

■ Uses for C's `offsetof()` Macro

Nigel Jones ■

C's seldom-used `offsetof()` macro can actually be a helpful addition to your bag of tricks. Here are a couple of places in embedded systems where the macro is indispensable, including packing data structures and describing how EEPROM data are stored.

If you browse through an ANSI C compiler's header files, you'll come across a very strange looking macro in `stddef.h`. The macro, `offsetof()`, has a horrid declaration. Furthermore, if you consult the compiler manuals, you'll find an unhelpful explanation that reads something like this:

The `offsetof()` macro returns the offset of the element name within the struct or union composite. This provides a portable method to determine the offset.

At this point, your eyes start to glaze over, and you move on to something that's more understandable and useful. Indeed, this was my position until about a year ago—whence the macro's usefulness finally dawned on me. I now kick myself for not realizing the benefits earlier—the macro could have saved me a lot of grief over the years. However, I console myself by realizing that I wasn't alone, since I'd never seen this macro used in any embedded code. Offline and online searches confirmed that `offsetof()` is essentially not used. I even found compilers that had not bothered to define it.

How `offsetof()` works

Before delving into the three areas where I've found the macro useful, it's necessary to discuss what the macro does, and how it does it.

The `offsetof()` macro is an ANSI-required macro that should be found in `stddef.h`. Simply put, the `offsetof()` macro returns the number of bytes of offset before a particular element of a struct or union.

The declaration of the macro varies from vendor to vendor and depends upon the processor architecture. Browsing through the compilers on my computer, I found the example declarations shown in Listing 1. As you can see, the definition of the macro can get complicated.

```
// Keil 8051 compiler
#define offsetof(s,m) (size_t)&(((s *)0)->m)

// Microsoft x86 compiler (version 7)
#define offsetof(s,m) (size_t)(unsigned long)&(((s *)0)->m)

// Diab Coldfire compiler
#define offsetof(s,memb) \
    ((size_t)((char *)&(((s *)0)->memb)-(char *)0))
```

Listing 1. A representative set of `offsetof()` macro definitions

Regardless of the implementation, the `offsetof()` macro takes two parameters. The first parameter is the structure name; the second, the name of the structure element. (I apologize for using a term as vague as “structure name.” I'll refine this shortly.) A straightforward use of the macro is shown in Listing 2.

```

typedef struct
{
    int    i;
    float  f;
    char   c;

} SFOO;

void main(void)
{
    printf("Offset of 'f' is %u", offsetof(SFOO, f));
}

```

Listing 2. A straightforward use of offsetof()

To better understand the magic of the offsetof() macro, consider the details of Keil's definition. The various operators within the macro are evaluated in an order such that the following steps are performed:

- ((s *)0): takes the integer zero and casts it as a pointer to s.
- ((s *)0)->m: dereferences that pointer to point to structure member m.
- &(((s *)0)->m): computes the address of m.
- (size_t)&(((s *)0)->m): casts the result to an appropriate data type.

By definition, the structure itself resides at address 0. It follows that the address of the field pointed to (Step 3 above) must be the offset, in bytes, from the start of the structure. At this point, we can make several observations:

- We can be a bit more specific about the term "structure name." In a nutshell, if the structure name you use, call it s, results in a valid C expression when written as (s *)0->m, you can use s in the offsetof() macro. The examples shown in Listings 3 and 4 will help clarify that point.
- The member expression, m, can be of arbitrary complexity; indeed, if you have nested structures, then the member field can be an expression that resolves to a parameter deeply nested within a structure
- It's easy enough to see why this macro also works with unions
- The macro won't work with bitfields, as you can't take the address of a bitfield member of a structure or union

Listings 3 and 4 contain simple variations on the usage of this macro. These should help you get you comfortable with the offsetof() basics.

```

struct sfoo
{
    int    i;
    float  f;
    char   c;

};

void main(void)
{
    printf("Offset of 'f' is %u", offsetof(struct sfoo, f));
}

```

Listing 3. A struct without a typedef

```

typedef struct
{
    long   l;
    short  s;

} SBAR;

```

```

typedef struct
{
    int    i;
    float  f;
    SBAR   b;

} SFOO;

void main(void)
{
    printf("Offset of 'l' is %u", offsetof(SFOO, b.l));

}

```

Listing 4. Nested structs

Now that you understand the semantics of the macro, it's time to take a look at a few use examples.

struct padding bytes

Most 16-bit and larger processors require that data structures in memory be aligned on a multibyte (for example, 16-bit or 32-bit) boundary. Sometimes the requirement is absolute, and sometimes it's merely recommended for optimal bus throughput. In the latter case, the flexibility is offered because the designers recognized that you may wish to trade off memory access time with other competing issues such as memory size and the ability to transfer (perhaps via a communications link or direct memory access) the memory contents directly to another processor that has a different alignment requirement.

For cases such as these, it's often necessary to resort to compiler directives to achieve the required level of packing. As the C structure declarations can be quite complex, working out how to achieve this can be daunting. Furthermore, after poring over the compiler manuals, I'm always left with a slight sense of unease about whether I've really achieved what I set out to do.

The most straightforward solution to this problem is to write a small piece of test code. For instance, consider the moderately complex declaration given in Listing 5.

```

typedef union
{
    int    i;
    float  f;
    char   c;

    struct
    {
        float  g;
        double h;
    } b;

} UFOO;

void main(void)
{
    printf("Offset of 'h' is %u", offsetof(UFOO, b.h));

}

```

Listing 5. A union containing a struct

If you need to know where field `b.h` resides in the structure, then the simplest way to find out is to write some test code such as that shown in Listing 5.

This is all well and good, but what about portability? Writing code that relies on offsets into structures can be risky—particularly if the code gets ported to a new target at a later date. Adding a comment is of course a good idea—but what one really needs is a means of forcing the compiler to tell you if the critical members of a structure are in the wrong place.

Fortunately, one can do this using the `offsetof()` macro and the technique in Listing 6.

```
typedef union
{
    int    i;
    float f;
    char   c;

    struct
    {
        float g;
        double h;
    } b;
} UFOO;

static union
{
    char wrong_offset_i[offsetof(UFOO, i) == 0];
    char wrong_offset_f[offsetof(UFOO, f) == 0];
    ...
    char wrong_offset_h[offsetof(UFOO, b.h) == 2]; // Error
};
```

Listing 6. An anonymous union to check struct offsets

The technique works by attempting to declare a union of one-char arrays. If any test evaluates to false, its array will be of zero size, and a compiler error will result. One compiler I tested generated the specific error “Invalid dimension size [0]” on the line defining array `wrong_offset_h[]`.

Thus the `offsetof()` macro can be used both to determine and to validate the packing of elements within C structs.

Nonvolatile memory layouts

Many embedded systems contain some form of nonvolatile memory, which holds configuration parameters and other device-specific information. One of the most common types of nonvolatile memory is serial EEPROM. Normally, such memories are byte addressable. The result is often a serial EEPROM driver that provides an API that includes a read function that looks like this:

```
ee_rd(uint16_t offset, uint16_t nBytes, uint8_t * dest)
```

In other words, read `nBytes` from offset `offset` in the EEPROM and store them at `dest`. The problem is knowing what offset in EEPROM to read from and how many bytes to read (in other words, the underlying size of the variable being read). The most common solution to this problem is to declare a data structure that represents the EEPROM and then declare a pointer to that struct and assign it to address `0x00000000`. This technique is shown in Listing 7.

```
typedef struct
{
    int    i;    float f;
    char   c;

} EEPROM;

EEPROM * const pEE = 0x00000000;

ee_rd(&(pEE->f), sizeof(pEE->f), dest);
```

Listing 7. Accessing data in serial EEPROM via a pointer

This technique has been in use for years. However, I dislike it precisely because it does create an actual pointer to a variable that supposedly resides at address 0. In my experience, this can create a number of problems including:

- Someone maintaining the code can get confused into thinking that the EEPROM data structure really does exist
- You can write perfectly legal code (for example, `pEE->f = 3.2`) and get no compiler warnings that what you're doing is disastrous
- The code doesn't describe the underlying hardware well

A far better approach is to use the `offsetof()` macro. An example is shown in Listing 8

```
typedef struct
{
    int    i;
    float  f;
    char   c;
} EEPROM;

ee_rd(offsetof(EEPROM,f), 4, dest);
```

Listing 8. Use `offsetof()` to access data stored in serial EEPROM

However, there's still a bit of a problem. The size of the parameter has been entered manually (in this case "4"). It would be a lot better if we could have the compiler work out the size of the parameter as well. No problem, you say, just use the `sizeof()` operator. However, the `sizeof()` operator doesn't work the way we would like it to. That is, we cannot write `sizeof(EEPROM.f)` because EEPROM is a data type and not a variable.

Normally, one always has at least one instance of a data type so that this is not a problem. In our case, the data type EEPROM is nothing more than a template for describing how data are stored in the serial EEPROM. So, how can we use the `sizeof()` operator in this case? Well, we can simply use the same technique used to define the `offsetof()` macro. Consider the definition:

```
#define SIZEOF(s,m) ((size_t) sizeof(((s *)0)->m))
```

This looks a lot like the `offsetof()` definitions in Listing 1. We take the value 0 and cast it to "pointer to s." This gives us a variable to point to. We then point to the member we want and apply the regular `sizeof()` operator. The net result is that we can get the size of any member of a typedef without having to actually declare a variable of that data type.

Thus, we can now refine our read from the serial EEPROM as follows:

```
ee_rd(offsetof(EEPROM, f), SIZEOF(EEPROM, f), &dest);
```

At this point, we're using two macros in the function call, with both macros taking the same two parameters. This leads to an obvious refinement that cuts down on typing and errors:

```
#define EE_RD(M,D) \
    ee_rd(offsetof(EEPROM,M), SIZEOF(EEPROM,M), D)
```

Now our call to the EEPROM driver becomes much more intuitive:

```
EE_RD(f, &dest);
```

That is, read `f` from the EEPROM and store its contents at location `dest`. The location and size of the parameter is handled automatically by the compiler, resulting in a clean, robust interface.

Protect nonvolatile memory

Many embedded systems contain directly addressable nonvolatile memory, such as battery-backed SRAM. It's usually important to detect if the contents of this memory have been corrupted. I usually group the data into a structure, compute a CRC (cyclic redundancy check) over that structure, and append it to the data structure. Thus, I often end up with something like this:

```

struct nv
{
    short    param_1;
    float    param_2;
    char     param_3;
    uint16_t crc;
} nvram;

```

The intent of the `crc` field is that it contain a CRC of all the parameters in the data structure with the exception of itself. This seems reasonable enough. Thus, the question is, how does one compute the CRC? If we assume we have a function, `crc16()`, that computes the CRC-16 over an array of bytes, then we might be tempted to use the following:

```

nvram.crc =
    crc16((char *) &nvram, sizeof(nvram)-sizeof(nvram.crc));

```

This code will only definitely work with compilers that pack all data on byte boundaries. For compilers that don't do this, the code will almost certainly fail. To see that this is the case, let's look at this example structure for a compiler that aligns everything on a 32-bit boundary. The effective structure could look like that in Listing 9.

```

struct nv
{
    short    param_1;           // offset = 0
    char     pad1[2];           // 2 byte pad
    float    param_2;           // offset = 4
    char     param_3;           // offset = 8
    char     pad2[3];           // 3 byte pad
    uint16_t crc;               // offset = 12
    char     pad3[2];           // 2 byte pad
} nvram;

```

Listing 9. An example struct for a compiler that aligns everything on a 32-bit boundary

The first two pads are expected. However, why is the compiler adding two bytes onto the end of the structure? It does this because it has to handle the case when you declare an array of such structures. Arrays are required to be contiguous in memory, too. So to meet this requirement and to maintain alignment, the compiler pads the structure out as shown.

On this basis, we can see that the `sizeof(nvram)` is 16 bytes. Now our naive code in Listing 9 computes the CRC over `sizeof(nvram) - sizeof(nvram.crc)` bytes = $16 - 2 = 14$ bytes. Thus the stored `crc` field now includes its previous value in its computation! We certainly haven't achieved what we set out to do.

```

struct nv
{
    struct data
    {
        short param_1;           // offset = 0
        float param_2;           // offset = 4
        char  param_3;           // offset = 8
    } data;

    uint 16_t crc;               // offset = 12
} nvram;

```

Listing 10. Nested data structures

The most common solution to this problem is to nest data structures as shown in Listing 10. Now we can compute the CRC using:

```

nvram.crc =

```

```
crc16((uint8_t *) &nvram.data, sizeof(nvram.data));
```

This works well and is indeed the technique I've used over the years. However, it introduces an extra level of nesting within the structure—purely to overcome an artifact of the compiler. Another alternative is to place the CRC at the top of the structure. This overcomes most of the problems but feels unnatural to many people. On the basis that structures should always reflect the underlying system, a technique that doesn't rely on artifice is preferable—and that technique is to use the `offsetof()` macro.

Using the `offsetof()` macro, one can simply use the following (assuming the original structure definition):

```
nvram.crc =  
    crc16((uint8_t *) &nvram, offsetof(struct nv, crc));
```

Keep looking

I've provided a few examples where the `offsetof()` macro can improve your code. I'm sure that I'll find further uses over the next few years. If you've found additional uses for the macro I would be interested to hear about them.

This article was published in the March 2004 issue of *Embedded Systems Programming*. If you wish to cite the article in your own work, you may find the following MLA-style information helpful:

Jones, Nigel. "Learn a new trick with the `offsetof()` macro" *Embedded Systems Programming*, March 2004.