# MICROCHIP MPLAB C18 C COMPILER

# GUÍA DEL ESTUDIANTE

# *1. Tipos de datos*

## Tipos enteros

El compilador C18 soporta los tipos de datos enteros del estándar ANSI. Los rangos de cada tipo se muestran a continuación:

| Tipo | tamaño | Rango | |
|---|---|---|---|
| char | 8 bits | -128 | 127 |
| Signed char | 8 bits | -128 | 127 |
| Unsigned char | 8 bits | 0 | 255 |
| int | 16 bits | -32,768 | 32,767 |
| Unsigned int | 16 bits | 0 | 65,535 |
| short | 16 bits | -32,768 | 32,767 |
| Unsigned short | 16 bits | 0 | 65,535 |
| Short long | 24 bits | -8,388,608 | 8,388,607 |
| Unsigned short long | 24 bits | 0 | 16,777,215 |
| long | 32 bits | -2,147,483,648 | 2,147,483,647 |
| Unsigned long | 32 bits | 0 | 4,294,967,295 |

## Tipos en punto flotante

Se admiten formatos de punto flotante double o float, basados en el estándar IEEE-754. A continuación se muestran los rangos de ambos tipos:

| Tipo | Bits | Exp. | | Rango | |
|---|---|---|---|---|---|
| float | 32 bits | -126 | 128 | $2^{-126} \approx 1.17549435e - 38$ | $2^{128} * (2-2^{-15}) \approx 6.80564693e + 38$ |
| double | 32 bits | -126 | 128 | $2^{-126} \approx 1.17549435e - 38$ | $2^{128} * (2-2^{-15}) \approx 6.80564693e + 38$ |

## Almacenamiento de los datos "endianness"

El ordenamiento de los datos en una variable multi-byte se realiza en little-endian. El byte menos significativo ocupa la posición más baja.

```
#pragma idata test=0x0200
long l=0xAABBCCDD;
```

Da como resultado:

| Dirección | 0x0200 | 0x0201 | 0x0202 | 0x0203 |
|---|---|---|---|---|
| Contenido | 0xDD | 0xCC | 0xBB | 0xAA |

## *2. Clases de almacenamiento*

Se soportan los siguientes tipos de almacenamiento definido en el estándar ANSI: auto, extern, register, static y typedef.

### Almacenamiento overlay

Además se introduce el almacenamiento *overlay*, que se aplica a variables locales sólo no se aplica a parámetros de funciones, definición de funciones o variables globales. El linker MPLINK intentará superponer variables locales tipo *overlay* sobre las mismas posiciones de memoria para funciones que no pueden activarse a la vez. Por ejemplo:

```
int f (void)
{
overlay int x = 1;
return x;
}
int g (void)
{
overlay int y = 2;
return y;
}
```

Si f y g nunca se activan juntas x e y pueden compartir la mismas posiciones de memoria. La ventaja de usar *ovelays* es que estas variables se sitúan estáticamente lo que significa que es necesario menos instrucciones para acceder a ellas, mientras que la cantidad de memoria requerida es menor que en el caso de haberlas definido directamente como *static*, al estar superpuestas.

Una función recursiva no puede contener variables de tipo *overlays.*

El formato por defecto para las variables locales es auto. Esto puede ser modificado explícitamente mediante los prefijos *static* u *overlay*.

### Argumentos para funciones

Los parámetros para las funciones pueden ser almacenados como auto o static. Un parámetro auto se almacena en la pila software que implementa el compilador. Uno static se considera global y habilita el acceso directo generando menos código. Los parámetros static son posibles sólo si el compilador esta en modo no-extendido. Los parámetros por defecto se definen como auto.

## *3. Calificadores del almacenamiento*

Además de los calificadores ANSI (const, volatile) el MPLAB introduce los siguientes tipos nuevos: far, near, rom y ram. A continuación se muestra la posición donde se almacena la variable asociada en función del calificador.

| | **rom** | **ram** |
|---|---|---|
| far | Cualquier posición de memoria de | Cualquier posición de memoria de |

| | programa | datos (es el valor por defecto) |
|---|---|---|
| `near` | Cualquier posición de memoria de programa menor de 64 KB | En las 0x5f posiciones más bajas de memoria RAM. Se permite acceso en modo banco. |

Las variables tipo RAM residen en la memoria de datos y dependiendo del calificador far/near, se supondrá que están en cualquier banco o restringidas a la posiciones más bajas de la RAM (0-0x5f) que permite el acceso en modo banco.

Las variables tipo ROM residen en memoria de programa. Si se utiliza el calificativo far la variable puede estar en cualquier posición de ROM y si es un puntero puede contener direcciones por encima de los 64KB. Al contrario si es near la variable se sitúa en posiciones menores a 64KB o si es un puntero puede acceder sólo hasta posiciones por debajo de 64KB.

Ejemplos de definición de punteros:

| Tipo de puntero | Ejemplo | tamaño |
|---|---|---|
| Puntero a datos | `char * dmp;` | 16 bits |
| Puntero cercano a memoria de programa | `rom near char * npmp;` | 16 bits |
| Puntero lejano a memoria de programa | `rom far char * fpmp;` | 24 bits |

# *4. Divergencias del ISO*

## Tamaño de los operandos

Por defecto el compilador C18 realiza las operaciones aritméticas utilizando el tamaño del operando más grande implicado en la operación. Por ejemplo:

```
unsigned char a, b;
unsigned i;
a = b = 0x80;
i = a + b; /* ISO require que i == 0x100, pero en C18 i == 0 */
```

En el caso de las constantes se elige el tipo de datos más pequeño que pueda representar el valor sin desbordamiento. Por ejemplo:

```
#define A 0x10 /* A se considera un char*/
#define B 0x10 /* B se considera un char*/
#define C (A) * (B)
unsigned i;
i = C; /* ISO require que i == 0x100, pero en C18 i == 0 */
```

Para que funciona conforme el ISO y todas las operaciones se realicen por defecto con tamaño de 16 bits (int) hay que compilar con la opción –Oi.

## Constantes numéricas

Las constantes se definen en diferentes formatos dependiendo del prefijo:

0x (hexadecimal), 0 (octal), 0b (binario).

### Constantes de tipo cadena

Las variables almacenadas en memoria de programa suelen ser cadenas de caracteres. Este tipo de variable se define como: *const rom char[]*

La sección .stringtable de la sección romdata contiene todas las cadenas constantes de memoria de programa. También se pueden declarar tablas de caracteres. Ejemplos:

```
rom const char table[][20] = { "string 1", "string 2", "string 3", "string 4"
};
```

Table es una variable que contiene 80 caracteres en memoria de programa, ya que está compuesta de 4 cadenas de 20 caracteres cada una.

Los punteros a variables no son compatibles a menos que apunten a objetos compatibles situados en el mismo tipo de memoria.

Ejemplo de copia de una cadena en ram a otra en rom:

```
void str2ram(static char *dest, static char rom *src)
{
      while ((*dest++ = *src++) != '\0');
}
```

## *5 Extensiones del lenguaje*

### Ensamblador en línea

Se puede inserter código en ensamblador mediante _asm y _endasm. Aunque dentro de un bloque en ensamblador no se deben usar ni directivas del ensamblador, las etiquetas deben acabar con :, no se soporta direccionamiento indirecto, la constantes se especifican como en C.

Ejemplo:

```
_asm
                                /* User assembly code */
            MOVLW 10                  // Move decimal 10 to count
            MOVWF count, 0
                                /* Loop until count is 0 */
      start:
            DECFSZ count, 1, 0
            GOTO done
            BRA start
      done:
_endasm
```

## *6 Secciones*

Una **sección** es una porción de una aplicación situada en una dirección de memoria específica. Las secciones pueden contener datos o código y se pueden situar en memoria de datos o programa. Hay dos tipos de secciones para cada tipo de memoria.

Memoria de programa
- `code` – contiene instrucciones
- `romdata` – contiene variables y constantes

Memoria de datos
- `udata` – contiene variables no inicializadas estáticas
- `idata` – contains variables inicializadas estáticas

Las declaración de sección mediante la palabra reservada #pragma cambia la sección en la cual actualmente el compilador está insertando el programa.

Las **secciones** se pueden definir como:

Absolutas: cuando se especifica su posición de comienzo.

Directive de sección:

```
      # pragma udata [attribute-list] [section-name [=address]]
|     # pragma idata [attribute-list] [section-name [=address]]
|     # pragma romdata [overlay] [section-name [=address]]
|     # pragma code [overlay] [section-name [=address]]
```

*attribute-list*:

```
      access
|     overlay
```

*section-name*: identificador
*address*: constante

Existen 4 secciones por defecto:

| Tipo | Nombre |
|---|---|
| code | .code_*filename* |
| romdata | .romdata_*filename* |
| udata | .udata_*filename* |
| idata | .idata_*filename* |

El código se engloba dentro de la sección code. Las variables en rom dentro de la sección romdata. Las variables en ram dependiendo de si están inicializadas o no en udata o idata. En el siguiente programa de ejemplo cada una de las variables y funciones pertenecen a una sección distinta dependiendo de su definición:

```
rom int ri;
rom char rc = 'A';

int ui;
char uc;

int ii = 0;
char ic = 'A';

void foobar (void)
{
static rom int foobar_ri;
static rom char foobar_rc = 'Z';
...
}
void foo (void)
{
```

```
static int foo_ui;
static char foo_uc;
...
}
void bar (void)
{
static int bar_ii = 5;
static char bar_ic = 'Z';
...
}
```

Secciones asociadas a cada objeto:

| Object | Section Location |
|---|---|
| ri | romdata |
| rc | romdata |
| foobar_ri | romdata |
| foobar_rc | romdata |
| ui | udata |
| uc | udata |
| foo_ui | udata |
| foo_uc | udata |
| ii | idata |
| ic | idata |
| bar_ii | idata |
| bar_ic | idata |
| foo | code |
| bar | code |
| foobar | code |

## Atributos

El atributo **access** le indica al compilador que debe incluir la sección el espacio definido por el modo de acceso a datos *access bank* sin necesidad de modificar el banco actual de la ram. Ejemplo:

```
#pragma udata access my_access
/* acceso sin bancos */
near unsigned char av1, av2;
```

El atributo **overlay** permite que otras secciones se sitúen en las mismas posiciones físicas. Esto permite conservar memoria situando variables en las mismas posiciones, siempre que no se activen simultáneamente. Pero hay restricciones:

1. Las dos secciones deben pertenecer a ficheros distintos.
2. Deben tener el mismo nombre
3. La dirección debe ser la misma, y si una utiliza el modo access la otra también.

file1.c:
```
#pragma udata overlay my_overlay_data=0x1fc
/* 2 bytes en 0x1fc y 0x1fe */
int int_var1, int_var2;
```
file2.c:
```
#pragma udata overlay my_overlay_data=0x1fc
/* 4 bytes en 0x1fc */
long long_var;
```

### Dirección

Después de la directiva #pragma code todo el código generado se asigna a ese sección hasta que se encuentre otra directiva #pragma code. Un ejemplo de una directiva que permite situar código en una determinada posición es:

```
#pragma code my_code=0x2000
```

Después se puede volver al segmento por defecto:

```
#pragma code high_vector=0x08
...
/*
* A continuación vuleve a la sección code por defecto
*/
#pragma code
```

De forma similar se pueden definir las posiciones ocupadas por los datos en la ram:

```
#pragma udata my_new_data_section=0x120
```

o en la rom:

```
#pragma romdata const_table
const rom char my_const_array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

## *7 Interrupciones*

La directiva *#pragma interruplow nombre* y *#pragma interrup nombre* definen rutinas de servicio de interrupción (ISR). La primera de baja primoridad *interruplow* y la segunda de alta prioridad *#pragma interrup*. Una interrupción suspende la ejecución de una aplicación, salva el contexto actual y transfiere el control a una ISR.

Los registros WREG, BSR y STATUS deben salvarse al procesar una interrupción. Una interrupción de alta prioridad utiliza los registros de sombra (pila de un nivel) para almacenar estos registros. Mientras que una ISR de baja prioridad utiliza la pila software.
Ejemplo:

```
void foo(void);
...

#pragma interrupt foo

void foo(void)
{
/* interrupción */
}
```

Las ISR son funciones como cualquier otra, pero con las restricciones de que:

- No pueden devolver ni aceptar parámetros.
- Las variables globales que utilice se deben declarar como *volatile*.
- No se pueden invocar desde otros puntos del programa.

El C18 no sitúa automáticamente las ISR en las posiciones de los vectores de interrupción. Normalmente se pone un goto para llevar el control desde los vectores de interrupción hasta las subrutinas ISR. Por ejemplo:

```
#include <p18cxxx.h>
void low_isr(void);
void high_isr(void);
/*
* PIC18 low interrupt vector 00000018h.
* 18h es la dirección del vector de interrupción para baja prioridad
* La instrucción GOTO low_isr salta a la función ISR low_isr
*/
#pragma code low_vector=0x18

void interrupt_at_low_vector(void)
{
_asm GOTO low_isr _endasm
}


#pragma code /* vuelve a la sección de código por defecto */
#pragma interruptlow low_isr

void low_isr (void)
{
/* ... */
}
/*

* PIC18 high interrupt vector 00000008h.
* 8h es la dirección del vector de interrupción para alta prioridad
* La instrucción GOTO high_vector salta a la función ISR high_vector
*/
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
_asm GOTO high_isr _endasm
}

#pragma code /* vuelve a la sección de código por defecto */
#pragma interrupt high_isr
void high_isr (void)
{
/* ... */
}
```

Las ISR por defecto sólo salvan el mínimo contexto (registros W,STATUS y BSR), para salvar otras variables o zonas de memoria hay que indicarlo explícitamente. Esta acción se realiza mediante la opción save=. Ejemplo para salvar el contenido de la variable global myint:

```
#pragma interrupt high_interrupt_service_routine save=myint
```

También se pueden salvar registros internos o secciones completas de datos. Para salvar toda una sección de datos, por ejemplo sección mydata.

```
#pragma interrupt high_interrupt_service_routine save=section("mydata")
```

Si una ISR llama a otra subrutina se debe salvar la sección de datos temporales utilizadas por las funciones ".tmpdata".

```
#pragma interrupt high_interrupt_service_routine save=section(".tmpdata")
```

Si una ISR llama a otra subrutina que devuelve un parámetro de 16 bits, el registro interno del micro PROD debe de salvarse. Esto se indica durante la definición de la ISR.

```
#pragma interruptlow low_interrupt_service_routine save=PROD
```

Si utiliza la librería matemática o llama a funciones que devuelven valores de 24 o 32 bits, se debe salvar la sección de datos "MATH_DATA".

```
#pragma interrupt high_interrupt_service_routine save=section("MATH_DATA")
```

## *8 Declaración de los registros internos*

Cada microcontrolador tiene un fichero asociado de cabecera donde se incluyen las declaraciones externas de sus registros internos. Se incluyen en el proyecto mediante la sentencia #include <pic18f2550.h>. Por ejemplo dentro de estos ficheros incluirá la declaración externa del registro PORTA:

```
extern volatile near unsigned char PORTA;
```

PORTA está declarada como variable de 8 bits y también se declara como una unión de bits, como los pines se pueden denominar de forma distinta dependiendo del uso se definen todos los nombres que se les puede asignar:

```
extern volatile near union {
struct {
unsigned RA0:1;
unsigned RA1:1;
unsigned RA2:1;
unsigned RA3:1;
unsigned RA4:1;
unsigned RA5:1;
unsigned RA6:1;
} ;
struct {
unsigned AN0:1;
unsigned AN1:1;
unsigned AN2:1;
unsigned AN3:1;
unsigned T0CKI:1;
unsigned SS:1;
unsigned OSC2:1;
} ;
struct {
unsigned :2;
unsigned VREFM:1;
unsigned VREFP:1;
unsigned :1;
unsigned AN4:1;
unsigned CLKOUT:1;
} ;
struct {
unsigned :5;
unsigned LVDIN:1;
} ;
} PORTAbits ;
```

Para asignar un valor al registro el código sería:

```
PORTA = 0x34;       /* 0x34 a los 8 bits del puerto */
PORTAbits.AN0 = 1; /* pone a uno el pin cero AN0 */
PORTAbits.RA0 = 1; /* pone a uno el pin cero RA0, es el mismo pin que en la
línea anterior*/
```

Además de la definición de los registros internos los ficheros incluyen las siguientes macros que ejecutan directamente algunas instrucciones en ensanblador:

| Macro | Acción |
|---|---|
| `Nop()` | (`NOP`) |
| `ClrWdt()` | (`CLRWDT`) |
| `Sleep()` | (`SLEEP`) |
| `Reset()` | (`RESET`) |
| `Rlcf(var, dest, access)` | Rota *var* a la izq. con el carry |
| `Rlncf(var, dest, access)` | Rota *var* a la izq sin el carry |
| `Rrcf(var, dest, access)` | Rotates *var* a la derecha con el carry |
| `Rrncf(var, dest, access)` | Rotates *var* a la derecha sin el carry |
| `Swapf(var, dest, access)` | Intercambia los nibbles de *var* |

Los ficheros de definición de los registros internos son ficheros en ensamblador que contienen la declaración de todos los registros internos del micro. El registro de definición una vez compilado genera un fichero objeto que debe ser enlazado junto con la aplicación. Los ficheros se encuentran dentro de una librería por ejemplo para el 18f2550 esta librería es p18f2550.lib.

### Código de arranque

El C18 introduce un rutina de arranque que comienza en la posición del vector de reset, dirección 0x00000. Esta rutina iniciliza los registros FSR y la pila por software. También las secciones idata y posteriormente salta a la función main. Esta rutina de Star-up está contenida en los ficheros c018i.o o c018i_e.o para el modo de compilación extendido o no extendido.

## *9 Recursos utilizados por el compilador*

Hay ciertos recursos y posiciones de la memoria del microcontrolador que son usados por el compilador y no que no pueden ser utilizados por los programas de usuario. En la siguiente tabla se muestran estos recursos. La tercera columna indica cuando el compilador automáticamente salva el contenido de ese recurso al procesar una interrupción.

| Recursos | Uso | Se salva automáticamente la llamar a ISR |
|---|---|---|
| `PC` | Control de la ejecución | X |
| `WREG` | Cáclculos intermedios | X |
| `STATUS` | Resultado de las operaciones | X |
| `BSR` | Selección de banco | X |
| `PROD` | Resultado de multiplicación, paso de parámetros a funciones | |

| | | |
|---|---|---|
| section `.tmpdata` | Cálculos intermedios | |
| `FSR0` | Puntero a RAM | X |
| `FSR1` | Puntero a pila | X |
| `FSR2` | Puntero a marco | X |
| `TBLPTR` | Dirección para acceso a memoria de programa | |
| `TABLAT` | Valores para acceso a memoria de programa | |
| `PCLATH` | Recarga del PC | |
| `PCLATU` | Recarga del PC | |
| section `MATH_DATA` | Argumentos, valores de vuelta y posiciones temporales para la librería matemática. | |

# *10 Compilación*

## Directorios del compilador

El directorio donde se instala el C18 contiene los siguientes subdirectorios donde se almacena los programas y ficheros asociados:

| Directorio | Descripción |
|---|---|
| `bin` | Contiene los ejecutables del compilador y el linker. |
| `Example` | Contiene el código fuente ejemplo. |
| `doc` | Contiene los documentos de ayuda. |
| `h` | Tiene los ficheros de cabecera para la librería estándard de C. |
| `lib` | Tiene la librería estándar (`clib.lib` o `clib_e,lib`), las librerías específicas de cada micro (`p18`*`xxxx`*`.lib` o `p18`*`xxxx`*`_e.lib`, donde *`xxxx`* es el identificador del pic) y los módulos de arranque (`c018.o`, `c018_e.o`, `c018i.o`, `c018i_e.o`, `c018iz.o`, `c018iz_e.o`). |
| `lkr` | Contiene los scripts del linker.. |
| `mpasm` | Contiene el ensamblador MPASM y los ficheros de cabecera MPLAB C18 (`p18xxxx.inc`) y los ficheros de cabecera para el ensamblador. |
| `src` | Es el código fuente de las librerías en formato C y ensamblador. |

## Programas asociados

| Ejecutables | Descripción |
|---|---|
| `mcc18.exe` | Es la shell del compilador. Toma el fichero entrada en C (i.e., `file.c`) e invoca al ejecutable del compilador Extendedido o No-extendedido. |

| | |
|---|---|
| `mcc18-extended.exe` | Este es el ejecutable del compilador Extendido. |
| `mcc18-traditional.exe` | Este es el ejecutable del compilador No-Extendido. |
| `cpp18.exe` | Este es el preprocesador. |
| `mplink.exe` | Este es el programa de invocado del linker.. Toma los ficheros objeto, las librerías y el script del linker y se lo pasa al ejecutable del linker `_mplink.exe`. |
| `_mplink.exe` | Este es el linker. |
| `mplib.exe` | Este es el librarian. |

## Opciones para el compilador

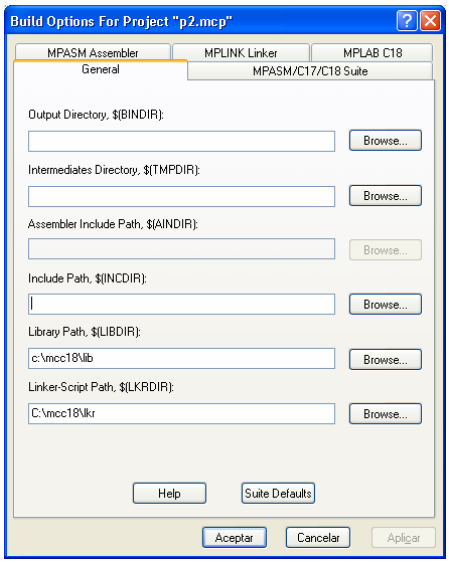
Las opciones del compilador más relevantes son:

| Opción | Descripción |
|---|---|
| `-?, --help` | Ayuda |
| `-I=<path>` | Añade 'path' a el path de los ficheros de cabecera |
| `-fo=<name>` | Nombre del fichero objeto |
| `-fe=<name>` | Nombre del fichero de errores |
| `-k` | Asignar el tipo char a el tipo unsigned char |
| `-ls` | Tamaño de la pila (puede ocupar varios bancos) |
| `-ms` | Modo pequeño de la memoria ( valor por defecto) |
| `-ml` | Modo grande de memoria. |
| `-O, -O+` | Habilita todas las optimizaciones (por defecto) |
| `-O-` | Deshabilita todas las optimizaciones. |
| `-Oi+` | Habilita que como mínimo las operaciones se realizan sobre rangos de 16 bits. |
| `-Oi-` | Deshabilita que como mínimo las operaciones se realizan sobre rangos de 16 bits. |
| `-sca` | Habilita la definicón de variables locales como auto (por defecto). Sólo para modo No-extendido. |
| `-scs` | Habilita la definición de variables locales como static por defecto. Sólo para modo No-extendido. |
| `-sco` | Habilita la definición de variables locales como overlay por defecto. Sólo para modo No-extendido. |
| `-Oa+` | Modo por defecto para las variables es access bank. |
| `-Oa-` | Modo por defecto para las variables no es access bank. |
| `-p=<processor>` | El identificador del pic |
| `-D<macro>[=text]` | Define una macro |
| `-w={1\|2\|3}` | Nivel de los warnings (defecto = 2) |
| `--extended` | Genera código para modo Extendido. |
| `--no-extended` | Genera código para modo No-Extendido. |

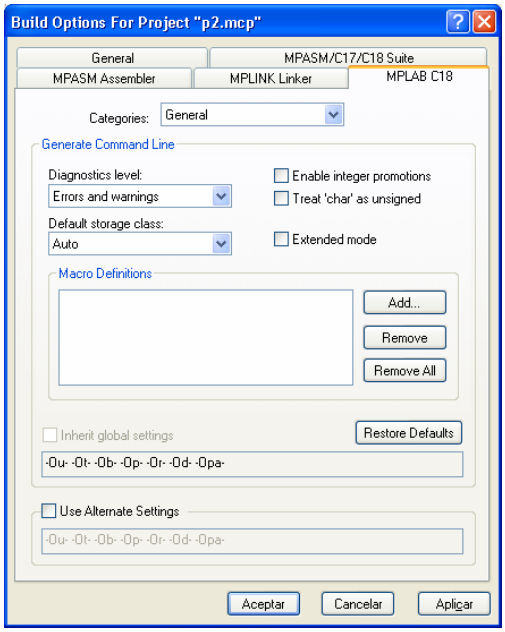
## Opciones a través del MPLAB

Las opciones para el compilado se introducen en la ventana de opciones del proyecto: *Project->Build options->Project*

Lo primero que hay que comprobar son los directorios de la librería y de los scripts del linker. Estos deben ser: *c:\mcc18\lib* y *c:\mcc18\lkr*, respectivamente.

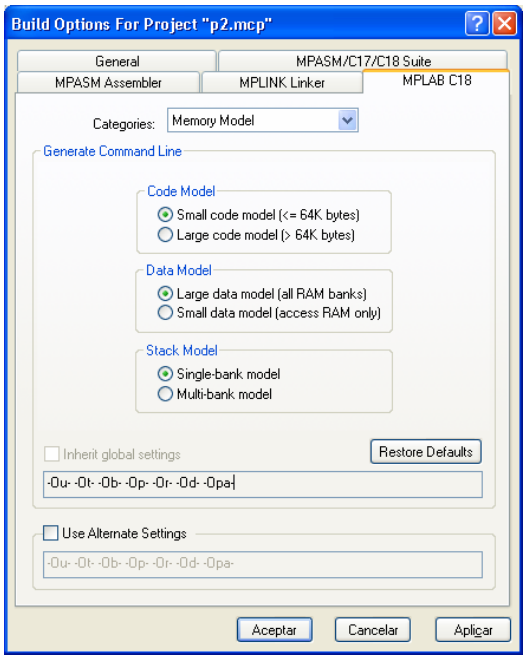



Opciones generales del compilador:



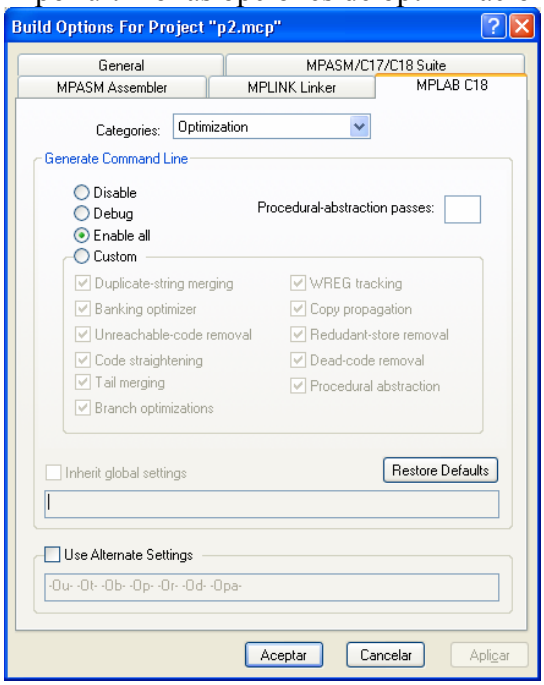

Las tres casillas del menú anterior habilitan las opciones: `--extended -Oi+ -k`, explicadas anteriormente. También se puede elegir entre los modos de definición de variables locales: `-sca`, `-scs`, `-sco`. Y el nivel de información sobre errores y warnings.

Opciones de tamaño de la memoria:




Se puede elegir entre modelo grande o pequeño para la memoria de código, datos y pila las opciones son respectivamente: -ms/-ml, -0a+ , -ls.

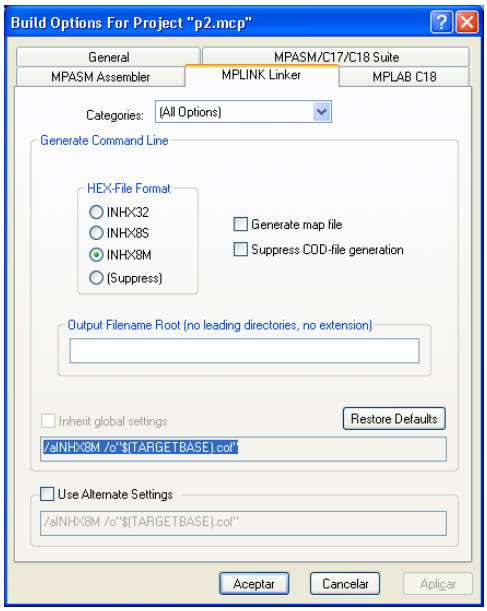
Y por último las opciones de optimización:




Se pueden seleccionar diversos grados de optimización. Para tener información más detallada sobre los tipos de optimización consultar el manual del compilador en el directorio c:\mcc18\doc.

## Linker

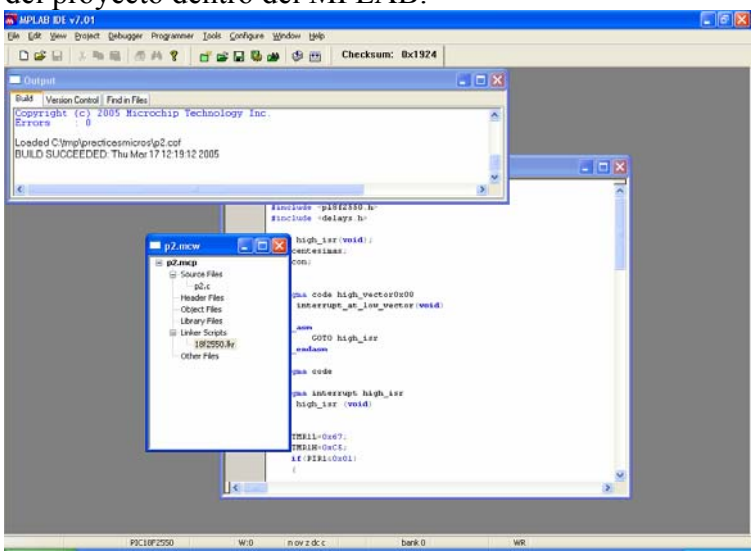
Las opciones del linker son:




Con las que se controla los ficheros de salida y su formato.

## Ficheros para compilar el proyecto

Los ficheros que se van a utilizar para compilar el proyecto, se detallan en la ventana del proyecto dentro del MPLAB:




En los ficheros fuente (source files) en este caso tendremos un o varios ficheros en c (Ejemplo p2.c). Los ficheros de cabecera (header files) se especifican dentro del fichero fuente y no es necesario añadirlos. Los módulos de arranque y las librerías estándar y del procesador estarían dentro de las clases (object y library files), pero no hace falta especificarlas porque están incluídas en el script del linker (linker script). Si hubiera otros ficheros con módulos o librerías que hay que enlazar juntos habría que añadirlos al proyecto.

El fichero de script de linker indica que librerías debe utilizar, la inicialización del micro y como debe distribuir el código en la memoria. Existen para cada micro 4 ficheros script diferentes dependiendo del tipo de proyecto que se vaya a realizar:

| | |
|---|---|
| 18fxxxx.lkr | Proyectos de aplicación en modo No-extendido. |
| 18fxxxxi.lkr | Para proyectos en modo No-Extendido que se van a depurar con el MPLAB ICD 2. La "i" indica que se van a guarder recursos para el debugger MPLAB ICD 2. |
| 18fxxxx_e.lkr | Proyectos de aplicación en modo Extendido. |
| 18fxxxxi_e.lkr | Para proyectos en modo Extendido que se van a depurar con el MPLAB ICD 2. La "i" indica que se van a guarder recursos para el debugger MPLAB ICD 2. |

## *11 Librerías*

Existen dos tipos de librerías:

La librería estándar es "clib.lib (para modo no-extendido) o clib_e.lib (para modo extendido)". Esta librería es independiente de microcontrolador y el código fuente se encuentra en los directorios:

- `src\traditional\math`
- `src\extended\math`
- `src\traditional\delays`
- `src\extended\delays`
- `src\traditional\stdclib`
- `src\extended\stdclib`

La librería específica contiene funciones escritas para un miembro individual de la familia 18. El nombre de estas librerías de pende del micro *p*processor`.lib` (No-extendido) y *p*processor`_e.lib` (Extendido). Ejemplo p18f2550.lib. El código fuente se encuentra en los directorios:

- `src\traditional\pmc`
- `src\extended\pmc`
- `src\traditional\proc`
- `src\extended\proc`

Algunas funciones de librería interesantes son:

### Delay1TCY

Función: Retraso de periodo igual al periodo de 1 instrucción.
Include: `delays.h`
Prototipo: `void Delay1TCY( void );`

### Delay10TCYx

Función: Retraso en múltiplos de 10 instrucciones indicado por el parámetro *unit*. Retraso= (***unit*** * 10) periodos de instrucción.
Include: `delays.h`
Prototype: `void Delay10TCYx( unsigned char` ***unit*** `);`

### Delay100TCYx

Function: Retraso en múltiplos de 100 instrucciones indicado por el parámetro *unit*. Retraso= (***unit*** * 100) periodos de instrucción.

Include: `delays.h`
Prototype: `void Delay100TCYx( unsigned char` ***unit*** `);`

### Delay1KTCYx

Function: Function: Retraso en múltiplos de 1000 instrucciones indicado por el parámetro *unit*. Retraso= (***unit*** * 1000) periodos de instrucción.
Include: `delays.h`
Prototype: `void Delay1KTCYx( unsigned char` ***unit*** `);`

### Delay10KTCYx

Function: Retraso en múltiplos de 10000 instrucciones indicado por el parámetro *unit*. Retraso= (***unit*** * 10000) periodos de instrucción.
Include: `delays.h`
Prototype: `void Delay10KTCYx( unsigned char` ***unit*** `);`

Para más información sobre las funciones de librería consultar el documento MPLAB-C18-Libraries.pdf dentro del directorio *c:\mcc18\doc*

## *12 Ejemplos*

1º) Práctica 1 escrita en C.

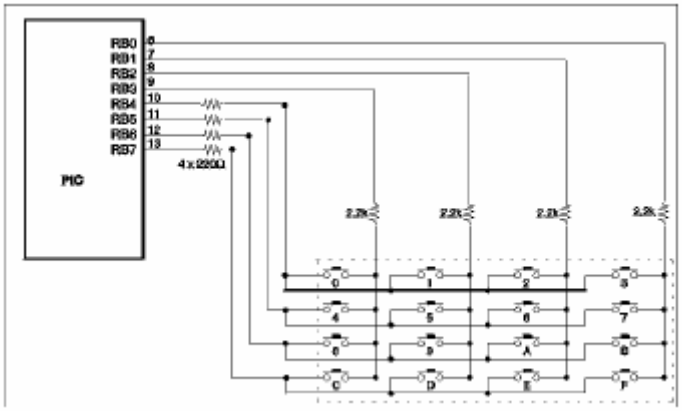
```
#include <p18f2550.h>
#include <delays.h>

/* cálculo del valor para retraso de 1 seg.
       freq=6000000
       periodo_instruccion=4/freq
       unit=1seg/(10000*periodo_instruccion)=150*/

#define retraso1s   150

void fleds(void);
void display(void);

void main(void)
{
       ADCON1=0x0F;
       TRISB=0x00;
       TRISA=0b00011111;

       while(1)
       {
              if(PORTA&0x10)                          //bit4 a 1
                     fleds();
              else
                     display();                // bit4 a 0

              Delay10KTCYx(retraso1s);         //retraso 1s, delay_ms(10);
       }
}

void fleds(void)
{
              switch(PORTA&0x0F)
              {
                     case 0: PORTB=0b00111111;
                     break;
                     case 1: PORTB=0b00000110;
```

```
                    break;
                    case 2: PORTB=0b01011011;
                    break;
                    case 3: PORTB=0b01001111;
                    break;
                    case 4: PORTB=0b01100110;
                    break;
                    case 5: PORTB=0b01101101;
                    break;
                    case 6: PORTB=0b01111100;
                    break;
                    case 7: PORTB=0b00000111;
                    break;
                    case 8: PORTB=0b01111111;
                    break;
                    case 9: PORTB=0b01110011;
                    break;
                    case 10: PORTB=0b10111111;
                    break;
                    case 11: PORTB=0b10000110;
                    break;
                    case 12: PORTB=0b11011011;
                    break;
                    case 13: PORTB=0b11001111;
                    break;
                    case 14: PORTB=0b11100110;
                    break;
                    case 15: PORTB=0b11101101;
      }
}

void display(void)
{
            switch(PORTA&0x0F)
            {
                    case 0: PORTB=0b00000000;
                    break;
                    case 1: PORTB=0b00000000;
                    break;
                    case 2: PORTB=0b00000001;
                    break;
                    case 3: PORTB=0b00000001;
                    break;
                    case 4: PORTB=0b00000011;
                    break;
                    case 5: PORTB=0b00000011;
                    break;
                    case 6: PORTB=0b00000111;
                    break;
                    case 7: PORTB=0b00000111;
                    break;
                    case 8: PORTB=0b00001111;
                    break;
                    case 9: PORTB=0b00001111;
                    break;
                    case 10: PORTB=0b00011111;
                    break;
                    case 11: PORTB=0b00011111;
                    break;
                    case 12: PORTB=0b00111111;
                    break;
                    case 13: PORTB=0b00111111;
                    break;
                    case 14: PORTB=0b01111111;
                    break;
                    case 15: PORTB=0b01111111;
      }
}
```

2º) Ejemplo que va encendiendo los bits del puerto b de uno en uno secuencialmente de forma ascendente y descendente. Para cambiar el led que está encendido se utiliza como periodo el tiempo en que tarda en desbordarse el temporizador 0 en modo temporizador de 16 bits sin divisor. Para detectar los desbordamientos del timer 0 se utiliza la interrupción.

```
#include <p18f2550.h>


#define ASCENDENTE 1
#define DESCENDENTE 0

void high_isr(void);

volatile int contador;
volatile char sentido;

#pragma code high_vector=0x08
void interrupt(void)
{
	_asm
		GOTO high_isr
	_endasm
}
#pragma code

#pragma interrupt high_isr
void high_isr (void)
{

	if(INTCON&0x04)
	{
		if(sentido)  PORTB=PORTB<<1;
		else PORTB=PORTB>>1;

		if(PORTB==0x80) sentido=DESCENDENTE;
		if(PORTB==0x01) sentido=ASCENDENTE;

		INTCONbits.TMR0IF=0;
	}
}

void main(void)
{

	RCONbits.IPEN=0;
	T0CON=0b10001000;
	INTCONbits.TMR0IE=1;
	INTCONbits.GIE=1;
	INTCONbits.PEIE=1;

	ADCON1=0x0F;
	contador=0;
	sentido=ASCENDENTE;

	TRISB=0;
	PORTB=1;
	while(1);
}
```

3º) Ejemplo teclado matricial. Lee una tecla y la escribe en el PortA.




```
#include "lcd.h"
#include <p18f2550.h>

#define freq        6000000
#define periodo     4/freq
#define ciclos100us (100/1000000)/periodo




char Lee_Tecla(void)
{
       char tecla,Temp,Temp1,Temp2;

       TRISB=0xF0;
       INTCON2=INTCON2&0x7F;      // bit INTCON2<7>=#RBPU habilita pull-ups
       PORTB=0x00;

       Temp=0x00;
       Temp1=PORTB;               // Comprueba la fila
       while ((Temp1 & 0x10) && (Temp<4))  //cuenta hasta llegar a un bit a '1'
       {
              Temp1>>=1;
              Temp++;
       }
       if (Temp<4)                //Fila correcta; se ha pulsado tecla
       {
              Temp2=Temp;         //Guardo Fila

              TRISB=0x0F;
              INTCON2=INTCON2&0x7F;
              PORTB=0x00;

              Temp=0x00;
              Temp1=PORTB;        // Comprueba la columna
              while ((Temp1 & 0x01) && (Temp<4))  // cuenta hasta llegar a ‘1’
              {
                     Temp1>>=1;
                     Temp++;
              }
              if (Temp<4)         //Columna correcta; cálculo la tecla…
              {
                     Temp1=Temp;
                     Temp2=(Temp2<<2)+Temp1;
                     tecla=Temp2;
                     return(tecla);
              }
              return(0xFF);
```

```
      }
      return(0xFF);
}



void main(void)
{

       ADCON1=0x0F;
       TRISA=0x00;
       while(1)
       {
              PORTA=Lee_Tecla();
              Delay100TCYx(ciclos100us);
                           //100*ciclos100us=>retraso 10ms;
       }
}
```