Knowledge of baseline

Assembly - Two Flashing LEDs : Modular

Here's a version with modular source.
The hardware connections remain the same, but the source is varied.

What we want to achieve

We have just seen how the solution of creating libraries is an element of considerable effect on programming: stretches of code that perform certain functions can be collected and recalled when necessary without rewriting them.

The first way is simple: through the `#include` directive we can "include" in the compilation a stretch of code, pre-compiled, that is outside the source.

This method, while easy to understand, has its downsides. For example, when your library code requires RAM resources, you must manually insert them into the source.
In our example, the delay subroutine requires two counters, **`d1`** and **`d2`**, and a third counter **`d3`** for register W as an indicator of the number of times it should be repeated.
We need to deduce this information from the text of the **`Delay10msW.asm`** and insert these locations into the program. The method could be an obstacle in compiling complex programs where a lot of such resources are used, and it is still a contradiction when we are trying to automate and optimize the writing of the source as much as possible.

Isn't there a way to make this happen automatically?

The answer is yes: the **MPASM Macro Assembler** is not the only component that works on the compilation, but it is associated with a linker, **MPLINK,** whose purpose is to neatly bring together various stretches of code in a single compilation, automatically drawing on the resources available for the processor that has been specified (unfortunately the "automatically" does not exclude that the programmer has to tell the Assembler what he wants to achieve. A different definition of the various parts is enough).

In this different definition of some source elements lies the possibility of creating relocatable and modular programs, as well as the possibility of creating library files (***.lib***). Let's see the details.

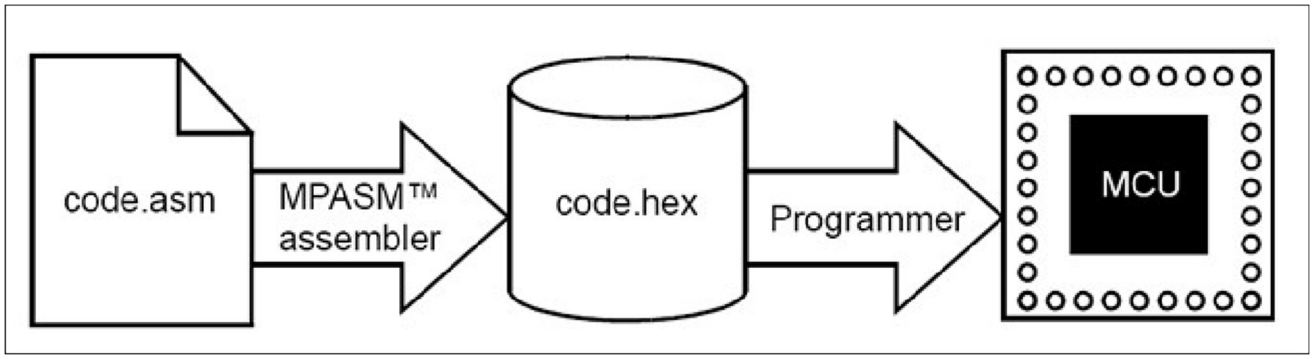
RILOCABLE MODULES

Using the `#include` Directive is a simple operation, but it can have some disadvantages:

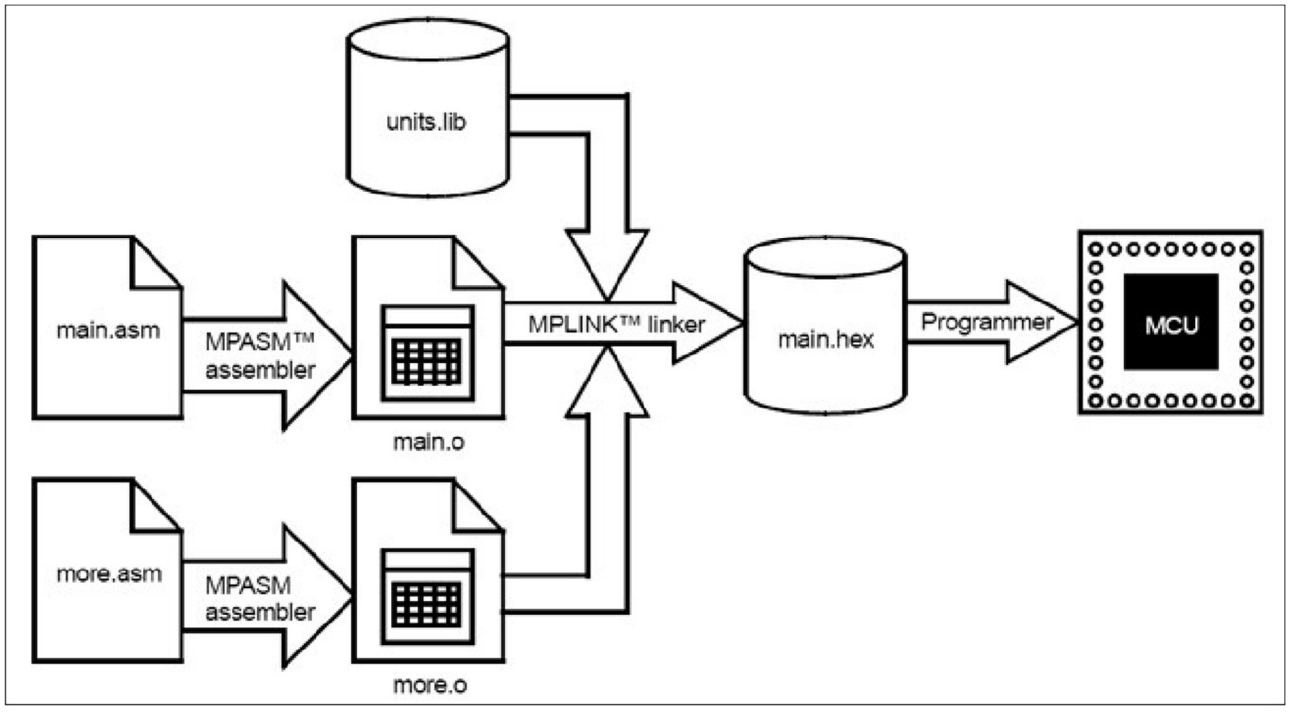
- If the included code needs RAM memory locations, they must be added to the list of memory definitions in the source, which must be done manually and can cause errors
- The labels used in the included file must not conflict with those present in the source, otherwise an error will be generated during compilation and the necessary lines will have to be manually corrected.

These problems can be avoided by compiling the part code to be included in a separate object; The source is also compiled as a separate object, after which the various parts are linked together to obtain the final executable code.

From a practical point of view, to further clarify, we can outline the action of compilation in the generic case of a non-relocatable code, which is defined as "absolute". This is the situation of the vast majority of examples proposed on the WEB:



The ***.asm*** source is directly compiled by the **MPASM MacroAssembler** and originates a file ***.hex*** file that will be written to the chip through a programming device. In the situation in which we intend to use the **Linker** to associate elements from different files in a single compilation, we can schematize the situation as follows:



The source, e.g. ***main.asm***, originates on the ***main.o*** object file and so does every other component of the compilation (in this example, the ***more.asm file*** produces a ***more.o***). In the example, the object files (together with another ***units.lib file***, which is a library file that has been associated with the project) are "assembled" by the Linker to form a single ***main.hex*** that can be written to the microcontroller's memory.

This operation is completely transparent to the programmer; All you have to do is associate all the necessary files to the project, written in the appropriate form.

To achieve this, each section of code must be independent of the others and have its own variables.

There is an essential difference between `the #include` of an "absolute" compilation and a relocatable compilation: in the first case, **the compiler places the instructions specified in the include in program memory at the point where it was inserted in the source text and which is determined by the value of the memory counter at that point**.
This, if on the one hand allows you to know exactly where the code is in memory, on the other hand makes it subject to the risks indicated in the previous exercise in relation to banks and pages. The same applies to the necessary RAM locations, which must be entered manually. This solution is the one commonly used for small programs (and this is the reason why it appears as practically the only way of writing in the examples normally available), where it is possible to keep an eye on the overall memory situation, but the same thing is difficult when the program begins to take on larger dimensions.

At this juncture, the "relocatable" structure offers the only way to minimize the problems arising from the assignment of memories, since they are solved by the Linker. On the other hand, since it is the task of the linker to position the various sections of code in the overall program, their positions are not a priori definable, precisely because it is up to the compiler to assign them to the appropriate locations. The relocatable structure, in addition to several other advantages, allows you to create object code (***.o***) libraries, which can contain, for example, subroutines of time, device management, communication, mathematics, etc., written directly or imported by third parties, with the possibility of being inserted into an Assembly, Basic or C source.

In practice, it does not make sense to rewrite or reinvent a section of the program every time, for example, to generate delays or for floating point mathematical operations when they are available, ready and working, from Microchip or others.

The problem, as mentioned, is that most of what is immediately available, including application notes, is written in absolute assembly. It is necessary to transform these sources into a relocatable form, so that they can be used when needed, without effort.

Transforming an absolute source into a relocatable one

Let's see how it is possible in practice to convert the "absolute" writing of the source into a "relocatable" writing and let's use as a starting point what we wrote in the previous exercise.

The operation is not complex, but it is subject to the formal rules of the Assembler, which must be adhered to.

For this we can print the files related to the **3Aw** project and, in particular, we can start to consider and make playable the tempo subroutine, **DelayW10ms.asm**, which, given its nature as an element external to the source and of general use, lends itself well to this conversion.

First we need to inform the Linker that some parts of the compilation are present in the source, others are external, in the various modules.
In our case, the external module is, in fact, of the delay subroutine. Let's rewrite it, under the name **Delay10ms_W** , in a relocatable form.

Reading the text, we almost immediately find a directive that has not yet been seen: **GLOBAL.**

GLOBAL

GLOBAL tells the compiler that the label is made common to all compiler actions for that project.
Its syntax is simple:

sp	GLOBAL	sp	Label	sp	; Optional Comment

The label is mandatory and implies the set of instructions that will be made "global".

Then we need to modify the subroutine by adding:

```
;#####################################################################
;                              SUBROUTINE
  GLOBAL       Delay10ms_W      ; 10ms x W delay routine
```

This makes the label **Delay10ms_W** , and what it implies, available for modular compilation: we have made the indicated label become a "global" resource during compilation.

The Directive **GLOBAL** works in tandem with another directive, **EXTERN** ,which must be inserted into the main code.

EXTERN

The Directive **EXTERN** tells the compiler that the label **Delay10ms_W** , where it is referred to in any other text, is NOT there, but comes "from the outside". Its syntax is similar to the previous one:

sp	EXTERN	sp	Label	sp	; Optional Comment

The label in question will be searched in the set of those defined by **GLOBAL** and used for the

compilation.

To sum up:

- **EXTERN** and **GLOBAL** work in pairs
- **GLOBAL** is used in a "form" file to make it available for compilation

- **EXTERN** should be used when you want to call external modules.

So, in the source, we will have:

```
    EXTERN      DelayW10ms      ; 10ms x W delay routine
```

It is now necessary to adjust the definition of the RAM memory that the module uses.

RAM in a relocatable way

As we have mentioned, RAM memory is generally divided into banks, to access which it is necessary to act on appropriate "bank switches".

A first problem concerns the fact that different chips have different resources, both as data memory and as special registers (SFR) and they can be divided into one or more banks (up to 32!), starting at different addresses. To use absolute values, you need a very detailed knowledge of the memory map of the component you want to use. Not to mention that the differences between components make it difficult to port codes that use absolute references. For example, the data RAM of 10F200 starts at address 10h and is on only one bank; The data RAM of 12F519 starts at 07h and is divided into two banks.

We must also take into account the presence of data memory shared between banks (shared), i.e. accessible identically from any bank, and data memory accessible only if you are on a specific bank.
Making RAM definitions relocatable means passing the compiler the task of assigning data RAM requests to the right addresses.

A few pages of Theory

It is necessary to introduce some comments on the main problems facing the ICPs and, more markedly, the Baselines. These are:

- **Dividing RAM into BANKS**
- **Dividing Program Memory into PAGES**

These two elements are highlighted at most in the Baselines, which have opcodes that are developed on only 12 bits. In the RISC structure, the opcode contains both the code of the actual statement and the subject of the instruction.
And the objects of the instructions are essentially three:

- Numeric values (literal) : opcode **`movlw`**, **`andlw`**, etc.
- Addresses in RAM (SFR data or registers): opcode **`bsf`**, **`btsfc`**, **`movf`**, etc.
- Program memory addresses: opcode **`call`**, **`goto`**

In 12bit, while you can enter 8-bit numbers, you can't have space for the entire address, whether it's RAM or program memory. That is, only a part of these areas can be addressed directly with the opcode, and to be exact 512 locations of program memory and 32 of RAM.
If the chip has more memory, it is divided into BANKS (for RAM) and PAGES (for Flash) and a mechanism outside the opcode is required to address these areas. In particular, the **`Baseline call`** statement has an additional address limit of only 256 locations.

These elements make the Assembly programming of small PICs "unpleasant" (starting from the Enhanced Midrange and PIC18F, systems have been implemented to lighten the situation) and it is necessary to pay attention to these factors, on which the operation of the program may depend when you exceed Page 0 of the program memory or you need to work with RAM registers on banks other than Bank0.

A Note on 16F84

Even if in the simplest exercises of this course the problem of pages does not arise, since you never exceed page 0, it is advisable to be aware of it.

In this sense, a basic criticism that can be made of the excessive use of the famous 16F84 (which, by the way, is a Midrange and not a Baseline) in tutorials is precisely the fact that this PIC has a single page memory and large enough to contain even discrete programs. This means that those who have only tried their hand at this component have no idea of the pagination problem. So, by using other PICs, you can end up in serious difficulty in getting your program to work that has gone beyond page 0 of the program memory.

For this reason, we have opened a window on the subject, so that we can begin to learn about it and, although we will not implement here all the possible mechanisms of modular programming, at least we intend to make users aware of the subtle traps that lurk in the Assembly programming of small PICs.

NO "ERRORLEVEL –302" PLEASE !

Still on this topic, in the introductory lines to too many examples of programming available on the WEB, even supposed "tutorials" or in the school/educational field, we note the presence of lines like this:

```
errorlevel -302
errorlevel -306
```

This is one of the biggest nonsense you can find in this area.
Because the beginner copies the example, considering it a sacred text, but, not being informed of the specific functions of the various lines, he then finds himself in serious difficulty when he wants to experiment on his own.

ERROLEVEL

is an Assembler directive that is intended to enable/disable the printing of compiler-generated compiler error messages in the ***.lst*** file and ***.err*** file.
Its syntax is simple:

sp	**errorlevel**	sp	**object[, object,...]**	sp	**[; comment]**

The object is composed like this:

{0|1|2|+msgnum|-msgnum} [, ...]

with the following functions:

Value	Function
0	Messages, Warnings and Errors
1	Warnings and Errors
2	Errors
- msgnum	Prevents printing
+ msgnum	Enable Printing

Compiler errors cannot be disabled, but Warnings can be disabled using setting 2, and Messages can be disabled using settings 1 or 2. In addition, messages can be deactivated individually, with their identification number.

For example, **`errorlevel 0`** inhibits the printing of messages, Warnings, and Errors.

Fortunately, options 0, 1, 2 don't seem to be very well known; But, unfortunately, so are those related to specific messages. And, by the way, only in disabling mode.
Thus, the **`much-seen errorlevel -302`** is rampant, which inhibits the printing of messages related to operations in which attention is required at the desk of the register in question.

This means that the programmer's attention is not directed to these critical points and he is therefore not able, except with great difficulty, to trace the cause of possible errors in lines where the question of desks has not been taken into account.

The use of the directive is RESERVED EXCLUSIVELY for those who know what its function is; And it is precisely in this case that the experienced user will be careful not to delete an indispensable

error indication.

Only in the final version, perfectly debugged, in the unlikely event that you want ***to eliminate the presence of the warnings related to the banks from the printouts of the*** .lst ***and*** .err files, you can introduce the directive.

In other cases, `errorlevel -xxx` is a nonsensical, unmotivated, and potentially harmful addition.

Baselines and Subroutines

This page does not affect the exercises, so you can postpone reading it to a later time, if it introduces too many "strange" concepts for you, but it becomes essential to understand the reasons for some subsequent choices.

However, we believe that it is appropriate to mention, at least in principle, this topic as a clarification for some choices and as a preliminary indication of what we will be able to see later. In the practice of programming, in fact, it is advisable to become aware of the characteristics and limits of what is being used and act accordingly even if there is no real need at that time; This makes it possible to avoid finding oneself with apparently insurmountable difficulties when one comes up against those mechanisms and limitations without being informed and aware of them.

So let's make a note about the subroutines in the Baselines, which you can read or skip at will, but which could be indispensable for debugging more complex programs than the one being written here.

We have seen from flow chart that the subroutine is a block of instructions outside the main program, but written in the same source. You might be wondering where it should be written in the source. For the PIC18F, it doesn't matter where to insert subroutines, since they don't have paged program memory. For Midranges, pagination must be considered. For Baselines, additional limitations must be added:

1. the Baseline stack is minimal, only two levels, i.e. you can call a subroutine (whose call copies the return address to the stack), which can in turn call another subroutine, which in turn copies the return address, thus occupying the two available layers of the stack. However, at this point, the stack is full and no further call is possible unless you have re-entered from a previous one. In fact, if we were to add an additional **`call`**, the stack, which works as a **LIFO** (Last In First Out), whereby the data is extracted in the reverse order of the one in which it was inserted, would eliminate the first address entered, "throwing it away", to make room for the last one and thus making it impossible to return from the first subroutine (stack overflow).

2. For the Baseline structure, a call to a subroutine can only be made if the subroutine address is within the first 256 addresses of the program memory page.

For the first case, it is necessary to simply take care of the nesting of the subroutines so as never to exceed the two levels of the stack (it should be kept in mind that Baselines do not have protection systems for overflow or underflow of the stack and an error in its management causes the program to crash, which are instead present in PIC18).

For the second case, in Baselines the pages are 512 words wide, it is necessary to manage the change when the program memory exceeds this limit. In addition, care must be taken to locate the subroutines within the first 256 addresses.

This is one of the significant limitations for these small ICPs, which, on the other hand, are deliberately very simplified and therefore lack the mechanisms and performance of the higher families.

In relation to this problem, you can find an explanatory page here.

Let's take a practical example with our code: let's put the subroutine in the first 256 elements of the program.

```
MAIN:
; Reset Initializations
     CLRF    GPIO          ; GPIO preset latch to

; TRISGPIO      --0----- GP5 out
     movlw   b'11011111' ; mask direction PORT
     tris    GPIO          ; To the Management

     Goto    Mainloop

;=====================================================================
;=                         SUBROUTINES                               =
=====================================================================
; Delay = 1 second @ Clock = 4 MHz
; FOSC/4 = 1 us , 1 second = 1000000 cycles
Delay1s:              ; 999990 cycles
  movlw   0x07        ; Initialize Counters
  movwf   d1
  movlw   0x2F
  movwf   d2
  movlw   0x03
  movwf   D3
Delay1s_0:
  decfsz  d1, f
   goto   $+2
  decfsz  d2, f
   goto   $+2
  decfsz  d3, f
   Goto   Delay1s_0

; End of Count 999990 Cycles - Now Sum i    10 missing
  Goto    $+1           ; 6 Cycles
  Goto    $+1
  Goto    $+1
  retlw   0             ; 4 Cycles including
=========================call=======================================
```

```
Mainloop:              ; <<--<<<--
; lights up            |          |
  LED_ON LEDs          ;          |
                       ;          |
; Standby 1s                      |
  Call     Delay1s     ;          |
                       ;          |
; turns off LEDs                 Loop
  LED_OFF              ;          |
                       ;          |
; Standby 1s                      |
  Call     Delay1s     ;          |
; Loop                 ;          |
  Goto     Mainloop    ; >>-->>---
```

Note the need to insert a jump (**goto mainloop**) that allows you to bypass the instructions of the subroutine. If this line is missing, the program counter, after executing **three GPIOs**, will continue executing the subroutine instructions, up to **retlw** where it will be replaced by a random value taken from the stack, which was not loaded by the **call**, crashing the program.

This is a simple solution if the number of subroutines used is large enough to stay in the space of 256 words. This space, however, is not that big and if we use various subroutines it can be easily filled, also because it is the area required to place tables (lookup tables).

A second option is to direct to the subroutines with an indirect system.

```
MAIN:
; Reset Initializations
     CLRF     GPIO          ; GPIO preset latch to

; TRISGPIO      --0----- GP5 out
     movlw    b'11011111'   ; mask direction PORT
     tris     GPIO          ; To the Management

     Goto     Mainloop

; Table Jumps to Subroutines
rDelay1s Goto      Delay1s

================================================================================
Mainloop:              ; <<--<<<--
; lights up            ;          |
  LED_ON LEDs          ;          |
                       ;          |
; Standby 1s                      |
  Call     rDelay1s    ;          |
                       ;          |
; turns off LEDs                 Loop
  LED_OFF              ;          |
                       ;          |
; Standby 1s                      |
```

```
  Call     rDelay1s    ;          |
                       ;          |
; Loop                ;          |
  Goto     Mainloop    ; >>-->>---

;=====================================================================
;=                        SUBROUTINES                                =
=====================================================================
; Delay = 1 second @ Clock = 4 MHz
; Fosc/4 = 1 us , 1 second = 1000000 cycles
Delay1s:              ; 999990 cycles
```

The mechanism is simple:

- `Call rDelay1s` sends not to the subroutine, but to the label that corresponds to a `Goto Delay1s`
- The `call` has saved the return address on the stack; this address corresponds to the instruction following the `call` itself (in the first step `LED_OFF` and in the second `Goto mainloop`).
- The `goto` does not involve any change in the contents of the stack, but only changes the PC by directing it to the label in question.
- once the instructions to **`rDelay1s are exhausted`**, the `final retlw` retrieves the return address from the stack after the call

Then the step, such as in the first call, :

```
          call rDelay1s ; the stack saves the return address
rDelay1s goto Delay1s ; the PC is diverted to Delay1s
Delay1s ....              ; The subroutine instructions are executed
          retlw0          ; The return address is taken from the
Stack
LED_OFF                   ; return to main and LED power off
```

Since goto does not have the limitation of 256 addresses, the subroutine can be found even beyond this limit.
This solution makes it possible not to crowd the first 256 addresses. The `goto mainloop` is always needed to move beyond indirect calls to subroutines.

Program Memory Pages

Even this page does not affect the exercises, so you can postpone reading it to a later time, if it introduces too many "strange" concepts for you, but it becomes essential to understand the reasons for some subsequent choices.

It must be considered that in 8-bit PICs, with the exception of PIC18, there is the problem of program memory pagination. In addition to this, as we will see later, there is the

RAM memory problem in banks: **banks and paging** are the main tribulation of Assembly programming on PICs.

In short, program memory, when it extends beyond a certain size, is divided into pages, due to the limited number of bits on which an instruction is encoded (12 for Baselines) and the structure of the Program Counter.
Likewise, if the RAM area exceeds a certain size, it must be considered divided into banks. The

problem is highlighted in this way:

- As far as pagination is concerned, executing instructions in a certain area of program memory (i.e. on a certain page) may not be possible to switch to another page except through a switching mechanism that requires the use of additional bits to be handled appropriately in the program
- As far as banks are concerned, accessing the contents of a certain bank also requires action on particular bits that must be treated by the program in the appropriate way

Where does this issue matter?
Essentially when the program becomes more complex than what we are seeing now and, especially, when using modular programming techniques: in these cases it can be difficult to determine where the compiler will place a certain stretch of code in program memory.
For example, even in the case of our small PIC12F519, the program memory extends over two pages ; if the program becomes large, it is quite possible that part of the instructions will no longer find space on page 0 and will have to be written on page 1. To access it, you will need to switch the page using the appropriate switch.

Of course, it is possible to check the address situation from the compilation, and MPASM also provides warning messages in relation to pages and banks. However, it is always a tedious revision of what has been written and therefore a commitment of time and with the possibility of finding the object filled in with addressing errors.

One technique to overcome this situation is to apply page switching anyway.
MPASM provides a specific command for this operation: **`pagesel`**.
Its function is to position the page switch at the right value according to the indicated label:

- **`Pagesel Label`** causes the page switch to be switched for the page it contains **`Label`**

Obviously, the operation can also be carried out by an instruction or more opcodes, but **`pagesel`** is much more functional because:

- The page switch is one bit if the memory has two pages, but it becomes two if the memory has 4 pages, and so on. **`Pageel`** switches the number of bits needed according to the microcontroller declared at the beginning of the source without the programmer having to check the number of bits.

- **`Pageel`** switches the bank according to the indicated label and then autonomously evaluates which page it is on and activates the page switches appropriately, without the

programmer should take the trouble to locate the label page. An extra automatism to ensure that the source treats only symbols and not absolutes.

In our case we can exemplify the application of the command:

```
MAIN:
; Reset Initializations
     CLRF    GPIO          ; GPIO preset latch to

; TRISGPIO     --011111   GP5 out
     movlw   b'11011111'
     tris    GPIO          ; To the Management

     Pagesel Mainloop
     Goto    Mainloop

; table jumps to Delay1s
subroutines
     Pagesel Delay1s
     Goto    Delay1s
```

This way we are sure that wherever **`mainloop`** and **`rDelay1s`** are allocated, the jumps will be executed correctly.

It should be kept in mind that the switches of the pages (and banks) are set to 0 on reset, i.e. they allow access to Page 0 (and Bank0), but, once modified by the program, they remain at the imposed value until they are modified again or a reset occurs.

This means that, in a multi-page program, the problem of maintaining the right page for each element of the program is not secondary and requires an intensive use of page turning or a structure in which the development of algorithms is as concentrated as possible in order to minimize the need for page turning. This leads either to an increase in the number of instructions to be executed (since it is compiled as the instruction or set of instructions needed to change pages), which slows down the execution and increases the use of program memory, or an exaggerated attention to the problem, which distracts the programmer from the rest of the work.

There are not many absolutely valid strategies to minimize the problem, although, as mentioned, it is possible to decide to concentrate, for example, all the subroutines on page 1 so as to know for sure where they are placed and thus have the opportunity to minimize the use of page switches.

For now, let us take note of these considerations; We will see in the course of the exercises something more on the subject where it is necessary. In particular, in the next exercise we will deal with the same topic as the present one, but in the light of a modular writing of the program.

Still on the benches

It is appropriate to say a few words about the problem of benches.

For design reasons, PICs have RAM memory that can be divided into several banks (from 1 to 32). And in the RAM area we find:

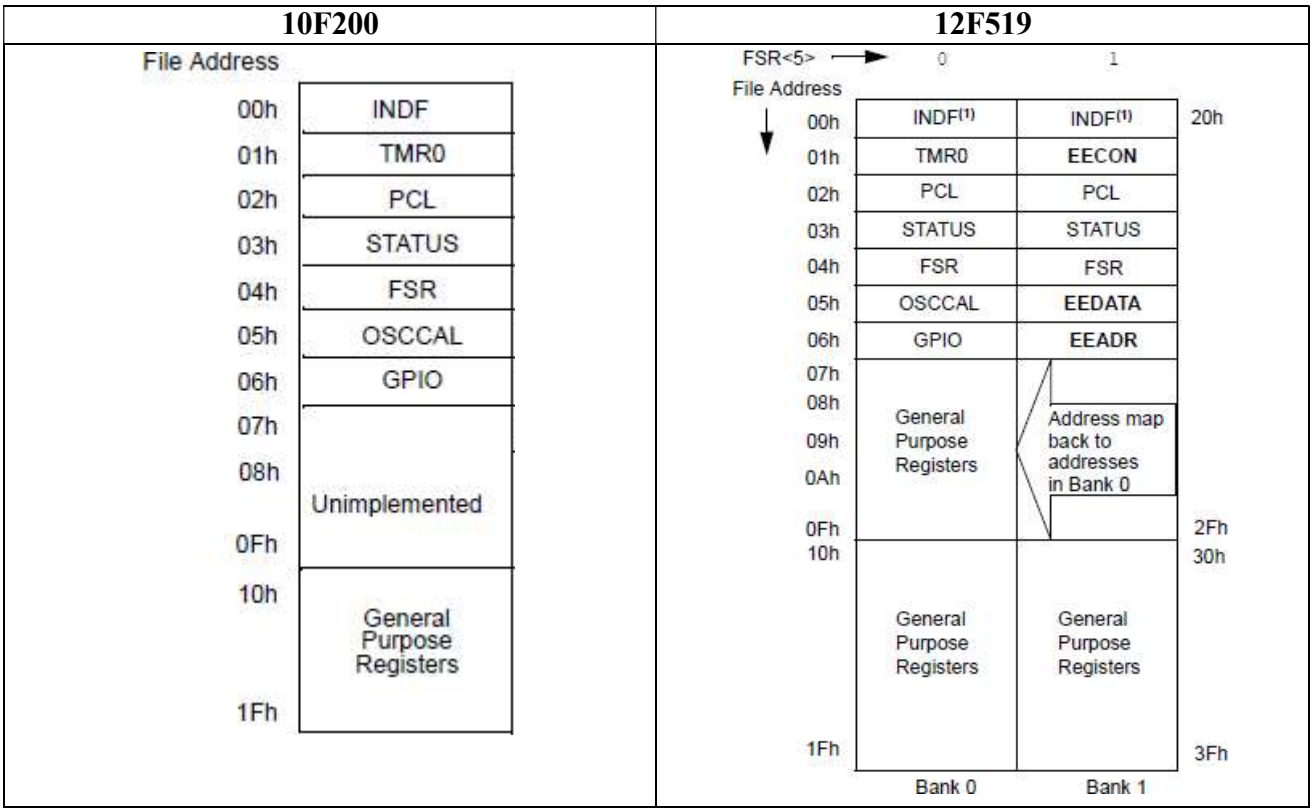
- **SFRs**, i.e. the control logs of peripherals and the system in general
- **Data RAM**, i.e. RAM memory cells dedicated to supporting the program's activity during its operation

This RAM area is in turn divided into:

- Shared RAM between banks, i.e. memory cells that can be read/written from any bank (***shared RAM*** or ***common RAM***)
- RAM only accessible if you are in a bank.

In addition, there may be areas of RAM whose addresses correspond to cells that are not implemented, where the write is lost and the read yields 0, i.e. they are not usable (***Unimplemented***).

Let's take a look at the two processors mentioned above: 10F200 and 12F519:



In 10F200, minimal PIC, a single bank contains all the RAM area. If we look at the map presented, we notice that there is an ***Unimplemented zone***.

In 12F519, you need two banks. If we look at the latter, we notice that the area between 07h and 0Fh is identical to that between 17h and 2Fh, i.e. it is a common RAM zone, while the zones 10h-1Fh and 30h-3Fh are part of **Bank0** and **Bank1 respectively**.

	Bench 0	Bench 1
SFR	from 00h to 06h	from 20h to 26h
Shared RAM	07h to 0Fh	from 27h to 2Fh
Local RAM	from 10h to 1Fh	from 30h to 3Fh

Incidentally, Shared RAM was introduced by Microchip's designers with the aim of making available a number of registers whose access is independent of the bank in which it is currently working. This is extremely useful for speeding up in-memory operations. In particular, in families of PIC equipped with interrupts (Midrange), the shared area is used for the management of the memory used in saving the context during the interruption. It should be noted that the PIC18F and Enhanced Midrange also have the bench situation, although there are mechanisms that simplify the use.

Let's specify even better what this entails, since the problem of banks (and pages...) is one of the critical points of the ICPs and the mechanism must be understood well: in the case of 10F200, since there are no banks, there is no problem, but in the case of 12F519 the thing takes on a completely different aspect.
At the default of the POR (after the reset due to the arrival of the supply voltage), the content of the FSR register is 0. This means that its bit 5 (FSR<5>), which is the bank switch, is located at 0. This makes the contents of Bank0 accessible to the program's instructions.
Thus, by addressing 01h we will be able to access TMR0; directing 05h to OSCCAL and 06h to GPIO; as well as the RAM from 10h to 1Fh and the registers INDF, PCL, STATUS, FSR and the data RAM from 07h to 0Fh.

If we bring bit 5 of FSR (`bsf FSR,5) to 1`, Bank1 is selected, i.e. bit 5 will be added to the RAM address of the instruction, adding 20h to the addresses indicated by the instructions. Thus, by addressing 01h we will not access the content of TMR0, but that of EECON; by addressing 05h, we will end up at 25h, i.e. not at OSCCAL, but at EEDATA and 06h will become 26h, i.e. not GPIO, but EEADR. The data RAM will be between 30h and 3Fh.
On the other hand, RAM from 10h to 1Fh and the INDF, PCL, STATUS, FSR registers and data RAM from 07h to 0Fh will always be accessible, since the corresponding ones in Bank1 are identical.
This means that, after the ROP:

```
movf    01h, W      ; copy TMR0 to W
movwf   10h         ; copy W to RAM
```

But if I want to access the EEPROM registers, I won't be able to do it with:

```
movwf   21h         ; copy W to EECON
```

With this line, the contents of W will end up in TMR0 ! This is because the amplitude of the Baseline

instructions (12bits) does not allow you to directly address beyond 1Fh. To exceed this value and

To access the block between 20h and 3Fh I have to use the artifice of adding the necessary bit through FSR,5:

```
Bsf     FSR,5       ; Step in Bank1
movwf   21h         ; copy W to EECON
```

However, be careful: once the dealer switch is set to 1, it remains in its value until it is changed again through the program instructions. So, once you have selected, for example, Bank1, you will access the RAM area in this bank:

Then, writing:

```
Bsf     FSR,5       ; Step in Bank1
movf    21h, W      ; I copy EECON in
                    W
movwf   10h         ; copy W to RAM
```

you get that the content of W ends up not in 10h, but in 30h ! To get the destination to be 10h, you need to:

```
Bsf     FSR,5       ; Step in Bank1
movf    21h, W      ; I copy EECON in
                    W
Bcf     0FSR,5      ; step in Bank0
movwf   10h         ; copy W to RAM
```

On the contrary, it is correct:

```
 BSF     FSR,5       ; Step in Bank1
MovF     STATUS,w ; copy Status in W
MovWF    0Fh      ; copy W to ram
```

It gives the desired result, since both STATUS and the 0Fh location are accessible from both counters.

It is clear that the problem of a correct management of the banks is essential for the correct functioning of the program: if, for example, using 12F519 I want to act on the EEPROM and I neglect to switch the bank from 0 to 1, I find myself writing and reading completely different registers, going to "mess" **TMR0**, **GPIO** and **OSCCAL,** with the imaginable consequences.
The issue of benches is present in all PCIs, but in the PIC18F it has a mode of access that, at least for SFRs, allows the problem to be forgotten; and the Enhanced Midrange also have some facilities. Therefore, the greatest weight is obtained using Baseline and Midrange, where any Assembly programmer is conditioned by the need to ensure the right bench for the operation that the program has to perform.

Also, as we mentioned in another tutorial, the bank switch can be even larger than just one bit, since there are chips with more than two banks. And there are other positions of the bank switches: in the Midrange it is no longer an element of FSR, but the RP1:0 bits of the STATUS. It follows that,

unless some automatism is used, it is necessary for the

Assembly programmer has a very detailed knowledge of the chip you are using. Fortunately, as we have already seen before, the use of the banksel pseudo-opcode offered by MPASM makes things much easier, since we give this macro the task of determining what needs to be modified and where to access a certain register, among other things called through labels and not absolutes, depending on the microcontroller that we have specified in the compilation project.

Obviously, if we are dealing with processors that have only one memory bank, such as 10F200 or 16F84, or with programs that use minimal resources, such as to be contained in a single page, the problem of banks in the use of data RAM does not arise.
But if the memory demands expand, as happens in non-minimal programs, it becomes essential to also use the RAM present on banks other than the first one (Bank0) and it is necessary to know the mechanism. It is clear that the availability of a system that automates the allocation and use of data RAM is desirable.

The use of banksel relieves us of the need for continuous consultation of the memory map, as it will be enough to write:

```
      Banksel    logname
```

to be taken directly to the desk where that given register resides. It should be noted that high-level languages, such as BASIC and C, automatically take care of the management of desks (and pages); so those who use these generally don't need to worry, unless assembly lines are introduced.

Going back to our program, if we look at the source of the **DelayW10ms**, we find that the definition of the RAM memory has taken on a different appearance from what we have seen so far:

```
; ******************************************************************
;                              RAM
; general purpose RAM
      UDATA
D1    RES 1      ; counters
D2    RES 1
D3    RES 1
```

Let's try to clarify how it works.

UDATA

The **UDATA** (***User DATA) directive*** indicates that in the general purpose RAM area of the processor it is necessary to reserve (**res**) the locations indicated in the indicated quantity, associated with the relative labels. In our case, for **12F519** the UDATA **RAM** consists of:

RAM	Bench 0	Bench 1
UDATA_SHR	07h-0Fh	
UDATA	10h-1Fh	30h-3Fh

Let's see that it is composed of the two segments described above:

- `UDATA_SHR` is the shared area, i.e. shared and accessible from all desks indifferently.
- `UDATA` which is the area of data Ram relative to the various banks.

This information is provided to the Assembler, for each chip, by the ***processorname.inc*** and ***processorname.lkr***, which we always find in the folder that contains the MPASM suite.
The directives accept a label, in the usual form:

[Label]	sp	UDATA	sp	[; comment]

However, a label for `UDATA_SHR` indicates the name of the linker section in which the variables will be assigned. Most PICs only have one shared memory zone, and it's not very useful to set it explicitly by attaching it to a label.

Operating in this way, it is clear that the knowledge of the data memory map of a given processor is not particularly important, as it will be the task of the Assembler to retrieve the right addresses, taking them from the reference files.

It should be noted that, unlike `CBLOCK,` labels need to be self-defined starting in the first column. For each label, the `res command` reserves the number of RAM locations specified.

CBLOCK vs. UDATA

When we use `CBLOCK,` this directive causes the symbols to be associated with RAM addresses starting from the source address indicated in the subject of the directive.

However, the Assembler has limited ability to verify if it is really usable RAM for data and there are no checks to ensure that the memory really exists at these addresses; if we were to address an area of unimplemented RAM, writing/reading it would be useless. It also does not avoid the mistake of using the same memory cell by defining it in two different places with different labels, which can create serious problems at execution, among other things difficult to debug.

The use of `UDATA` , on the other hand, allocates bytes in the available RAM by the clinker who will choose the exact address, as well as ensuring that there are no more variables defined with `overlapping res` . This makes the code more portable, and that available memory will be used on any chip. So it doesn't matter where the RAM starts for that processor, because this information is defined in the compiler's support files.

In fact, if with **CBLOCK** we have a precise correspondence between label and address, when using the linker it is this that determines the relationship between location and label. That means a certain label could go anywhere within the section."***udata_shr***" of the linker, and that the memory regions in that section are defined with **SHAREBANK.** Normally, you don't have more than one shared memory section, so it's common to use the name ". ***udata_shr*** " and do not use an explicit name in the source code.

The trend will be to use **UDATA** for general variables, except when you explicitly want the variables to be in the shared memory region.

Subroutines made relocatable

The subroutine, modified, now no longer makes use of any absolute reference: it has become "relocatable".

```
;*****************************************************************
; Delay10ms_W.asm
;-----------------------------------------------------------------
;
; Title          :    Modular Subroutine.
;                     Performs a 10ms waste time cycle for the number
;                     times indicated in W. Fosc=4MHz
; Processor      :    PIC Baseline
; Date           :     15-04-2013
; Modified on :
; Version        :    V0.0
; Hardware ref. :
; Author         :    Afg
;
;-----------------------------------------------------------------
;*****************************************************************

    #include <p12F519.inc>      ; Any Baseline

     GLOBAL Delay10ms_W

;=================================================================
;               DEFINING VARIABLES
      UDATA
D1    RES 1      ; counters
D2    RES 1
D3    RES 1

; ################################################################
;                       SUBROUTINE
      TAILS

; Delay = 100 ms - Clock frequency = 4 MHz - 0.1s = 100000
cycles Dly10ms_W:     ; 99993 cycles
  movlw     0x1E       ; Initialize Counters
```

```
  movwf     D1
  movlw     0x4F
  movwf     D2
Dly10ms_W0:             ; Counting Loops
  decfsz    d1, f       ; d1=d1-1
   Goto     $+2         ;
  decfsz    d2, f       ; d2=d2-1
   Goto     $+2         ;
  decfsz    d3, f       ; d3=d3-1
   Goto     Dly10ms_W0
; End of Count 99993 Cycles - Now Sum the 7 Missing
  Goto      $+1         ; 3
  NOP
  Retlw     0           ; 4 cycles including call

     END
```

In addition to the aforementioned **GLOBAL** and **UDATA**, two other changes have been introduced:

- **CODE** , which tells the compiler when a stretch of code begins.
- The code must be terminated with an **END.** Notice that this closes the local code, but not the combined compilation of a set of code blocks.

CODE vs. ORG

CODE is analogous to **ORG,** but unlike ORG, it does not require a mandatory object (starting address); the directive tells the compiler that a stretch of code starts at that point. If an address is not expressed, it means that it will be up to the linker to place it where it belongs.
It is also possible to assign a start address to CODE, where it is necessary to:

```
RESET_VECTOR     TAILS 0x00
```

This line replaces the one we have already seen:

```
RESET_VECTOR     ORG 0x00
```

within a relocatable modular structure. But in our case, the directive has no object as it is "relocateable", i.e. it can be placed in any appropriate area of program memory, by the Linker. Therefore, no absolute value needs to be defined:

```
MAIN     TAILS
```

The MAIN

Of course, we must also amend the text of the main programme to adapt it to the situation of modularity and relocation. The first step is to insert the directive that

allows the linker to retrieve the modular subroutine we have written, by inserting the **EXTERN:**

```
        LIST       p=12F519
        #include <p12F519.inc>

     EXTERN    Delay10ms_W         ; 10ms delay modulated by W
```

Note that line `#include <p12F519.inc>` defining the type of processor used is still required. In the context of the creation of a relocatable object, this line is essential, but it is not inadequate, since it is, as we have seen, the list of labels equivalent to the absolute values of the individual registers of the processor and that the compiler will use during its work.

The already seen will also have to be found in the main file, to replace the ORG directive: how does the modified source appear:

```
; ##################################################################
;                                  RESET ENTRY
; Reset Vector
ResVec    TAILS
0x00
```

Notice how here, unlike the subroutine, the directive has as its object the program memory address where the next code will be placed, i.e. the initial Reset vector. From here, the compiler will begin to place the following instructions.

Pages & PAGESEL

We have mentioned the problem of memory pagination in PCIs, a fact that is of considerable importance in Baselines.

The problem is quite similar to that seen for banks: 12-bit opcodes are not sufficient to contain addresses to exceed 512 locations of program memory and, as for banks, it is necessary to complete the address with bits taken from another register. Typically this is the **STATUS,** PA0/PA1 **bits**, but other families of PICs will have different control bits. This makes it necessary for the programmer to know this feature in detail for all the chips he intends to use.

However, the Assembler also offers the solution here, through the pseudo **pagesel** instruction of MPASM, which allows us to be independent of the characteristics of the chip used: the position in which the page switching bits are located and how they are moved is the task of the compiler, which draws the necessary information from its databases and replaces the directive with the right command instructions of the page switches. The syntax is the usual:

[label]	sp	Pagesel	sp	object	sp	[; comment]

The initial label is and the command should not start in the first column.

The object is the label whose page you want to reach. Valid items such as:

```
      Pagesel Init    ; select Init page Pagesel $
                      ; Select the current page
      Pagesel 1       ; Select Page 1
```

This action can also be performed by instructions of the processor set, but this requires a greater knowledge of the mechanisms of the Program Counter, as well as knowing on which page the label in question is located; with the **pageel** we can do without this knowledge and trust in the action of the compiler that will select the right addresses based on the value stored for those labels.

In fact, it is the linker that generates the appropriate code of the page to be selected. For the 12-bit core (Baseline), the set of instructions needed to change the bits of the STATUS is generated.
If the chip contains only one page of program memory, no code is generated. We could also use a

macro that automates it even more. For example,:

```
; call for a subroutine that is on a different page than the current
one
fcall MACRO address
      Pagesel address      ; Select the subroutine page
      Call    address      ; Call
      pagesel $            ; Reposition the current page
      ENDM
```

The macro is created with an object, here defined by the label **address**, which, in this case, will correspond to the subroutine you want to access. So in its use it will be, for example:

```
      fcall   address
```

which will generate the compilation of:

```
      pagesel address
      call    address
      pagesel $
```

There is also an additional possibility, which is to use the **LCALL** macro command that MPASM makes available for 10/12/16 PICs.
This performs the function of the first two lines of the macro and must be completed with the **$ pageel**
final.

```
      LCALL   address
      pageel $
```

LCALL has the advantage that the return to the current page can only be inserted where it is needed. For example, where you often call a sequence of subprograms, which you have taken care to collect on a specific page, you certainly don't want to waste instructions between one and the next, unnecessarily restoring the current page, which will only be done when the sequence exits.

Also for PIC10/12/16 There is also a pseudo **LGOTO instruction** that has the same function in relation to unconditional jumping.

Particular care must be taken in cases where you intend to insert sections of Assembly code in C or BASIC sources, in order to avoid program crashes due to incorrect manipulation of the pages.

Although this part may seem rather obscure for now, we will see with repeated examples that things are less complicated than they seem.
The comprehension of the operations necessary for the addressing of pages or banks in PICs is quite a complex topic and requires to have in mind the mechanisms related to the ***Program Counter*** and the limitations of the PCL/PCLAT registers.

We can also not use macros and macro commands:

```
; calibrate internal oscillator
      movwf   OSCCAL
      Pagesel Init
      Goto    Init

; subroutine
      Pagesel DelayW10ms
      Goto    DelayW10ms
```

Equally, we have to condition every other jump, so that it is independent of the page:

```
; Loop
      Pagesel Loop
      Goto    Loop
```

Let's adjust the source.

We then adjust the rest of the source, even if the instructions and structure are unchanged, inserting the page management for each label directed to the program memory.

```
; Skip to Main
      Pagesel Main
      Goto    Main

; Vector for Subroutine
Delay10ms                  ; delay W x 10ms
      Pagesel Delay10ms_W
      Goto    Delay10ms_W
```

Note that, when a label defining a section of code appears, the CODE directive must be repeated :

```
; ******************************************************************
;                                MAIN PROGRAM
;
Main      TAILS
; Reset Initializations
      CLRF    GPIO          ; GPIO preset latch to
      Goto    Mainloop
```

The rest of the instructions are identical to what we saw above, with the only inclusion of automatic page selections.

We note that:

- A #include is no longer required for the subroutine: the "inclusion" is now done through the linker operation, invoked by a different compiler command.
- you don't need to declare RAM because the necessary RAM is already provisioned in the subroutine

With this we have inserted the elements necessary for a modularity of the source and that allow the compiler to generate the executable file regardless of the situation of use of memory resources.

Let's try to clarify the functions of page selection: let's take for example the first one in the text:

```
; Skip to Main
      Pagesel Main
      Goto     Main
```

The jump to Main has a reason to exist because, as we mentioned in the previous exercise, Baselines have strong limitations in the placement of subroutines due to the limited possibility of opcodes and PC.
A possible solution mentioned is to place the subroutine calls at the top of page 0 and invoke them indirectly with a goto. Then, immediately after the oscillator calibration instruction and the return of the Main later in memory.
This could lead him off the page, so let's adjust the situation with the first `pageel` . At this point, **the Main label can be located in any memory location**.

And we also need to adapt the page to the subroutines, since these can be compiled on any page.

```
; Vector for Subroutine
Delay10ms                 ; delay W x 10ms
      Pagesel Delay10ms_W
      Goto    Delay10ms_W
```

Calling the **DelayW10ms** subroutine has similar problems, because the compiler can place it anywhere in program memory. With the page selection command, access is done in the following sequence:

- the program invokes **call Delay10ms**
- this brings the PC to the location of the **Delay10ms label**
- the page is adjusted (if you are with the **Delay10ms_W** on a different page)
- a **goto** brings the PC to the beginning of the **Delay10ms_W**
- At its end, the **retlw** returns the execution to the next point after the call, the address of which was placed on the stack since the **initial** call

And so the calls to the delay subroutine look like this:

```
; Wait 100ms
      movlw    .10
      Pagesel Delay10ms
      Call     Delay10ms

; turns off LEDs
      LED_OFF

; Wait 400ms
      movlw    .40
      Pagesel Delay10ms
      Call     Delay10ms
```

The loop is closed with:

```
      Pagesel Mainloop
       Goto     Mainloop
```

where a **pagesel** causes the current position of the PC to be adjusted to the **goto mainloop** following.

Even if it seems somewhat intricate, it is only a matter of time to grasp its aims and mechanisms and thus make it a normal practice.

A few thoughts

With this example, we are faced with a process that is not common to find exemplified on the WEB, since it presupposes a reasonable knowledge of the problem.
This is therefore rather unusual for those who have already done something in Assembly with PIC at an amateur level, but it should not be considered an "oddity" dedicated only to professionals.

In general, this happens because such modularity is not necessary in simple programs and it is possible to build programs even of a certain complexity without it.

However **, it is the only way to be able to work in Assembly on large programs without excessive difficulty and, even on small programs, to be able to reuse code already written in other similar applications.** The use of modules and libraries is, in practice, the only way to write effective, non-dispersive, readable and reasonably maintained sources.

In later tutorials, we'll use both this method and a less "formal" one. It is up to you to understand the advantages and disadvantages and create your own way of working that is functional to the result to be obtained.

Modular Programming - Compilation

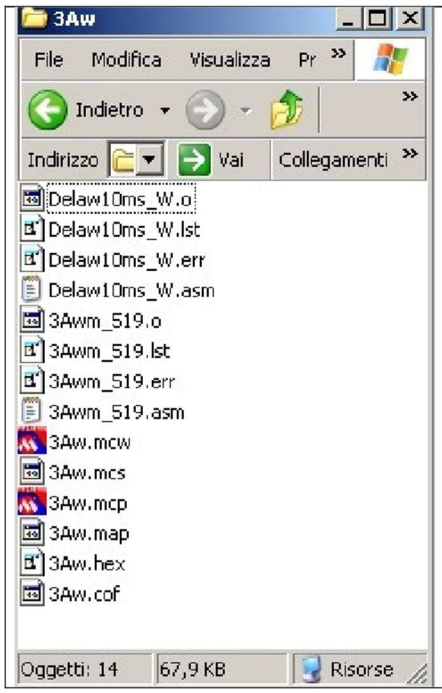
The complete design of this modular version is available.

In addition to the main, all the modules that will be called up must also be available to the compiler.

To compile you need to select not the ***Make button***  , but ***BuildAll***  which starts the build, but including all the source files in the project. The same commands can be given from the drop-down menu that unfolds from the ***Project*** item in the main menu.

This operation, in which the Linker comes into play, results in the creation of ***.o***, ***.err*** and ***.lst files*** for each ***.asm*** file present.
The ***.o*** files are used to generate a single ***.hex*** related to the compilation This

can be seen by looking at the folder for the project.



When we open the folder that contains the ***3Aw_519.mcp*** project, we see numerous files appear, whose presence and function have already been mentioned above.

We find, in addition to ***.mcp files***, ***.mcw*** and ***.mcs*** created by MPLAB for project management, a series of other files that result from the activity of Assembler and Linker:

- .asm ***sources***
- .lst ***Compilation Listing***
- .err ***error list***
- hexadecimal for ***.hex programming***
- .o ***object files***
- Descriptive file of ***.map resources***

Files are present for each ***.asm*** source that participated in the build

We can edit (without modifying it...), for example, the ***.lst*** file related to the subroutine and observe its contents, of which we report a part

```
0000                    00034 Delay10ms_W
0000   ???? ????       00035         Banksel D3       ; 2 extra
```

```
0002    00??            00036        movwf   D3
0003    0CCE            00037 dlwlp0 movlw   0xCE     ; 9993 cycles
0004    00??            00038        movwf   D1
0005    0C08            00039        movlw   0x08
0006    00??            00040        movwf   D2
0007    02??            00041 dlwlp1 decfsz  D1,F
0008    0A??            00042         Goto   $+2
0009    02??            00043        decfsz  d2,f
000A    0A??            00044         Goto   dlwlp1
000B    02??            00045        decfsz  D3,F     ; 2 cycles
000C    0A??            00046         Goto   dlwlp0
000D    0800            00047        retlw   0
```

It can be seen that the **banksel d3** directive is undefined (**???? ????** instead of hexadecimal values), since the address of **d3** is not yet defined. The same is true for instruction encoding that refers to the other memory locations used by the routine (**??** as the word seeds of the hexadecimal code of the statement).

The completion of the codes will be carried out by the clinker in the creation of the ***overall .hex*** of the project.

Let's see the action by editing (always taking care not to modify it) the .map file, which lists the sections of memory used and their addresses:

```
MPLINK 4.49, Linker
Linker Map File - Created Thu Sep 18 18:49:14 2014

                               Section Info
                  Section       Type    Address    Location Size(Bytes)
                  ---------  ---------  ---------  ---------  ---------
                  RES_VEC       tails   0x000000    program   0x00000a
                   .cinit     Romdata   0x000005    program   0x000004
                   .tails       tails   0x000007    program   0x000042
   .config_0FFF_3AWM_519_O      tails   0x000fff    program   0x000002
               .udata_shr     That is   0x000007       date   0x000001
                           why it is
                                  so
                           important
                   .udata     That is   0x000010       date   0x000003
                           why it is
                                  so
                           important

                              Program Memory Usage
                               Start         End
                            ---------   ---------
                             0x000000    0x000027
                             0x000fff    0x000fff
     41 out of 1093 program addresses used, program memory utilization is 3%
```

Among other things, it also reports the percentage usage of program memory.
The file also contains the list of symbols used in the compilation:

```
                        Symbols - Sorted by Name
           Name     Address    Location    Storage  File
      ---------   ---------   ---------   --------- ---------
     Delay10ms_W   0x00001a    program      Extern C:\3Aw\Delaw10ms_W.asm
            Main   0x000007    program      static C:\3Aw\3Awm_519.asm
     _.code_0008   0x000022    program      static C:\3Aw\Delaw10ms_W.asm
          dlwlp0   0x00001d    program      static C:\3Aw\Delaw10ms_W.asm
          dlwlp1   0x000021    program      static C:\3Aw\Delaw10ms_W.asm
        Mainloop   0x00000a    program      static C:\3Aw\3Awm_519.asm
              D1   0x000010       date      static C:\3Aw\Delaw10ms_W.asm
              D2   0x000011       date      static C:\3Aw\Delaw10ms_W.asm
```

```
          D3    0x000012        date      static C:\3Aw\Delaw10ms_W.asm
        sGPIO   0x000007        date      static C:\3Aw\3Awm_519.asm
```

The symbols used are listed by name and the membership of the program memory or data memory is specified, with an indication of the file in which they appear.

MPLAB and the MPASM Assembler macro provide a very effective system for dealing with modular programming as well.

It should be noted that .o files can be external modules for other compilations, as well as no longer needing to be recompiled per se, being already in the form of a relocatable object.

.

The MPLAB.

The entire working environment (project) for MPLAB 8.83 is provided to allow a better approach to compilation.

Other versions

You can easily check the versions for the various 12F508/509 and 16F505/526

As far as 10F200/202 is concerned, MPASM, using the modular elements seen, would generate an error in compilation:

```
Message[312] C:\DELAY10MS_W.ASM 35 : Page or Bank selection not needed for this
device. No code generated.
```

This is a Message and not an Error, but the executable is not produced anyway.
This is because the affected PICs have program memory on a single page and FSR on a single bank, so they don't need banksel and pagesel.
In fact, due to the small size of the memory, modular encoding is not strictly necessary, but if you want to implement it, you will need to eliminate references to banks and pages.

Note

If you read the compile exit message carefully, you'll notice the presence of a message like this:

```
Message[301] C:\PROGRAM FILES\MICROCHIP\MPASM SUITE\P12F519. INC 33 : MESSAGE:
(Processor-header file mismatch. Verify selected)
```

This derives from the ***processorname.inc*** that is inserted into the relocatable procedure, and that does not match that of the main compilation. This is necessary in the form of compilation adopted to provide the clinker with the necessary elements; In fact, the ***processorname.inc*** of any Baseline is enough. However, the situation does not create any problems for compilation and the output files are generated correctly.

Delay10ms_W.asm

```
;*****************************************************************
;-----------------------------------------------------------------
; Delay10ms_W.asm
;
;    Title          : Delay subroutine for 4MHz clocks.
;                     Relocatable version.
;                     Performs 9.995ms N loops; N passed with W.
;                     Total time is (N x 9.995ms)+ 6us
;    PIC            :Baseline
;    Support        : MPASM
;    Version        : 1.0
;    Date           : 01-05-2013
;    Hardware ref.  :
;    Author         :Afg
;
;*****************************************************************
; ###############################################################################

        #include <p12F519.inc>     ; any GLOBAL

        Baseline Delay10ms_W

; ###############################################################################
;                          RAM
;
     UDATA             ; general purpose RAM
D1    Res   1          ; Delay counters
D2    Res   1
D3    Res   1

;###############################################################################
            CODE

Delay10ms_W
       Banksel D3       ; 2 extra movwf
       cycles  D3
DLWLP0 movlw   0xCE     ; 9993 MovWF
       Cycles  D1
       movlw   0x08
       movwf   D2
DLWLP1 decfsz D1,F
        Goto   $+2
       decfsz d2,f
        Goto   dlwlp1
       decfsz d3,f      ; 2 goto
        cycles dlwlp0
       retlw   0        ; 4 extra cycles including call

;*****************************************************************
      END
```

12F519 - 3Aw_519.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 2A1_519
;                     Alternately flashes two LEDs connected to
;                     GP4 and GP5 at a rate of 1/2 second
;                     Modular code.
;    PIC            : 12F519
;    Support        : MPASM
;    Version        : 1.0
;    Date           : 01-05-2013
;    Hardware ref.  :
;    Author         :Afg
;
;-----------------------------------------------------------------
;
; Pin use :
;  ---------------
;       12F519 @ 8 pin
;
;                   |‾‾\/‾‾|
;             Vdd -|1    8|- Vss
;             GP5 -|2    7|- GP0
```

```
;             GP4 -|3    6|- GP1
;        GP3/MCLR -|4    5|- GP2
;                  |______|
;
;    Vdd                1: ++
;    GP5/OSC1/CLKIN     2: Out  LED5
;    GP4/OSC2           3: Out  LED4
;    GP3/! MCLR/VPP     4:
;    GP2/T0CKI          5:
;    GP1/ICSPCLK        6:
;    GP0/ICSPDAT        7:
;    Vss                8: --
;
;*****************************************************************
;=================================================================
;                   DEFINITION OF PORT USE
;
; Shadow definitions to avoid RMW
; sGPIO map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|--------|
;|LED5 |LED4 | in |      |     |     |
;
;#define    sGPIO,GP0    ;
;#define    sGPIO,GP1    ;
;#define    sGPIO,GP2    ;
;#define    sGPIO,GP3    ; Input only
#define    LED1 sGPIO,GP4    ; LED between pins
and Vss #define              LED2 sGPIO,GP5    ;
LED between pin and Vss
;
; ################################################################
```

```
        LIST      p=12F519          ; Processor definition
        #include <p12F519.inc>

        Radix     DEC

        EXTERN    Delay10ms_W

; ********************************************************************
;====================================================================
;                           CONFIGURATION
;
; Internal oscillator, no WDT, no CP, pin4=MCLR;
 __config  _IntRC_OSC & _WDTE_OFF & _CP_OFF & _CPDF_OFF & _MCLRE_ON


; ********************************************************************
;                          LOCAL RAM MEMORY
;
       UDATA_SHR         ; shared RAM
sGPIO   RES 1            ; shadow for GPIO

; ********************************************************************
;                         LOCAL MACROS
; Controls for the
LED1_ON LED   MACRO
        Bsf     LED1
        Bcf     LED2
        movf    sGPIO, w
        movwf   GPIO
          ENDM
LED2_ON   MACRO
        Bcf     LED1
        Bsf     LED2
        movf    sGPIO, w
        movwf   GPIO
          ENDM

; ********************************************************************
;                           RESET ENTRY
;
; Reset Vector
RES_VEC    TAILS   0x00

; Calibrate Internal Oscillator
        MOMOVWF OSCCAL
        pagesel Main
        goto    Main

; Delay
subroutine
        Pagesel Delay10ms_W
        Goto    Delay10ms_W

; ********************************************************************
;                          MAIN PROGRAM
;
       TAILS
```

```
; Reset Initializations
Main        CLRF    GPIO            ; GPIO preset latch to 0

; TRISGPIO        --001111        GP4/5 out
        movlw   b'11111100'
        Tris    GPIO             ; To the    direction
                                 Registry

Mainloop:                       ; <---|
; Lights up                     ;     |
LEDs                            ;     |
        LED1_ON                 ;     |
                                ;     |
; Wait 500ms    .50             ;     |
        movlw
        Pagesel Delay           ;     |
        Call    Delay           ;     |
                                ;     |
 ; turns off                    ; Loop
        LEDs
        LED2_ON                 ;     |
                                ;     |
; Wait 500 ms                   ;     |
        movlw   .50             ;     |
        Pagesel Delay           ;     |
        Call    Delay           ;     |
                                ;     |
; Loop                          ;     |
        Pagesel Mainloop        ;     |
        Goto    Mainloop        ;->--|

;*****************************************************************
;                               THE END
        END
```

12F508-509 - 3Aw_5089.asm

```
;*****************************************************************
; 3Aw_509.asm
;-----------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 2A1_519
;                     Alternately flashes two LEDs connected to
;                     GP4 and GP5 at a rate of 1/2 second
;                     Modular code.
;    PIC            : 12F519
;    Support        : MPASM
;    Version        : 1.0
;    Date           : 01-05-2013
;    Hardware ref.  :
;    Author         :Afg
;
;-----------------------------------------------------------------
;  Pin use :
;  ----------------
;       12F508-509 @ 8 pin
;
;                    |‾‾\/‾‾|
;              Vdd -|1    8|- Vss
;              GP5 -|2    7|- GP0
;              GP4 -|3    6|- GP1
;         GP3/MCLR -|4    5|- GP2
;                    |______|
;
;    Vdd                 1: ++
;    GP5/OSC1/CLKIN      2: Out   LED at Vss
;    GP4/OSC2            3: Out   LED at Vss
;    GP3/! MCLR/VPP      4:
;    GP2/T0CKI           5:
;    GP1/ICSPCLK         6:
;    GP0/ICSPDAT         7:
;    Vss                 8: --
;
;*****************************************************************
;=================================================================
;                    DEFINITION OF PORT USE
;
; Shadow definitions to avoid RMW
; sGPIO map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|--------|
;|LED5 |LED4 | in |      |     |     |
;
;#define    sGPIO,GP0     ;
;#define    sGPIO,GP1     ;
;#define    sGPIO,GP2     ;
;#define    sGPIO,GP3     ; Input only
#define    LED1 sGPIO,GP4  ; LED between pin and
                                          Vss
#define    LED2 sGPIO,GP5  ; LED between pin and
                                          Vss
;
```

```
; ###########################################################################
```

```
    #ifdef ___12F509
        LIST      p=12F509          ; Processor Definition
        #include <p12F509.inc>
    #endif
    #ifdef ___12F508
       LIST      p=12F508           ; Processor definition
        #include <p12F508.inc>
    #endif

        Radix     DEC

        EXTERN    Delay10ms_W

; ######################################################################
;======================================================================
;                           CONFIGURATION
;
;Internal oscillator, no WDT, no CP, pin4=MCLR
 __config  _IntRC_OSC & _WDT_OFF & _CP_OFF & _MCLRE_ON


; ######################################################################
;                          LOCAL RAM MEMORY
;
       UDATA_SHR         ; shared RAM
sGPIO   RES 1            ; shadow for GPIO

; ######################################################################
;                        LOCAL MACROS
; Controls for the
LED1_ON LED   MACRO
        Bsf     LED1
        Bcf     LED2
        movf    sGPIO, w
        movwf   GPIO
          ENDM
LED2_ON   MACRO
        Bcf     LED1
        Bsf     LED2
        movf    sGPIO, w
        movwf   GPIO
          ENDM

; ######################################################################
;                           RESET ENTRY
;
; Reset Vector
RES_VEC    TAILS   0x00

; Calibrate Internal Oscillator
        MOMOVWF OSCCAL
        pagesel Main
        goto    Main

; Delay
subroutine
        Pagesel Delay10ms_W
```

```
        Goto    Delay10ms_W

; ###########################################################################
;                         MAIN PROGRAM
;
       TAILS
; Reset Initializations
Main        CLRF    GPIO            ; GPIO preset latch to 0

; TRISGPIO        --001111        GP4/5 out
        movlw   b'11111100'
        Tris    GPIO                ; To the    direction
                                    Registry

Mainloop:                       ; <---|
; Lights up                     ;     |
LEDs                            ;     |
        LED1_ON                 ;     |
                                ;     |
; Wait 500ms    .50             ;     |
        movlw
        Pagesel Delay           ;     |
        Call    Delay           ;     |
                                ;     |
 ; turns off                    ; Loop
        LEDs
        LED2_ON                 ;     |
                                ;     |
; Wait 500 ms                   ;     |
        movlw   .50             ;     |
        Pagesel Delay           ;     |
        Call    Delay           ;     |
                                ;     |
; Loop                          ;     |
        Pagesel Mainloop        ;     |
        Goto    Mainloop        ;->--|

;***********************************************************************
;                               THE END
        END
```

16F526-505 - 3Aw_526.asm

```
;*****************************************************************
; 3Aw_526.asm
;-----------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 3Aw
;                     Alternately flashes two LEDs connected to
;                     GP4 and GP5 at a rate of 1/2 second
;                     Modular code.
;    PIC            : 16F526-16F505
;    Support        : MPASM
;    Version        : 1.0
;    Date           : 01-05-2013
;    Hardware ref.  :
;    Author         :Afg
;
;-----------------------------------------------------------------
;  Pin use :
;  ----------------
;       16F505 - 16F526 @ 14 pin
;
;                    |__\/__|
;              Vdd -|1   14|- Vss
;              RB5 -|2   13|- RB0
;              RB4 -|3   12|- RB1
;         RB3/MCLR -|4   11|- RB22
;              RC5 -|5   10|- RC0
;              RC4 -|6    9|- RC1
;              RC3 -|7    8|- RC2
;                   |______|
;
;    Vdd                        1: ++
;    RB5/OSC1/CLKIN             2: Out   LED at Vss
;    RB4/OSC2/CLKOUT            3: Out   LED at Vss
;    RB3/! MCLR/VPP             4:
;    RC5/T0CKI                  5:
;    RC4/[C2OUT]                6:
;    RC3                        7:
;    RC2/[Cvref]                8:
;    RC1/[C2IN-]                9:
;    RC0/[C2IN+]               10:
;    RB2/[C1OUT/AN2]           11:
;    RB1/[C1IN-/AN1/]ICSPC     12:
;    RB0/[C1IN+/AN0/]ICSPD     13:
;    Vss                       14: --
;
;    [ ] only 16F526
;
;*****************************************************************
;===============================================================
;            DEFINITION OF PORT USE
;
;P ORTC not used
;
```

```
; PRTB map
; | 5 | 4 | 3 | 2 | 1 | 0 |
; |-----|-----|-----|-----|-----|--------|
; | LED | LED | MCLR|     |     |     |
;
#define      LED2 PORTB,RB5     ; LED between pin and
                                Vss
#define    LED1 PORTB,RB4     ; LED between pin
                                and Vss
;#define        PORTB,RB2      ; MCLR
;#define        PORTB,RB2
;#define        PORTB,RB1
;#define        PORTB,RB0

; ######################################################################
;                            PROCESSOR SELECTION
 #ifdef __16F526
      LIST p=16F526
      #include <p16F526.inc>
  #endif
  #ifdef __16F505
      LIST p=16F505
      #include <p16F505.inc>
 #endif

; ######################################################################
;                             CONFIGURATION
;
 #ifdef __16F526
; Internal Oscillator, 4MHz, No WDT, No CP, MCLR
 __config _IntRC_OSC_RB4 & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF & _CPDF_OFF
&
MCLRE_ON
 #endif


 #ifdef __16F505
; Internal Oscillator, 4MHz, No WDT, No CP, MCLR
 __config _IntRC_OSC_RB4EN & _WDT_OFF & _CP_OFF & _MCLRE_ON
 #endif


; ######################################################################
;                           LOCAL RAM MEMORY
;
       UDATA_SHR         ; shared RAM
sGPIO   RES 1            ; shadow for GPIO

; ######################################################################
;                         LOCAL MACROS
;
; Controls for LED1
LED       MACRO
        Bcf     LED2
        Bsf     LED1
        movf    sGPIO,w
        movwf   PORTB
```

```
            ENDM
LED2        MACRO
          Bcf       LED1
```

```
        Bsf     LED2
        movf    sGPIO,w
        movwf   PORTB
          ENDM

; ######################################################################
;                          RESET ENTRY
;
; Reset Vector
RESVEC      ORG     0x00

; MOWF Internal Oscillator
        Calibration OSCCAL

; Delay
subroutine
        Pagesel Delay10ms_W
        Goto    Delay10ms_W

; ######################################################################
;                         MAIN PROGRAM
;
Main
; Reset Initializations
     CLRF    PORTB         ; GPIO preset latch to 0

; TRIS         --001111    RB5/4 out
     movlw   b'11001111'
     Tris    PORTB         ; To the Management
                           Register

;=====================================================================
Mainloop:                  ; <<--<<<--|
; Lights up LEDs           ;          |
     LED1                  ;          |
                           ;          |
; Wait 100ms                           |
     movlw   .10           ; 10 x 10ms = 100ms
     pagesel Dealy
     Call    Delay         ;          |
                           ;          |
; turns off LEDs                   Loop
     LED2                  ;          |
                           ;          |
; Wait 400ms                           |
     movlw   .40           ; 40 x 10ms = 400ms
     pagesel Delay
     Call    Delay         ;          |
                           ;          |
; Loop                     ;          |
    Goto    Mainloop      ; >>-->>---|

;*********************************************************************
;                             THE END
        END
```