar to interpreted or script languages.

6.3.1 Data types

There are only two types of data in directives:

DATA TYPE	DESCRIPTION NUMBERS	EXAMPLE
Numeric values integers or	floating point.	$SET x = 3.14159 $SET x = -2000 $SET x = $FFFF
CHAINS	Character set.	$SET cad = 'hello' $SET cad = “hello world”

The boolean type does not exist within directives. For the case of the comparisons, such as:

X>0
string='hello'

The result is the number 0, when the expression is false and 1 when the expression is false. expression is true.

In general, the following rule applies to evaluate an expression, as value boolean:

Any numerical expression other than zero, or any non-null string expression, is considered TRUE.
Any other value will be considered false.

String constants are defined between single or double quotes:

'I am a chain'
"Me too"
"Me too, and I have a line break\n"

Strings enclosed in double quotes have the peculiarity that they accept escape sequences:

SEQUENCE	MEANING
\n	Operating system line break (for Windows is CR+LF, and for Linux it is LF)
\r	LF character ($0D)
\t	Tabulation

6.3.2 Variables

Variables are assigned with the $SET statement:

```
{$SET x = 1}
{$SET y = 1 + x}
{$SET x = 'now I am a string'}
```

$SET is not a declaration statement, but an assignment one. The first time When a value is assigned to a variable, it is created.

It is not necessary to declare a variable before using it, but if you try to read one variable, not created, you will get an error:

```
{$SET y = 1 + x} //gives an error, if “x” has not been created
```

The content of a variable can be displayed using instructions like $MSGBOX or $INFO:

```
{$MSGBOX 'x is worth = ' +x}
```

Variables can be used within expressions, together, with the operators and functions.

6.3.3 Operators

OPERATION	DESCRIPTION

Machine Translated by Google

OR	
'+'	Sum of two numbers: 1 + 2 = 20 String concatenation: '1' + '2' = '12' Only when the two addends are numbers, an addition is made, so Otherwise they are concatenated as a string.
'-'	Subtraction of two numbers: 5 - 2 = 3
'*'	Multiplication of two numbers: 2 * 5 = 10
'/'	Division of two numbers: 5 / 2 = 2.5
'\'	Integer division of two numbers: 5 \ 2 = 2
'%'	Remainder of a division of two numbers: 5% 2 = 1
'^'	Power of a number: 2^3 = 8
=,<>	Compare numbers or strings. String comparison is considering the box.
>,<,>=,<=	Compares numbers or strings. To compare strings, they must have the same length. The comparison is made character by character, until find a larger ASCII code.

6.3.4 Operator precedence

OPERATOR	PRECEDENCE
=, <>, >, <, >=, <=	4
'+',	5
'-' '*', '/', '\', '%'	6
'^'	8

The operators with the highest precedence will be executed first in an expression. To change precedence, you can group expressions using parentheses.

6.3.5 Functions

Within the directive language, there is a set of functions already defined, that can be used to handle numbers or strings.

abs()	DESCRIPTION
	Returns the absolute value of a number. It has the following syntax: abs(<number>) Example: {$MSGBOX abs(-1)} It will show the number 1.
sgn()	Returns 1 if the value is positive and -1 if the value is negative. Returns zero if the value is zero. Example: {$MSGBOX sgn(-1000)}

Machine Translated by Google

	It will display the
without()	value -1 Returns the sine of an angle in radians. Example: {$MSGBOX sin(3.1415)} It will show a value close to zero.
cos()	Returns the cosine of an angle in radians. Example: {$MSGBOX cos(0)} It will show the number 1.
so()	Returns the tangent of an angle in radians. Example: {$MSGBOX tan(0)} It will show the number 0.
log()	Returns the natural logarithm of a number. {$MSGBOX log(2,718)} It will show a number close to 1.
round()	Returns rounded value of a number. Example: {$MSGBOX round(2.6)} It will show the number 3. {$MSGBOX round(2.1)} It will show the number 2.
trunc()	Returns the integer part of a number. Example: {$MSGBOX trunc(2.9)} It will show the number 2.
length()	Returns the size of a string, in bytes. Example: {$MSGBOX length('Hello World')} It will show the value 10. {$MSGBOX length('you')} It will show the value 3, because internally PicPas works with UTF-8 encoding, where the accented Latin characters and the “ñ” occupy two bytes. upcase()
Returns an uppercase string.	Example: {$MSGBOX UpCase('Hello World')} It will show the string: “HELLO WORLD”
lowcase() Returns a lowercase string.	Example: Example: {$MSGBOX LowCase('Hello World')} It will show the string: “hello world”

6.4 System variables

Machine Translated by Google

There are certain variables already created, which allow access to parameters of the system. These are:

PIC_FREQUEN	Returns the frequency of the microcontroller, defined with {$FREQUENCY ...} Example: {$MSGBOX pic_frequen} Shows the current PIC frequency. The frequency is expressed in Hertz. For example, if the frequency is 8MHz, the value 8000000 will be displayed. Changing the PIC_FREQUEN value is equivalent to changing it using {$FREQUENCY }, considering that {$FREQUENCY } recognizes units such as KHz and MHz.
PIC_MODEL	Returns or sets the microcontroller model, defined with {$PROCESSOR ...} Example: {$MSGBOX pic_model} It will show the current processor model, which will be the model that was previously defined with the {$PROCESSOR} directive, or the default model.
PIC_MAXFREQ	Returns or sets the maximum frequency of the current microcontroller. For example, to set the maximum frequency to 1Mhz, you would do: {$SET PIC_MAXFREQ = 1000000} The frequency value is specified in Hertz. Setting a maximum frequency will not allow you to configure higher frequencies with the {$FREQUENCY} directive.
PIC_NUMBANKS	Returns or sets the number of RAM memory banks of the current microcontroller. For example: {$SET PIC_NUMBANKS=4} Set 4 banks for the current microcontroller. From one to four banks of RAM can be set.
PIC_NUMPAGES	Returns or sets the number of code pages for the current microcontroller. For example: {$SET PIC_NUMPAGES = 2} Sets 2 pages of program memory, for the current microcontroller. They can be set from one to four pages.

Machine Translated by Google

PIC_IFLASH	Returns or sets the current address in Flash memory or EEPROM, where the next instruction to be compiled will be written. When starting the compilation, the value of PIC_IFLASH is zero, and then it increases as instructions are written to the FLASH/ EEPROM memory of the microcontroller. For example: {$MSGBOX PIC_IFLASH} Displays the current value of PIC_IFLASH, at the current position, from the source code[9] . The following code can be used to calculate the amount of program memory used to compile an instruction: {$SET n1 =PIC_IFLASH} x := x + 1; {$SET n2 =PIC_IFLASH} {$MSGBOX n2-n1}
PIC_NPINS	Returns or sets the number of pins that the current microcontroller physically has. The number of physical pins that the microcontroller has does not affect the program or the compiler itself in any way. Rather, it is used as part of the information used by the integrated PicPas Simulator, for when it must physically graph the device[10] .
PIC_ENHANCED	Boolean variable. It is only defined when using the low-end PIC (Baseline) compiler. Indicates whether the current device is from the “Baseline Enhanced” family. This means that you have a broader set of instructions.
PIC_MAXFLASH	Returns or sets the maximum amount of FLASH or EEPROM memory to hold the program. For example: {$MSGBOX PIC_MAXFLASH} Shows the number of program memory cells available to the current microcontroller. The value of PIC_MAXFLASH can range from 0 to the maximum value allowed by the number of memory pages implemented. Thus, if you have 4 pages of program memory, PIC_MAXFLASH can be up to 8192, since each page can cover up to 2K of code.
SYN_MODE	Returns the compiler's syntax mode, defined with {$MODE ...}. It is read-only.
CURRBANK	Returns or sets the RAM bank, which the compiler assumes the PIC should have at that point (where the

[9] The value of PIC_IFLASH, which can be read within the directives, is only referential, because the real value of the pointer that the compiler uses to generate code is defined in the linking phase, while the directives are executed in the compilation phase, where PIC_IFLASH is reset when each procedure is compiled, and in the body of the program.

[10] This functionality is not active in the current version of PicPas.

Machine Translated by Google

	directive)11 . Example: {$MSGBOX CURRBANK} Shows the value of the current bank. The value of CURRBANK is a variable used by the compiler to determine when to generate change instructions of RAM banks. The compiler updates CURRBANK, as it goes accessing records from different banks. Although it can be modified, it is not convenient because it can produce compilation errors.

CURRBLOCK	Returns the current syntax block in which the compiler.
	Example:
	begin
	  {$MSGBOX CURRBLOCK}
	end;
	
	Returns a null string, because there is no block. But in the following example:
	
	begin
	  if x>1 then
	    {$MSGBOX CURRBLOCK}
	  end;
	end;
	
	The value 'sbiIF' will be displayed, which indicates that you are within a IF block.
	The possible values for CURRBLOCK are:
	
	• sbiIF • sbiFOR • sbiWHILE • sbiREPEAT

Most of these variables can be read or modified. Consider Modifying critical system variables could cause compilation errors.

11 The value of CURRBANK, which can be read within the directives, is only referential, because the real value that the compiler uses to generate code is defined in the linking phase, while the directives are executed in the compilation.

Machine Translated by Google

6.5 List of directives

6.5.1 $PROCESSOR

This directive is a short alternative to define the hardware of a microcontroller. The long form is the one indicated in section 6.7, but it is not usually used because the hardware is defined in the units that come with the compiler.

{$PROCESSOR} Allows you to define the microcontroller that will be used to compile the code. When used, it should always be included before starting the program. For example:

```
{$PROCESSOR PIC16F84}
program name; begin


end.
```

The list of microcontrollers supported, through this directive, includes:

LOW RANGE:

PIC10F200 PIC10F202 PIC10F204 PIC10F206

GAME MEDIA:

PIC12F629 PIC12F675 PIC12F629A PIC12F675A

PIC16F83 PIC16CR83 PIC16F84 PIC16CR84 PIC16F84A

PIC16F870 PIC16F871 PIC16F872 PIC16F873 PIC16F873A PIC16F874 PIC16F874A PIC16F876 PIC16F876A PIC16F877 PIC16F877A PIC16F887

PIC16F627A PIC16F628APIC16F648A

Indicate the microcontroller with {$PROCESSOR}, it does not define memory like STATUS or TMR0, but simply the hardware characteristics.

The {$PROCESSOR} directive, however, is not necessary when using units that define the microcontroller. For example, the following code does not require using the {$PROCESSOR PIC16F877} directive, because it is already included in the PIC16F877 unit:

```
program Test; use
PIC16F877;
var
   a, b, c, d: byte; begin

      ...
```

Machine Translated by Google


```
end.
```

In general, this way of working is preferred, because the unit not only defines the hardware to be used, but also defines variables for the GPR memory, such as STATUS, PORTA, TRISB,…. Additionally, there are more devices defined by units than those supported by {$PROCESSOR} (See Appendix).

6.5.2 $FREQUENCY

Defines the frequency of the clock (of the crystal or internal oscillator) at which the microcontroller. It must always be indicated before starting the program. For example:

```
{$PROCESSOR PIC16F84}
{$FREQUENCY 8Mhz}
program name; begin


end.
```

Frequencies can be expressed in KHz or MHz. For example, 1000Khz or 1Mhz.

The clock frequency information is used by the compiler internally to:

- Time the delays, using the delay_ms() routine.
- Timing some internal hardware devices, such as serial communication[12] .
- To correctly time the simulation in real time.

If none of these items are used, the clock rate can be almost any value that can be expressed as integer KHz or MHz values.

But if the program will use delay instructions or handling hardware that requires timing, then the compiler will only accept some frequency values. These values are:

1MHz, 2Mhz, 4Mhz, 8MHz, 10MHz, 12MHz, 16MHz or 20MHz.

If the frequency is not specified, 4Mhz is assumed by default.

6.5.3 $MODE

The MODE directive allows you to specify the way PicPas will work, with respect to the syntax used.

It can have two values:

[12] Not implemented yet, in the current version of PicPas.

Machine Translated by Google


```
{$MODE PICPAS}
{$PASCALMODE}
```

PICPAS mode is the default, and indicates that it will be compiled using the normal PicPas syntax. This syntax takes elements (just some) from the Modula-2 programming language.

In this mode of syntax, the control structures are different from those of Standard Pascal. These have the following syntax:

```
IF <condition> THEN
    <block of code>
END;

IF <condition> THEN
    <block of code>
ELSE
    <block of code>
END;

IF <condition> THEN
    <block of code>
ELSIF <condition> THEN
    <block of code>
ELSE
    <block of code>
END;

WHILE <condition> DO
    <block of code>
END;

REPEAT
    <block of code>
UNTIL <condition>;

FOR <variable> := <start-value> TO <end-value> DO
    <block of code>
END;
```

PASCAL mode works with the normal Pascal syntax, with respect to control structures, but some non-standard Pascal features are maintained, such as bit variables, or properties of basic types (See section 5.8.4).

To work in Pascal mode, {$MODE PASCAL} must necessarily be specified.

6.5.4 $MSGBOX

Displays a message on the screen during the compilation process:

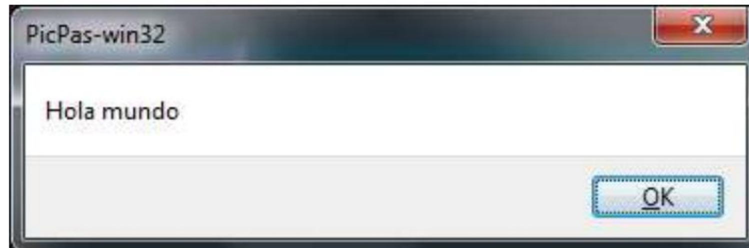
```
program name; begin
```

Machine Translated by Google


```
    {$MSGBOX 'Hello world'}

end.
```

The box will be displayed on the screen:



Strings must be enclosed in single quotes.

You can also use expressions:

```
    //Sample 9
    {$MSGBOX (1+2)*3 }

    //Shows the PIC frequency
    {$MSGBOX 'clock=' + PIC_FREQUEN }
```

6.5.5 $MSGERR

Displays an error message on the screen during the compilation process:

```
program name; begin

{$MSGERR 'Something bad happened :('} end.
```

The box will be displayed on the screen:

Error messages are displayed with an error icon.

Machine Translated by Google

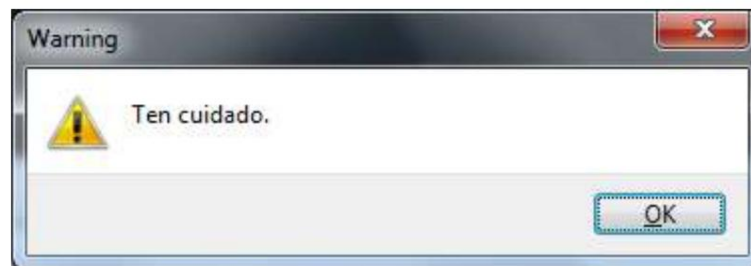
6.5.6 $MSGWAR

Displays a warning message on the screen during the compilation process:

```
program name; begin

{$MSGWAR 'Be careful.'}
end.
```

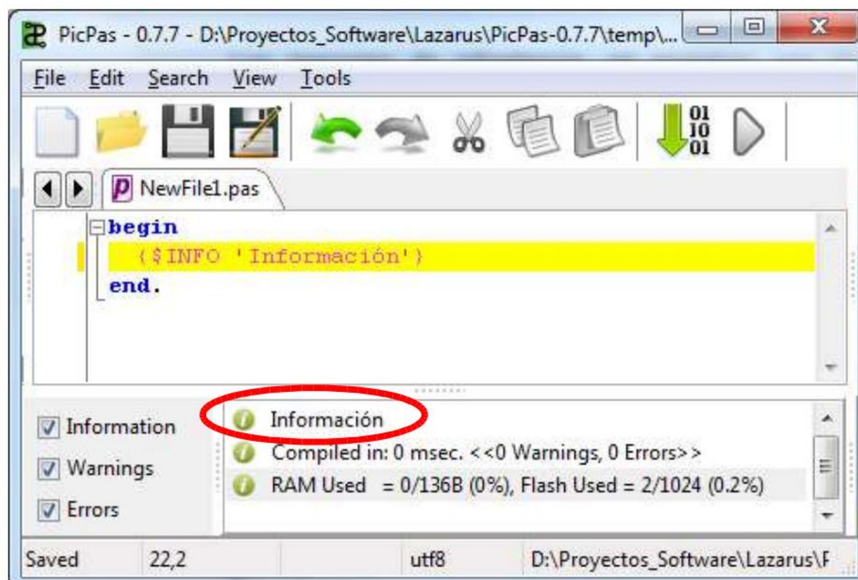
The box will be displayed on the screen:



Warning messages are displayed with an indicative icon.

6.5.7 $INFO

Displays an information message in the Message Panel:

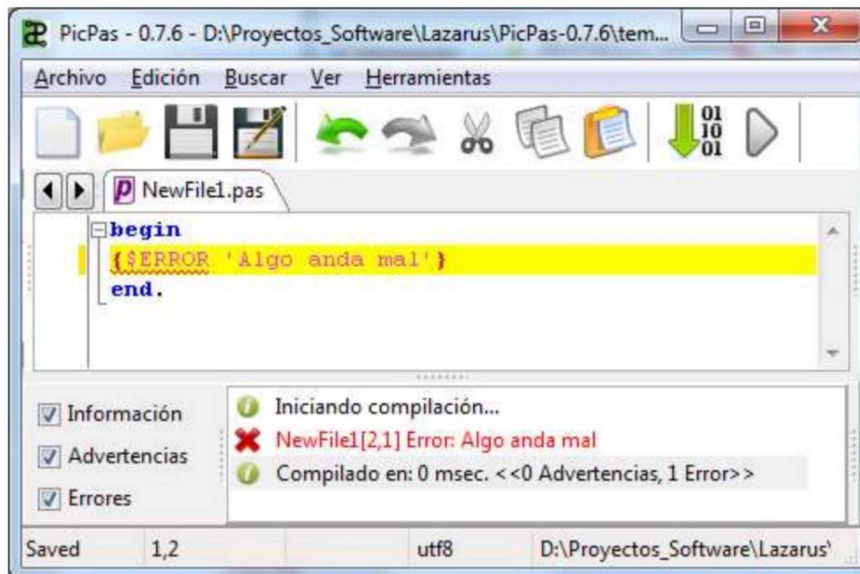


The message will appear when the program is compiled.

6.5.8 $ERROR

Machine Translated by Google

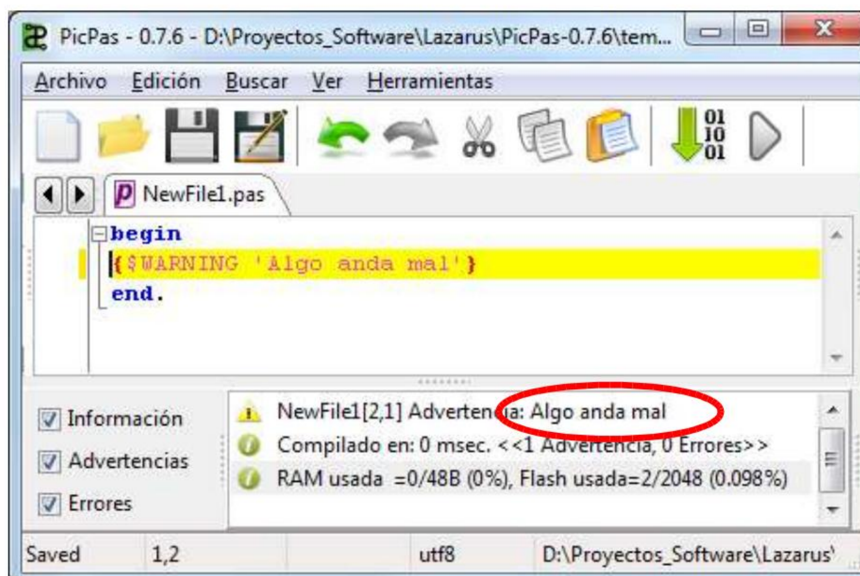
Display an error message in the Message Panel:



The message will appear when the program is compiled.

6.5.9 $WARNING

Displays a warning message in the Message Panel:



The message will appear when the program is compiled.

6.5.10 $CONFIG

Defines the configuration bits of the current device.

The basic form of the syntax is:

Machine Translated by Google

{$CONFIG <numeric value>}

Where <numeric value> can be any numeric value, in the range $0000 to $FFFF.

Some examples are:

```
{$CONFIG $3FFD}
{$CONFIG 16578}
```

The other form of the syntax is:

{$CONFIG <tag list>}

The list of tags must be a list of macros defined with {$DEFINE} and that represent binary values. This implies that before using $CONFIG in this way, You must first have the tag definitions:

```
{$define _CP_ON {$define            =     0x000F}
_CP_OFF {$define _WDT_ON            =     0x3FFF}
{$define _WDT_OFF {$define          =     0x3FFF}
_LP_OSC {$define _XT_OSC            =     0x3FFB}
{$define _HS_OSC                    =     0x3FFC}
                                    =     $3FFD}
                                    =     $3FFE}

{$CONFIG _CP_OFF, _XT_OSC, _WDT_OFF }
```

Tags must be separated by commas or spaces.

Note that hexadecimal notation for the definition of macros, you can use the character “0x”, or the character “$”, interchangeably.

The configuration bits are included in the output *.hex file, as a value at address 0x2007. If no {$CONFIG} directive is indicated in the code, no information is generated at this address.

6.5.11 $INCLUDE

Includes code from an external file.

The syntax is:

{$INCLUDE <file to include>}

The following are valid examples of using $INCLUDE:

Machine Translated by Google


```
{$INCLUDE yyy.pas}
{$INCLUDE d:\long_path_to_my_file\yyyy.txt}
```

If the full path is not specified, it is assumed that the file to be included is located in the same path as the file being compiled.

Including a file with {$INCLUDE } is equivalent to including all the text of that file. file, at the point where the call to {$INCLUDE } is made.

The file to include can be of any type, not necessarily Pascal code.

The {$INCLUDE} directive can be placed almost anywhere in the program, for example, in the middle of expressions:

```
var
   x: byte;
begin
   x := {$INCLUDE expression.txt}; end.
```

And this code will be correct if the content of expression.txt is a valid expression for the assignment.

6.5.12 $OUTPUTHEX

Allows you to specify the name of the output file *.hex.

By default, the binary file will take the name of the Pascal program that is compiled, but with the *.hex extension. So if the main program is called "hello. pas", the output file will take the name "hello.hex"

The $OUTPUTHEX directive allows you to define the name for the *.hex file.

The syntax is:

{$OUTPUTHEX <filename *.hex>}

The following are valid examples of using $OUTPUTHEX:

```
{$OUTPUTHEX "output.hex"}
{$OUTPUTHEX "d:\output.hex"}
```

If the full path is not specified, it is assumed that the *.hex file is located in the same path as the file being compiled.

Machine Translated by Google

The {$OUTPUTHEX} directive can be placed anywhere in the source code, and even multiple times. If $OUTPUTHEX is used multiple times, the *.hex file name will be the last one defined.

6.5.13 $DEFINE

Defines a symbol or macro.

To define a symbol, the syntax is used:

{$DEFINE <identifier>}

Where <identifier> can be any valid identifier starting with an alphabetic character (A-Za-z_) followed by any alphanumeric character (A-Za-z0- 9_)

The following are examples of symbol definitions:

```
{$DEFINE level}
{$define cpu_best}
{$define HAS_ADC}
```

A defined symbol is stored in memory throughout the entire creation process. compilation, and can be verified with the {$IFDEF} directive

The scope of the defined symbols is global. Once defined, they are accessible by the compiler from procedures and even from within units.

Let's consider the following code:

```
program test;
 {$DEFINE port=123} use
my_drive; begin end.
```

In this code, the “port” symbol is accessible throughout the “test” program. (because it has been defined at the beginning), and also from within the unit “my_unit”.

However, in the following code:

```
program test; use
my_drive; {$DEFINE
 port=123} begin end.
```

Machine Translated by Google

The “port” symbol is not accessible from within the “my_unit” unit, because it has been declared later.

The symbols can be used with the $MSGBOX instruction, so that the instruction:

```
{$msgbox $symbol}
```

Will show an empty string, if the symbol has been defined, but will show the string “$symbol”, if not defined.

The $DEFINE directive also allows defining macros.

A macro is a symbol, to which a value is assigned.

To define a macro, use the syntax:

{$DEFINE <identifier>=<content>}

Where <content> is any block of text that does not include the character “}”, because this is considered a delimiter.

The following are examples of macro definitions:

```
{$DEFINE port=PORTB}
{$DEFINE value = $ff}
{$DEFINE sum=a+b}
{$DEFINE sum=a+b}
{$DEFINE block = begin a:=1; b := 0; end}
```

The same rules for the scope of symbols apply to the definition of macros.

The definition of a macro can only occupy one line.

Once a macro is defined, its value can be accessed in the source code by writing the directive {$<macro name>}, as shown in the following example:

```
{$DEFINE output_pin=PORTB.0} uses
PIC16F84A; begin

SetAsOutput({$pin_output}); {$output_pin} :=
    1; end.
```

The content of a macro can be displayed with the $MSGBOX instruction, so that the instruction:

Machine Translated by Google


```
{$DEFINE macro="Hello"}
{$MSGBOX $macro}
```

It will display the string “Hello”.

Note that the “$” symbol is not included in the definition of a macro, but it is used to reference the macro, from outside the definition.

6.5.14 $IFDEF

Allows you to define conditional compilation blocks.

The $IFDEF directive evaluates the existence of some macro, or variable, and compiles or skips compiling blocks of code accordingly.

The most common syntax of the directive is:

```
{$IFDEF <identifier>}
...
{$ENDIF}
```

But the form can also be used:

```
{$IFDEF <identifier>}
...
{$ELSE}
...
{$ENDIF}
```

The following code shows an application case:

```
{$DEFINE output pin=PORTB.0} uses
PIC16F84A; begin
{$IFDEF
 outputpin}
   SetAsOutput({$pinOutput}); {$pinOut} :=
    1; {$ELSE}

   SetAsOutput(PORTB.1);
   PORTB.1 := 1;
 {$ENDIF}
end.
```

The instruction {$IFDEF pinOut} checks whether the macro $pinOut is defined, and if so, compiles all the text between {$IFDEF pinOut} and {$ELSE}, omitting the text between {$ELSE} and {$ ENDIF}. If not defined, the alternate block is compiled.

Machine Translated by Google

You could also write the previous code in the form:

```
{$DEFINE outputpin=PORTB.0} uses
PIC16F84A; begin
{$IFDEF
 outputpin}
 {$ELSE}
     {$DEFINE output pin=PORTB.1}
 {$ENDIF}
    SetAsOutput({$pinOutput}); {$pinOut} :=
    1; end.
```

6.5.15 $IFNDEF

It is the negated version of the $IFDEF directive.

As an application example, the same code as the previous example is shown, but using {$IFNDEF}

```
{$DEFINE outputpin=PORTB.0} uses
PIC16F84A; begin

{$IFNDEF outputpin}
     {$DEFINE output pin=PORTB.1}
 {$ENDIF}
    SetAsOutput({$pinOutput}); {$pinOut} :=
    1; end.
```

6.5.16 $IF

It allows defining conditional compilation blocks, according to an expression.

The $IF directive evaluates an expression, and compiles, or skips compiling, blocks of code accordingly.

The most common syntax of the directive is:

```
{$IF <expression>}
...
{$ENDIF}
```

But the form can also be used:

```
{$IF <expression>}
```

Machine Translated by Google

...
{$ELSE}
...
{$ENDIF}

The following code shows an application case:

```
{$IF value>255} var x:
word;
{$ELSE}
var x: byte;
{$ENDIF}
```

Since there are no boolean variables in directives. Comparison operators, such as = or <>, return the number 1 if the expression is true and 0 when it is false.

On the other hand, the {$IF} statement will consider as TRUE, any non-zero numeric expression, or any string expression, not null.

6.5.17 $IFNOT

It is the inverted version of $IF:

```
{$IFNOT value>255} var x:
byte;
{$ELSE}
var x: word;
{$ENDIF}
```

6.5.18 $SET

Allows you to define variables within the directives.

The directive variables are independent of the program variables in Pascal.

```
{$SET num = 1}                    //Now "num" exists and is equal to 1
{$SET value = num + 1} //Now "value" exists and is worth 2 {$SET name = 'Pedro'} //
Now "name" exists and is worth 'Pedro'
'Pedro'}                          //Now "num" is a string {$SET num =
{$SET num = new + 1} //Error: "num" does not exist.
```

Variables can be of two types: numbers or strings.

Machine Translated by Google

Variables are assigned with the $SET directive. If the variable does not exist, it is created. If the variable already exists, its value (and its type, if different) is updated.

All variables are removed at the start of the build (Except variables of the system).

Trying to access an uncreated variable will generate an error.

6.5.19 $SET_STATE_RAM

Configures the RAM memory status of the current device.

The state of a byte in RAM can have three values:

SFR	Special function registration. As in the case of STATUS or TRISB.
GPR	General purpose registry. It is a record intended to be used as free memory for the user.
NIM	Registry not implemented.

$SET_STATE_RAM, allows you to define the state of the RAM, specifying a address range.

The syntax of $SET_STATE_RAM is:

{$SET_STATE_RAM <command list>}

Commands are separated by commas. A command has the form:

<start address>-<end address>:<state>

A valid example would be:

```
{$SET_STATE_RAM '000-00B:SFR'};
```

What this command indicates is that you want to define the RAM addresses, between 0x000 and 0x00B, as special function registers.

Addresses are always expressed in hexadecimal, with three digits. The Directions are expressed linearly covering up to bank 3, in the form following:

RANGE	BANK
000-07F	Bank 0
080-0FF	Bank 1
100-17F	Bank 2

Machine Translated by Google

180-1FF	Bank 3

Any other value, outside the range of valid addresses, will generate a compile-time error.

There is no point in assigning state to banks of RAM, which do not exist in the current device. However, the compiler will not generate errors.

Other valid ways would be:

```
{$SET_STATE_RAM '000-00B:SFR, 00C-04F:GPR'}
{$SET_STATE_RAM '080-08B:SFR, 08C-0CF:GPR'}
```

You can express as many ranges as you want in the same $SET_STATE_RAM directive.

At the start of compilation, all memory locations start with the status “Not implemented”.

With each $PROCESSOR instruction, the state of the RAM memory is defined, according to the chosen PIC model. But this state can then be changed with $SET_STATE_RAM.

The purpose of $SET_STATE_RAM is to define custom models, or not existing microcontrollers, which cannot be defined with $PROCESSOR.

The compiler accesses the RAM state, to calculate the available RAM space and reserve space for variables.

Changing the state of RAM affects the compiler's workspace, potentially generating compilation or execution errors.

6.5.20 $SET_MAPPED_RAM

Defines the mapped regions of the RAM memory of the current device.

The RAM of the device may be implemented at an independent address or may be mapped to another address in some other bank of RAM.
Such is the case of the memory addresses of the STATUS or INTCON registers, which are mapped to all RAM memory banks.

Machine Translated by Google

$SET_MAPPED_RAM, allows you to define mapped regions of RAM, in GPR and SFR records. It makes no sense to define RAM-mapped zones. implemented.

The syntax of $SET_MAPPED_RAM is:

{$SET_MAPPED_RAM <command list>}

Commands are separated by commas. A command has the form:

<start address>-<end address>:<destination bank>

The destination bank can be: bnk0, bnk1, bnk2 or bnk3

A valid example would be:

```
{$SET_MAPPED_RAM ' 080-080:bnk0'};
```

What this command indicates is that you want to define the RAM address, 0x080, as a register mapped to bank 0, and it follows that it must be in the address 0x00.

Addresses are always expressed in hexadecimal, with three digits. The Addresses are expressed linearly covering all banks of the device, so the following way:

MID-RANGE PIC MICROCONTROLLERS:

RANGE	BANK
000-07F	Bank 0
080-0FF	Bank 1
100-17F	Bank 2
180-1FF	Bank 3

LOW END MICROCONTROLLER

RANGE	BANK
000-01F	Bank 0
020-03F	Bank 1
040-15F	Bank 2
060-07F	Bank 3
080-09F	Bank 4
0A0-0BF	Bank 5
0C0-0DF	Bank 6
0E0-0FF	Bank 7

Machine Translated by Google

Any other value, outside the range of valid addresses, will generate a compile-time error.

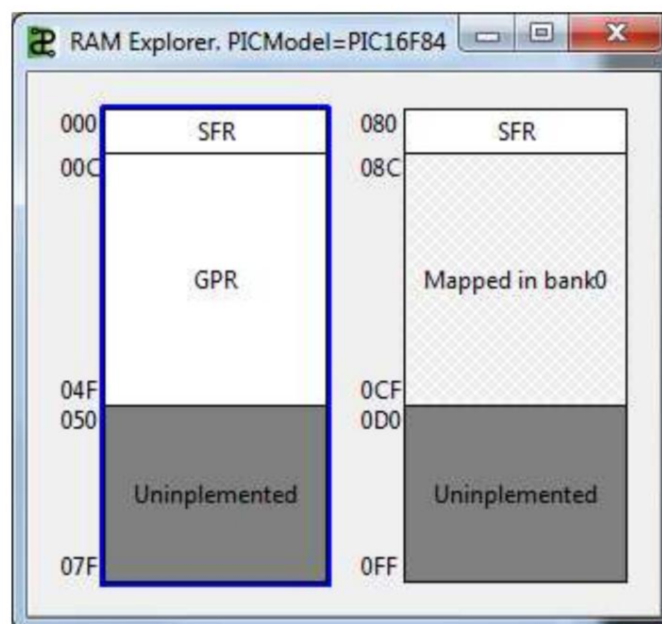
There is no point in mapping non-existing RAM banks (or to RAM banks) on the current device. However, the compiler will not generate errors.

Other valid ways would be:

```
{$SET_MAPPED_RAM '080-080:bnk0, 082-084:bnk0, 08A-08B:bnk0'}
{$SET_MAPPED_RAM '100-100:bnk0, 102-104:bnk0'}
```

You can express as many ranges as you want in the same $SET_MAPPED_RAM directive.

Mapping a byte or range of bytes to another bank implies that those addresses are not physically implemented, and all read or write operations will go to another bank of RAM. For example, it is known that in the PIC16F84, the memory addresses between 0x08C and 0x0CF are mapped to bank 0.



This means that accessing position 0x08C is equivalent to accessing address 0x00C. To achieve this configuration, you can use this directive:

```
{$SET_MAPPED_RAM '08C-0CF:bnk0'}
```

At the start of compilation, all memory locations start with the status “Unmapped”.

Machine Translated by Google

With each $PROCESSOR instruction, the mapped RAM areas are defined, so according to the chosen PIC model. But this state can then be changed with $SET_MAPPED_RAM.

The purpose of $SET_MAPPED_RAM is to define (along with $SET_STATE_RAM) custom or non-existing microcontroller models, that cannot be defined with $PROCESSOR.

The compiler checks the mapped areas of RAM to calculate the space of available RAM and reserve space for variables.

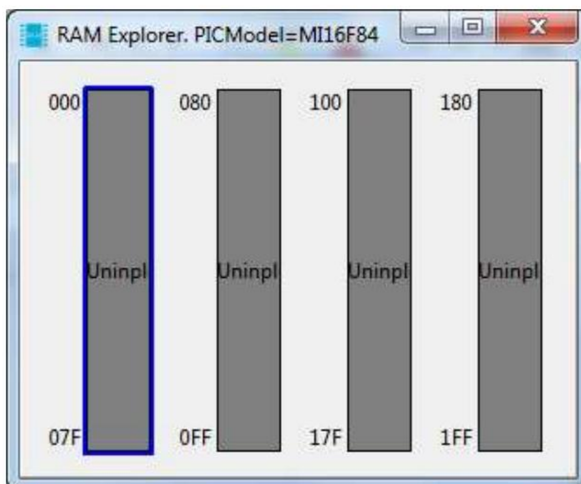
Changing the RAM mapped areas affects the compiler's workspace, potentially generating compilation or execution errors.

6.5.21 $CLEAR_STATE_RAM

It is used to start the RAM status.

$CLEAR_STATE_RAM, has the effect of setting the entire memory range addressable, in a state of <<Not Implemented>>, clearing any previous configuration.

Graphically, its effect would be:



It is used when you are going to start defining the RAM memory of a device, using the $SET_STATE_RAM and $SET_MAPPED_RAM directives.

6.5.22 $RESET_PINS

Clean the microcontroller pin configuration:

Machine Translated by Google


```
{$RESET_PINS}
```

This instruction is usually done before starting to define the names of the pins with the $SET_PIN_NAME directive.

It is important to clean the pin configuration before starting to define them, because otherwise some other previous configuration may be fixed.

Configuration cleaning is done based on the number of pins that have been configured. indicated in the PIC_NPINS system variable, therefore it must be defined first this variable:

```
{$SET PIC_NPINS =8}
{$RESET_PINS}
```

6.5.23 $SET_PIN_NAME

Allows you to define a name for a physical pin of the microcontroller.

This configuration allows you to see a descriptive label on the corresponding pin of the microcontroller, when it is shown in the diagram in the control window. simulation.

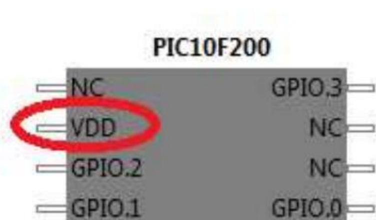
The syntax of $SET_PIN_NAME is:

{$SET_PIN_NAME <pin number>:<name>}

A valid example would be:

```
{$SET_PIN_NAME '2:VDD'}
```

This definition would place the label “VDD” on pin 2 of the PIC package and On the simulator diagram, it would look like this:



The PIN number can be given in decimal or hexadecimal.

Machine Translated by Google

The maximum size for a Pin name is 32 characters.

6.5.24 $MAP_RAM_TO_PIN

It is used to map RAM ports to physical microcontroller pins.
It allows mapping the bits of the GPIO, PORTA, PORTB, etc. ports to physical pins of the device.

This configuration is necessary to carry out the simulation graphically, so that the program knows which RAM positions write or read on the microcontroller pins.

The syntax of $MAP_RAM_TO_PIN is:

{$MAP_RAM_TO_PIN <address>:<association list>}

Associations are separated by commas. An association has the form:

<bit number>-<pin number>

A valid example would be:

```
{$MAP_RAM_TO_PIN '005:0-17,1-18,2-1,3-2,4-3'};
```

What this command indicates is that bits 0, 1, 2, 3 and 4, of address $05, will be mapped to pins 17, 18, 1, 2 and 3 respectively.

The values of the bit number and pin number are given in decimal.

An additional effect of using $MAP_RAM_TO_PIN is that the affected pins will be assigned names, similar to using the $SET_PIN_NAME directive.

The name that will be associated with the pin will be of the form PORTA.0, PORTA.1, ... which consists of joining the name of the memory address with the bit number. The name of the memory address corresponds to the one defined when a variable is declared like:

```
PORTA: byte absolute $0005; //Assign the name "PORTA" to address $005
```

If no name has been declared for the memory address, the generic name “PORT” will be used. So, for example, the following code:

```
{$RESET_PINS}
{$SET PIC_NPINS = 8}
{$MAP_RAM_TO_PIN '005:0-1'}
```

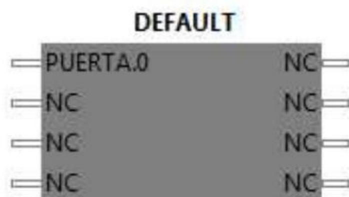
Machine Translated by Google

It will produce, in the simulator, the following configuration:

While the following configuration:

```
{$RESET_PINS}
{$SET PIC_NPINS var        =8}
DOOR: byte absolute $05;
{$MAP_RAM_TO_PIN '005:0-1'}
```

Will produce:

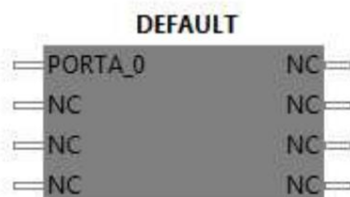


Because you already have a name for address $05 before executing the $MAP_RAM_TO_PIN instruction.

If you now want to change the final name of the pin, you can execute the $MAP_RAM_TO_PIN instruction:

```
{$RESET_PINS}
{$SET PIC_NPINS var        =8}
DOOR: byte absolute $05;
{$MAP_RAM_TO_PIN '005:0-1'}
{$SET_PIN_NAME '1:PORTA_0'}
```

The design remains as follows:



Changing the name of a PIN with $SET_PIN_NAME does not change at all the internal name that is assigned to a RAM memory location. It's just a label for the graphical interface.

Machine Translated by Google

6.5.25 $SET_UNIMP_BITS

Allows unimplemented bits to be defined in specific positions in RAM.

This setting is used to more accurately model the RAM of a device, so that the simulation of the program is done in a more real way.

The syntax of $SET_UNIMP_BITS is:

{$SET_UNIMP_BITS <command list>}

Commands are separated by commas. A command has the form:

<address>:<mask>

The address and mask are expressed in 3- and 2-digit hexadecimal. respectively.

A valid example would be:

```
{$SET_UNIMP_BITS '005:1F'};
```

Which indicates that bits 5, 6 and 7 of position $005 (PORTA) are not implemented in the hardware, because the value 1F, in binary, has these bits set to zero.

A bit set to not implemented, with $SET_UNIMP_BITS, means that it will always be read as zero.

The libraries that define the various PIC models use this directive internally to define the hardware precisely.

6.5.26 $SET_UNIMP_BITS1

Allows unimplemented bits to be defined in specific positions in RAM.

This command works similarly to $SET_UNIMP_BITS, except that unimplemented bits will always be read as 1, instead of 0.

The syntax of $SET_UNIMP_BITS1 is:

{$SET_UNIMP_BITS1 <command list>}

A valid example would be:

```
{$SET_UNIMP_BITS1 '004:E0'};
```

Machine Translated by Google

Which indicates that bits 5, 6 and 7 of position $004 are not implemented in the hardware, because the value E0, in binary, has these bits set to one.

A bit set to not implemented, with $SET_UNIMP_BITS1, means that it will always be read as one.

Machine Translated by Google

6.6 Variables and Macros

Variables or macros can be used interchangeably as part of expressions within directives or within Pascal code. For example, the following definitions do the same thing in terms of assigning a value to a symbol:

```
{$SET value = 123}
{$DEFINE macro = 123}
```

The difference between variables and macros is that variables are evaluated each time they are assigned with $SET, while macros are evaluated when they are referenced.

Let's consider the following statement:

```
{$SET //        value = 1+2}
Now the variable "value" exists, is numeric and has a value of 3.
```

Let us now consider the following statement:

```
{$DEFINE macro = 1+2}
//Now the macro "macro" exists, and its definition is "1 + 2".
//It is not a number or string.
```

A macro does not have a predefined type, it is rather a declaration of code, whose validity is not evaluated in its declaration.

You can think of a macro as a “code” type variable. This code is processed when the macro is referenced.

6.6.1 Variables and macros within directives

Variables and macros can be used within expressions in directives:

```
//Here the variable "value" already has a value and a predefined type
{$SET result = 1 + value}

//Here the macro "macro" is only defined and it is not known if its definition //corresponds to a string, number or
is code, or if it will generate an error.
{$SET result = 1 + macro}
```

To better clarify the evaluation of a macro, consider the following code:

```
{$DEFINE here = 1+2}

{$DEFINE something = 1+here}
```

Machine Translated by Google


```
{$MSGBOX something*5}
```

Only when, the $MSGBOX directive is executed, it will go to the definition of "something", to evaluate it, but in this evaluation, it will find that there is another macro, then it will go to the evaluation of "here", to evaluate it, and thus obtain the value of "something".

The result of this expression is 20. [13]

A similar case, using variables, would not require going back in the definitions:

```
{$SET here = 1+2}

{$SET something = 1+here}

{$MSGBOX something*5} //When calling "something", its value is already defined
```

6.6.2 Variables and macros within Pascal code

For when macros and variables are accessed from Pascal code. The case is similar:

```
var
   x: byte;
begin
   x := {$value};
end.
```

If value is defined as a variable, its value is read immediately. But if value is a macro, you must go to the definition of the macro (and probably other definitions), to determine its final content.

When a macro is evaluated, within the Pascal code, it can be thought of as a literal replacement of the content of the macro. For example, the following code:

```
{$DEFINE value=1+2}
var
   x: byte;
begin
   x := {$value}*5; end.
```

It will assign the value, 11 to the variable "x". Because by replacing the content of the macro, you will have: x := 1+2*5.

---

13 Note that "something" is evaluated independently before being used in the expression. That is, the content of the macro is not literally replaced. If so, we would have: 1+1+2*5, with the result 12, but what we really have is: (1 + (1+2))*5.

Note that this behavior is different from what you have when accessing a macro, from within the directives.

When using strings to replace code, it must be taken into account that the variables must be of type string. So if, in the following example, you want to use a variable instead of a macro:

```
{$DEFINE type = byte}
var
   my_var : {$type};
```

The code should look like this:

```
{$SET type = 'byte'}
var
   my_var : {$type};
```

6.6.3 Use of Variables or Macros

In general, it is recommended to use variables, when you only want store values, because they have less burden on the compiler.

It is recommended to use macros when their value depends on previous definitions. not known, at the time of the macro definition.

Macros and variables can be defined with the same name, but it is not recommended. In case of ambiguity in names, priority will be given to macros over variables.

It is recommended to use naming standards to differentiate macros from variables. So, for example, the character "_" could be used, prefixing it to the name of all macros.

6.7 Defining custom microcontrollers

PicPas contains predefined support for a limited number of microcontrollers. However, it is possible to define, through directives, the hardware of a new microcontroller, or an existing one.

To do this, system variables must be configured and RAM memory defined.

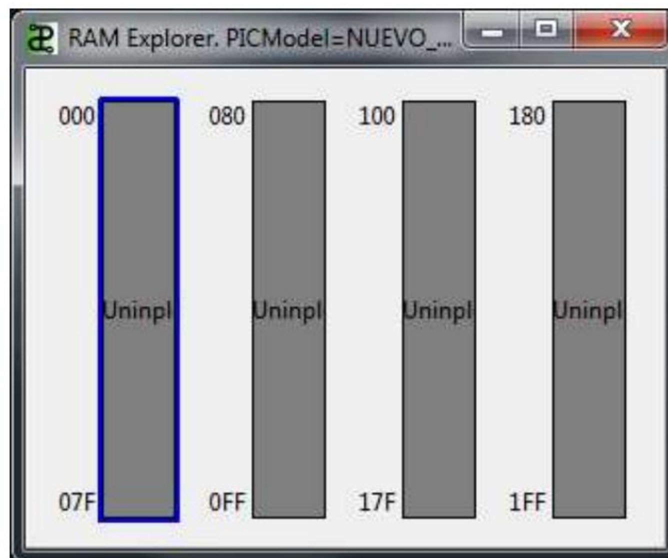
Suppose we want to define a model with 4 banks of RAM, and two FLASH pages. The code that defines these characteristics would be:

```
{$SET PIC_MODEL='NEW_PIC'}
{$SET PIC_NUMBANKS = 4}
{$SET PIC_NUMPAGES = 2}
{$CLEAR_STATE_RAM} //Clear RAM
```

Machine Translated by Google

With these simple directives, you already have the basic, but not complete, characteristics of a PIC. Note that the RAM state is cleared, with {$CLEAR_STATE_RAM} (See section 6.5.21), because by default, PicPas always defines an initial state for the RAM.

The RAM memory map of this new PIC would be like the one shown in the figure:



From here, you can define the RAM memory map.

The RAM memory map can be configured to measure, but it must always be in line with the internal architecture of the mid-range PIC microcontrollers. This architecture defines some common characteristics such as:

• The RAM banks have a size of 128 bytes. • Between 2 and 4 RAM banks can be defined. • FLASH pages have a size of 2048 14-bit words. • Between 1 and 4 pages of FLASH memory can be defined. • The special function registers (SFR) occupy the first positions in the RAM banks. • There are always some registers, such as PCL, STATUS or SFR that must be be mapped to all banks on the device.

The following code shows how a microprocessor similar to the PIC16F84 would be defined, by directives:

```
//Defines hardware characteristics {$SET
PIC_MODEL='MIPIC'}
{$SET PIC_MAXFREQ = 1000000}
{$SET PIC_NPINS = 18}
{$SET PIC_NUMBANKS = 2}
{$SET PIC_NUMPAGES = 1}
{$SET PIC_MAXFLASH = 1024}
//Start the memory, to start configuring it.
```

Machine Translated by Google


```
{$CLEAR_STATE_RAM}
//Defines RAM memory status
{$SET_STATE_RAM '000-00B:SFR, 00C-04F:GPR'}
{$SET_STATE_RAM '080-08B:SFR, 08C-0CF:GPR'}
//Defines memory mapped areas
{$SET_MAPPED_RAM '080-080:bnk0, 082-084:bnk0, 08A-08B:bnk0'}
{$SET_MAPPED_RAM '08C-0CF:bnk0'}
//Defines bits not implemented in RAM
{$SET_UNIMP_BITS '003:3F,083:3F,005:1F,085:1F,00A:1F,08A:1F'}
```

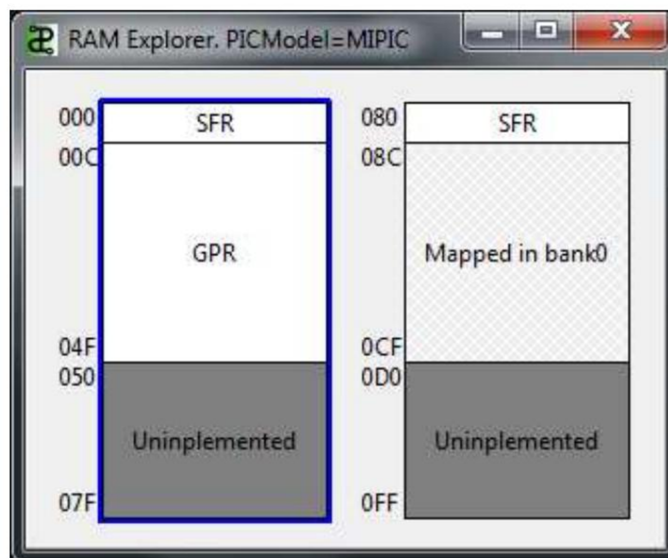
All of this code would be equivalent to what is achieved with the {$PROCESSOR PIC16F84} directive, except for the model name.

The instruction {$SET PIC_MAXFLASH = 1024} defines that there will only be 1024 free memory cells for the program, which are supposed to occupy the lower addresses of page 0, the only one that has been defined for this microcontroller.

The first line, which defines memory mapped areas, defines the registers: INDF, PCL, STATUS, FSR, PCLATH and INTCON, mapped to both RAM banks.

The hardware definition of the PIC does not refer to the name or function of the SFR registers, but simply to their status as an independent or mapped register.

This definition would generate a RAM map, as shown in the following figure:



Although the details of the mapped SFR records are not shown, they actually exist and are functional.

Mapping the GPR registers is necessary so that the compiler manages the RAM well, when assigning it to internal variables or registers of the compiler itself.

Mapping the SFR registers does not help compilation because the code generator uses its own memory map, but it is necessary for the tool

Machine Translated by Google

simulation (experimentally in the current version of PicPas) works correctly.

Defining the unimplemented bits, although not critical, also helps in device simulation.

Machine Translated by Google

7 APPENDIX

7.1 Supported Devices

PicPas supports, at the moment, only PIC devices from the Mid-Range family.

MID-RANGE FAMILY DEVICES;

PIC10F320 PIC10F322

PIC12F609 PIC12F615 PIC12F617 PIC12F629 PIC12F635 PIC12F675 PIC12F683 PIC12F752

PIC16F73 PIC16F74 PIC16F76 PIC16F77 PIC16F83 PIC16F84 PIC16F87 PIC16F88

PIC16F610 PIC16F616 PIC16F627 PIC16F627A PIC16F628 PIC16F628A PIC16F630 PIC16F631 PIC16F636 PIC16F639 PIC16F648A PIC16F676 PIC16F677 PIC16F684 PIC16F685 PIC16F687 PIC16F688 PIC16F689 PIC16F690

PIC16F707 PIC16F716 PIC16F720 PIC16F721 PIC16F722 PIC16F722A PIC16F723 PIC16F723A PIC16F724 PIC16F726 PIC16F727 PIC16F737 PIC16F747 PIC16F753 PIC16F767 PIC16F777 PIC16F785

PIC16F818 PIC16F819 PIC16F870 PIC16F871 PIC16F872 PIC16F873 PIC16F874 PIC16F874A PIC16F876 PIC16F877 PIC16F882 PIC16F883 PIC16F884 PIC16F886 PIC16F887

PIC16F913 PIC16F914 PIC16F916 PIC16F917 PIC16F946

Machine Translated by Google

table of Contents

1 Notes on PicPas... ....2 2
INTRODUCTION... ....3 2.1
Compiler Features.. ....3 2.2
Characteristics of the IDE...... ....3 2.3
Compiler limitations................... ....4 2.4
Installation................................ ....5 3
THE INTERFACE.................. 7
3.1 Code Explorer................... ....8 3.2
Compiler Selection................... ....9 3.3
Message panel........................... ....11 3.4
Editing Window.............................. ....13 3.4.1
Code folding................................ .... 14 3.4.2
Multiple Cursors.. ....15 3.4.3
Text Markers. 16 3.4.4
Synchronized Editing............. ....17 3.4.5
Search for declaration........................... .... 18 3.4.6
Automatic syntax checking... ....19 3.5
Compilation............................... 21
3.5.1 Partial Compilation................... 22
3.5.2 Automatic Compilation............................ 22
3.6 RAM Explorer... ....23 3.7
External Tools.. ....24 3.8 Theme
Configuration.................. ....26 4
DEBUGER/SIMULATOR...................... ....29 4.1
Starting the debugger/simulator................................ ....29 4.2
Memory maps......... ....30 4.3
Records Inspector.................. ....31 4.4
Assembler Panel..................... ....33 4.5
Execution control............................ ....34 4.6
Implementation information................................ 35
4.7 Real-time execution...... 36
5 LANGUAGE REFERENCE.................. ....38 5.1
Reserved Words.. 38
5.2 Comments... ....38 5.3
Numbers...................................... ....39 5.4
Identifiers............................. ....39 5.5
Structure of a program............................ ....40 5.6
Operators.. ....42 5.7
Operator precedence.. 45
5.8 Types of data.... 45
5.8.1 Char Type 46
5.8.2 Bit Type... 47
5.8.3 Word and DWord type................................ ....
.48 5.8.4 Type conversion... 48
5.8.5 Type properties. 50 5.9
Variables............... 53
5.9.1 Absolute Variables... 53

Machine Translated by Google

5.9.2 REGISTER variables.. 55
5.10 Constants........... 56
5.11 Types........ 57
5.11.1 Arrangements.. 58
5.12 Structures................................ ....60 5.12.1
Conditional IF … THEN........................ ....60 5.12. 2
REPEAT loop.. ....61 5.12.3
WHILE loop.. ....61 5.12.4 FOR
Loop.... ....61 5.13
System functions... ....62 5.14
Procedures................ ....65 5.15
Functions................... ....65 5.16
Interruption Procedure.................. ....67 5.17
Assembly code... ....69 5.17 .1
Labels and jumps.. ....70 5.17.2
Reference to variables.... ....71 5.17.3
Reference to constants..................... ....73 5.17.4
Reference to Procedures... ....74 5.17.5
Returning results for functions.................. ....75 5.17.6 The ORG
directive.......... ....76 5.17.7 Programming
only in Assembler.................. ....76 5.18
Units................... ....77 5.18.1 Use of
units.......... ....77 5.18.2 Creation of
units................... ....78 5.19
Pointers................................ ....80 5.19.1
Definition of pointers................................ ....80 5.19.2
Pointer Arithmetic... ....81 6
DIRECTIVES.............. ....82 6.1
Use of directives... ....82 6.2
Directive processing...... ....83 6.3 The language
of the directives............................ ....85 6.3. 1 Types of
data... ....85 6.3.2
Variables... .... .86 6.3.3
Operators... ....87 6.3.4
Operator precedence... ....87 6.3.5
Functions.............. ....88 6.4
System variables................. ....89 6.5 List
of directives.................... ....94 6.5.1
$PROCESSOR................... ....94 6.5.2
$FREQUENCY..................... ....95 6.5.3
$MODE......................... ....96 6.5.4
$MSGBOX................. ....97 6.5.5
$MSGERR...... ....98 6.5.6
$MSGWAR...... ....98 6.5.7
$INFO...... ....99 6.5.8
$ERROR... ....99 6.5.9
$WARNING................................ ....100 6.5.10
$CONFIG... 100
6.5.11 $INCLUDE.................................. ....101 6.5.12
$OUTPUTHEX.. 102

Machine Translated by Google

6.5.13 $DEFINE... 103
6.5.14 $IFDEF.. 105
6.5.15 $IFNDEF...................................... 106
6.5.16 $IF................................. 107
6.5.17 $IFNOT............................ 108
6.5.18 $SET............................ 108
6.5.19 $SET_STATE_RAM.................. 109
6.5.20 $SET_MAPPED_RAM................................. 110
6.5.21 $CLEAR_STATE_RAM.... 114
6.5.22 $RESET_PINS............................ 114
6.5.23 $SET_PIN_NAME................................ 115
6.5 .24 $MAP_RAM_TO_PIN.. 116
6.5.25 $SET_UNIMP_BITS.............. 118
6.5.26 $SET_UNIMP_BITS1............................ ....119 6.6
Variables and Macros 120
6.6.1 Variables and macros within directives. ....
120 6.6.2 Variables and macros within Pascal
code... ....121 6.6.3 Use of Variables or
Macros... ....122 6.7 Defining custom
microcontrollers................... ....123 7
APPENDIX.................. 126
7.1 Supported Devices. ....126 Table of content