Placing a variable at an absolute address :

In embedded ANSI C programming it is useful to be able to place a specific variable at an absolute address. This is interesting for instance when a variable is defined to access a specific I/O register.
There are three ways to do it:

- Using an ANSI C macro
- Using the modifier @
- Using a data segment

(The following examples use the register map of the MC68HC912B32 MCU)

Using a C macro:

A C macro can be used to define a variable paced at a specific absolute address. This is done in the following way:

```
        #define PORTA  (*((volatile unsigned char*) (0x0000)))
```

The previous macro defines PORTA to be the content of a pointer to an unsigned character located at address 0x0000. This means PORTA is an unsigned character located at address 0x0000.

Access to the port are done in the following way:

```
              PORTA = 0x1F ;    /* the port A register is set at the value 0x1F */
```

This notation is portable on any compliant ANSI C compiler. But it has two drawbacks.

- There is no real variable allocated when you define your variables this way. That means that you will not be able to see them in a debugger data window.
- The linker does not have any idea, that there is a variable allocated there. It will not detect any overlap between the variables you have defined using a macro and the application global variables.

Using the modifier '@':

The compiler has been extended with an additional operator or modifier @. This modifier can be used to tell the compiler that a variable or constants are placed at a specific absolute address.
The modifier @ is used in the following way to define a register:

```
        volatile unsigned char FEETST @0x00F6 ;
```

The previous definition line defines FEETST to be an unsigned char allocated at address 0x00F6.
Access to the register FEETST can be performed in the following way:

```
              FEETST=0xD7;
```

Be Careful: the notation @0x00F6 is not ANSI C standard. It may not work on another compiler.

Using a Data Segment:

Using the segmentation pragmas from the compiler, one can also ensure that a specific variable is allocated at an absolute address. This is done in two steps. First the variable is defined in a specific segment and then the segment must be placed at the appropriate address
In your source file, the variable should be defined in the following way:

```
        #pragma DATA_SEG PORTB_SEG
```

```
volatile unsigned char portb;
#pragma DATA_SEG DEFAULT
```

The definition above means that an unsigned char (8 bits) variable called ‘portb’ is defined in a segment called PORTB_SEG.

Now this segment should be placed at the appropriated place in the PRM file. This is done in the following way:

```
SECTIONS
    PORTB_SEG = READ_WRITE   0x0001 SIZE 1;
```

This means that the segment PORTB_SEG is allocated at address 0x0001 (the address of the I\O PORT B register).
The variable can then be accessed in the following way:

```
portb = 0x1F ;   /* the port B register is set at the value 0x1F */
```

Be Careful: the data segmentation is not supported in the same way by all ANSI C compilers. You may need to modify your source code when you switch to another compiler.

Special Notes Regarding the I/O Register definition:

Volatile Modifier

We recombined to use the keyword ‘volatile’ to define all I/O port variables. An ANSI C compiler is not allowed to optimize accesses on variables defined as volatile. This way you are sure that some code is generated for each access to an I/O Port.

Accessing the single bits in the I/O Port:

Most of the time, one needs to access the different bits in an I/O register individually. This can be done using bitfields structure. Then additional macros can be defined to allow to access the different I/O port bits using their mnemonics.

Definition of the MC68HC912B32 FEETST (Flash EEprom Test Register):

```
volatile union {
  struct {
    unsigned char MWPR:1;
    unsigned char STRE:1;
    unsigned char VTCK:1;
    unsigned char FDISVFP:1;
    unsigned char FENLV:1;
    unsigned char HVT:1;
    unsigned char GADR:1;
    unsigned char FSTE:1;
    } FEETST_BITS;
  unsigned char FEETST_BYTE;
} FEETST1 @0x00F6;

/* Define mnemonic to access the register */
```

```
#define FEETST FEETST1.FEETST_BYTE

/* Define mnemonics to access the different bits */
#define MWPR FEETST1.FEETST_BITS.MWPR
#define STRE FEETST1.FEETST_BITS.STRE
#define VTCK FEETST1.FEETST_BITS.VTCK
#define FDISVFP FEETST1.FEETST_BITS.FDISVFP
#define FENLV FEETST1.FEETST_BITS.FENLV
#define HVT FEETST1.FEETST_BITS.HVT
#define GADR FEETST1.FEETST_BITS.GADR
#define FSTE FEETST1.FEETST_BITS.FSTE
```

You can initialize the different bits in the I/O register using the following notation:

```
{ …..
    MWPR=1;          /*  The register FEETST is set to the value 0xD7 */
    STRE=1;
    VTCK=1;
    FDISVFP=0;
    FENLV=1;
    HVT=0;
    GADR=1;
    FSTE=1;
}
```

The previous portion of code can be replaced by the following one. In this case the register is access as a whole.

```
{
    FEETST=0xD7;
}
```