Explicando o uso de sizeof

o **sizeof** é um operador muito utilizado no C para obter o tamanho em bytes de estruturas de dados, array e tipos primitivos de dados (int, char, long etc).

Por exemplo:

long x =  1L;

int tamanhoX = sizeof x;

Na maioria das arquiteturas de hardware, é o mesmo que atribuir 4 à variável **tamanhoX** (explico o porquê do uso adiante).

Note no programa que eu crio um tipo definido (**typedef**) para representar as informações de uma banda.

typedef struct

{

  char \*name;

  uint64\_t minFreq; // Min. frequency value for the band (unit 0.01Hz)

  uint64\_t maxFreq  // Max. frequency value for the band (unit 0.01Hz)

} Band;

**Que tamanho tem o tipo definido Band?**

Tamanho da variável Band é:  (o tamanho do **ponteiro** char para **nome**) + (o tamanho de **minFreq**) + (o tamanho de **maxFreq**)

- o tamanho de uma ponteiro é normalmente 4 bytes (para a maioria das arquiteturas de hardware);

- o tamanho de **minFreq** é 8 bytes (64 bits); e

- o tamanho de **maxFreq** também é 8 bytes.

Diante disso, o tamanho do tipo definido que criei seria: 4 + 8 + 8 = 20 bytes.

**Por que usar o operador sizeof?**

Há várias razões.

1)    Dispensa o cálculo do tamanho em bytes da estrutura ou tipo de dados que você estiver usando. Em tempo de compilação, o compilador fará a conversão é trabalhará com o tamanho correto considerando a plataforma de hardware corrente.

2)    Deixa o código mais fácil de manter. Por exemplo, imagine  que eu queira incluir mais um atributo na minha estrutura conforme mostrado a seguir

typedef struct

{

  char \*name;

  uint64\_t minFreq; // Min. frequency value for the band (unit 0.01Hz)

  uint64\_t maxFreq; // Max. frequency value for the band (unit 0.01Hz)

  uint64\_t offset;

} Band;

Observe no código acima a inclusão de **offset**. A mudança mostrada no código acima, não demanda a necessidade de recalcular o novo tamanho da estrutura se eu usar sizeof. **Isto é, sizeof agora retornará 28 bytes.**

 3)    Facilita a migração do código para outras plataformas de microcontroladores. Outros microcontroladores podem usar tamanho padrões de tipos primitivos diferentes do Arduino, por exemplo. Há plataformas que o tipo int é de 32 bits (4 bytes)  e não 16 bits (2 bytes) e há plataformas que usam diferentes tamanhos para ponteiros (dependendo como essas plataformas trabalham com endereçamento de memória).  Com o uso do sizeof, eu não preciso me preocupar com isso neste contexto (tamanho em bytes).

**Por fim, porque estou usando sizeof?**

A principal razão é a tabela de bandas. Veja a seguir:

// Band database. You can change the band ranges if you need.

// The unit of frequency here is 0.01Hz (1/100 Hz). See Etherkit Library at <https://github.com/etherkit/Si5351Arduino>

Band band[] = {

    {"MW   ", 50000000LLU, 170000000LLU, 45500000LU},

    {"SW1  ", 700000000LLU, 1000000000LLU, 45500000LU},

    {"SW1  ", 1000000000LLU, 2000000000LLU, 45500000LU},

    {"FM   ", 7600000000LLU, 10800000000LLU, 1075000000LLU},

    {"AIR  ", 10800000000LLU, 13500000000LLU, 1075000000LLU},

    {"VFH  ", 13500000000LLU, 16000000000LLU, 1075000000LLU}};

Note que o array band[] é do tipo Band (o tipo definido que eu criei).

O meu objetivo é saber quantos elementos tem esse array band[].

Você agora deve pensar: ora, é só contar. E se fizer isso encontrará 5.

Naturalmente você poderia fazer isso:

Band band[5] = {

    {"MW   ", 50000000LLU, 170000000LLU, 45500000LU},

    {"SW1  ", 700000000LLU, 1000000000LLU, 45500000LU},

    {"SW1  ", 1000000000LLU, 2000000000LLU, 45500000LU},

    {"FM   ", 7600000000LLU, 10800000000LLU, 1075000000LLU},

    {"AIR  ", 10800000000LLU, 13500000000LLU, 1075000000LLU},

    {"VFH  ", 13500000000LLU, 16000000000LLU, 1075000000LLU}};

Bom, isso não estaria errado e a maioria dos programadores fazem assim. Porém, a cada vez que você mudar essa tabela (e isso normalmente ocorre várias vezes), você deverá alterar também a quantidade de elementos do array band e ainda considerar essa nova quantidade em todo o código onde você precisar usá-la.

Para evitar esse desconforto e preocupação eu uso a expressão a seguir:

const int lastBand = (sizeof band / sizeof(Band)) - 1;

**O que ela faz?**

Calcula o número de elementos do array band[] e subtrai de 1 porque 4 é o último elemento do array (lembre-se que em C o primeiro elemento é o 0).

(sizeof band / sizeof(Band)) => Considerando o primeiro exemplo do tipo Band seria: (20 + 20 + 20 + 20 + 20) / 20 = 100 / 20 = 5 ou

considerando o segundo exemplo do tipo definido  Band seria (28 + 28 + 28 + 28 +28) / 28 = 140 / 28 = 5.

**Note que mesmo mudando a estrutura do tipo Band, o resultado do tamanho continuou 5 elementos. Com isso, posso aumentar o número de linhas da tabela (inserir mais bandas), incluir novos atributos (variáveis) e não precisarei fazer nada no resto do código. Em princípio, isso pode parecer complicado, porém, na proporção que os requisitos do sistema ficam mais complexos, essa abordagem pode ajudar.**

Só mais uma coisa que talvez possa gerar dúvida:

As constantes “MW.  “, “SW1 “, “FM  “ etc.. NÃO fazem parte da estrutura implícita do array do tipo definido **Band**. A estrutura apenas armazena ponteiros para estas constantes. Em outras palavras, elas não são somadas ao tamanho em bytes da estrutura. Apenas os ponteiros para essas constantes são.