www.phparch.com

April 2019
Volume 18 - Issue 4

php[architect]

The New Frontend Fundamentals

PHP[TEK] 2019 Conference

Tickets On Sale NOW!

Special Pricing for Individual Tracks

A series of tracks dedicated to teaching you
in-depth information about a specific topic
lead by hand-picked experts
chosen for their knowledge and presentation skills.
A truly unique learning experience!

May 21-23, 2019 ◆ Atlanta, GA

tek.phparch.com

CONFERENCE

Patterns in the Code

Edward Barnard

The PHP compiler source code includes many patterns that can be frustrating and intimidating, because they are so different from typical PHP code, until we understand the structure and context. C's preprocessor has a significant role in these unfamiliar patterns. We look at several of these patterns in the compiler's PHP Array implementation.

Working With the Code Base

Where shall we start? With a large code base, that's always a problem. We're focused on the array implementation, called hash tables in the source code. The three files of interest are:

- Zend/zend_types.h[1]—the different types of variables, but also hash table details.
- Zend/zend_hash.h[2]—here is the API information for using hash tables, but a whole lot more; we need to talk.
- Zend/zend_hash.c[3]—the array (hash table) implementation; it's opaque until you get used to it.

Don't forget the cross-reference tool lxr[4] we described in January 2019. It is your friend.

Function Declaration

The source code is difficult to follow until you get used to it. Let's take a few lines (Listing 1 `Zend/zend_hash.c` near line 1056) to illustrate the problem.

We looked at function signatures in January so that we can make educated guesses about line 1:

- `ZEND_API` implies this function signature is part of the internal API, i.e., that it's a public function callable from anywhere. (I guessed incorrectly; see below.)
- This function returns a pointer to a `zval` structure. Conceptually that's like how a PHP function/method can return a PHP object, which is a pointer to the object. (A `zval` is a general-purpose structure that can represent any type of PHP variable.)
- Tell the compiler to generate the "fast call" calling sequence if it can.
- The function requires three arguments—a `HashTable` structure pointer; a `Bucket` structure pointer; a `zend_string` structure pointer.

When terms, such as "structure" or "calling sequence," aren't clear to me, I use Google. Most of the concepts are well described on Wikipedia, YouTube, or somewhere. We'll continue explaining below.

1 *Zend/zend_types.h: https://phpa.me/php-src-zend-types*

2 *Zend/zend_hash.h: https://phpa.me/php-src-zend-hash-hdr*

3 *Zend/zend_hash.c: https://phpa.me/php-src-zend-hash*

4 *lxr: https://php-lxr.adamharvey.name/source/*

Let's use `lxr` to search `"define ZEND_API"` (with the quotes). We have six definitions, all similar. Let's check Zend/configure.ac[5]. The top line of the file says "Process this file with autoconf to produce a configure script." We'll talk about `autoconf` some other time.

The `ZEND_API` lines are:

```
#if defined(__GNUC__) && __GNUC__ >= 4
# define ZEND_API __attribute__ ((visibility("default")))
#else
# define ZEND_API
#endif
```

According to the GCC Wiki[6] "visibility" is a C++ feature that:

- Very substantially improves load times of your DSO (Dynamic Shared Object)
- Lets the optimizer produce better code
- Reduces the size of your DSO by five to 20 percent
- Lowers the chance of symbol collision

Listing 1. 01-set-bucket.c

```
1. ZEND_API zval* ZEND_FASTCALL zend_hash_set_bucket_key
   (HashTable *ht, Bucket *b, zend_string *key)
2. {
3.     uint32_t nIndex;
4.     uint32_t idx, i;
5.     Bucket *p, *arData;
6.
7.     IS_CONSISTENT(ht);
8.     HT_ASSERT_RC1(ht);
9.     ZEND_ASSERT(!(HT_FLAGS(ht) & HASH_FLAG_PACKED));
10.
11.    p = zend_hash_find_bucket(ht, key, 0);
12.    if (UNEXPECTED(p)) {
13.        return (p == b) ? &p->val : NULL;
14.    }
15.
16.    if (!ZSTR_IS_INTERNED(key)) {
17.        zend_string_addref(key);
18.        HT_FLAGS(ht) &= ~HASH_FLAG_STATIC_KEYS;
19.    }
```

5 *Zend/configure.ac: https://phpa.me/zend-configure*

6 *GCC Wiki: https://gcc.gnu.org/wiki/Visibility*

This explanation clearly has nothing to do with PHP's array implementation. In fact, `ZEND_API` doesn't appear to have anything to do with declaring an API at all! What it does is tell the C compiler how to best generate the object code.

This is the difficulty with a large code base. You'll need to jump down quite a number of rabbit holes to see how things are structured. At first, you won't be able to discern what's important at the moment and what isn't. In many cases, you can guess from the context and move on. Go back and drill down to the definition as needed. As we learn how more and more pieces fit together, we get a clearer picture of the whole. Eventually, it starts to click and make sense.

Variable Declaration

Lines 3-4 declare three variables `nIndex`, `idx`, and `i`. PHP best practice opts for long descriptive variable names. C practice is often the opposite. These days, typing speed is not the determining factor in how many lines of code are produced per day. When C was first in use, typing speed was a limiting factor; often the programmer typed the code across a slow connection. Picture typing your code on a phone screen, in an underground garage where you lose the cell phone signal every few minutes (just before clicking "save"), and you'll get the idea. Or, more commonly, the code was typed on a keypunch with no such thing as a backspace key. Whatever the reason, the "short variable name" habit has carried forward.

The variables are type `uint32_t` which is the same as `UNSIGNED INT` in MySQL. It's a 32-bit number with no negative numbers (no minus sign). The leftmost bit, normally the sign bit, has no particular meaning.

PHP has no similar concept. Neither does C, by the way, even though you see it here. C programmers have vast control over the C compiler and have wide latitude in telling it what object code to generate. `uint32_t` is a typedef[7], that is, a definition of a data type. The C standard library and POSIX reserve the suffix `_t`.

Listing 2. struct-bucket.c

```c
typedef struct _Bucket {
    zval              val; /* hash value (or numeric index)   */
    zend_ulong        h;   /* string key or NULL for numerics */
    zend_string      *key;
} Bucket;
#define getBucket(b, n) ((Bucket *)((b) + ((n) * sizeof(Bucket))))
```

Typedef

Line 5 declares two `Bucket` pointers. How do we know what a `Bucket` is? In PHP, we would look for the `Bucket` class definition. In C it is likely to be in a header file, and we'll use `lxr` to find it. Click on any of the results, and then click on the word `Bucket`, and the click takes you to the definition around line 229 of Zend/zend_types.h[8]. See Listing 2.

Line 5 tells us `Bucket` is a `typedef struct _Bucket`. This is a common idiom in C, and it's somewhat like declaring class properties in PHP.

The C keyword `typedef` means "create an alias." For example:

```c
typedef long scoped;
```

We have just created a data type called `scoped`. It is not a variable; it is a data type. In PHP we have `string`, `int`, `float`, and so on. In C we similarly have `char`, `long`, `double`. Now, instead of declaring a variable of type `long`, we can declare it of type `scoped`. It's weird but is used continuously in C header files. It's aimed at making the C code more intuitive and readable.

The C keyword `struct` describes a "structured" data type. In this case, `_Bucket` contains three fields. If we, therefore, declare a variable of type `Bucket`, that means the variable contains those three fields. We could set the `h` field in the bucket like this:

```c
Bucket b;
b.h = 0;
```

The notation is somewhat like how JavaScript accesses object properties.

Memory Pointer

PHP, in comparison to C, is a refined, cultured language. C, by design, has all the power of an assembly language coupled with all the flexibility of an assembly language.

The Unix operating system was initially implemented[9] in assembly language. Because assembly code is tied closely to the bare metal, when moving to new hardware architecture, we can not port assembly-language programs. They need to be rewritten for the new platform. Since it had to be rewritten anyway, authors Dennis Ritchie and Ken Thompson considered rewriting it in a language called "B," which was Thompson's simplified version of BCPL.

However, B was unable to take advantage of some of the new hardware's features. It didn't have the power of an assembly language. Thus, C was invented to have all the power of an assembly language. The name "C" was chosen simply as the next after B.

As a side note, BCPL was created as a language for writing compilers. The fact that the PHP compiler is written in one of its direct successors, C, is both elegant and exceedingly obscure trivia. BCPL is a dead language, but its influence is still felt with Unix, Linux, iOS, and the existence of all C-like languages.

Straight-up C programming, as opposed to C++, C#, or other more-modern variants, is dangerous. A common outcome is the program ending (crashing) with a "segfault." Arrays, for example, have no bounds checks (by design!), so it's easy to corrupt the program space by storing

7 *typedef:* https://en.wikipedia.org/wiki/Typedef

8 *Zend/zend_types.h:* https://phpa.me/php-src-zend-types

9 *initially implemented:* https://phpa.me/wikip-c-language

an array value outside the array. C programmers need to be careful about memory management, pointer calculations—basically everything that's in the program space.

> *Val Kilmer, Iceman: "You're everyone's problem. That's because every time you go up in the air, you're unsafe. I don't like you because you're dangerous." Tom Cruise, Maverick: "That's right, Iceman. I am dangerous." — Top Gun (1986)*

If you're coming from PHP and beginning to read C, remember it's designed to work closely with the hardware. It's designed to *be* the computer, to *see* the computer. (See what we did there, with B and C?) C is designed to be the operating system (Unix/Linux) and the compiler. C is about bits and bytes and memory locations.

A C variable is just an alias for a memory location. If a `Bucket` structure is 128 bytes, an array of ten Buckets would be 1280 bytes. Walking through the array of buckets would mean skipping forward 128 bytes at a time.

There's no such concept in PHP. In PHP, if we had an array of ten Bucket objects, the array would contain ten pointers to the ten objects.

In contrast, in C, we can ask for a slab of memory. The memory allocator provides the starting address of that slab of memory. Meanwhile, we have a `Bucket` pointer. Just stick the memory slab's location in that bucket pointer:

```
Bucket *b;
b = (Bucket *)malloc(...);
```

To move to the second bucket, still inside that single slab of memory, we increment the pointer by the size of the `Bucket` (128 bytes):

```
Bucket *nextBucket = b;
nextBucket += sizeof(Bucket);
```

We can produce bucket *n* as follows:

```
#define getBucket(b, n) ((Bucket *)((b) + ((n) * sizeof(Bucket))))
```

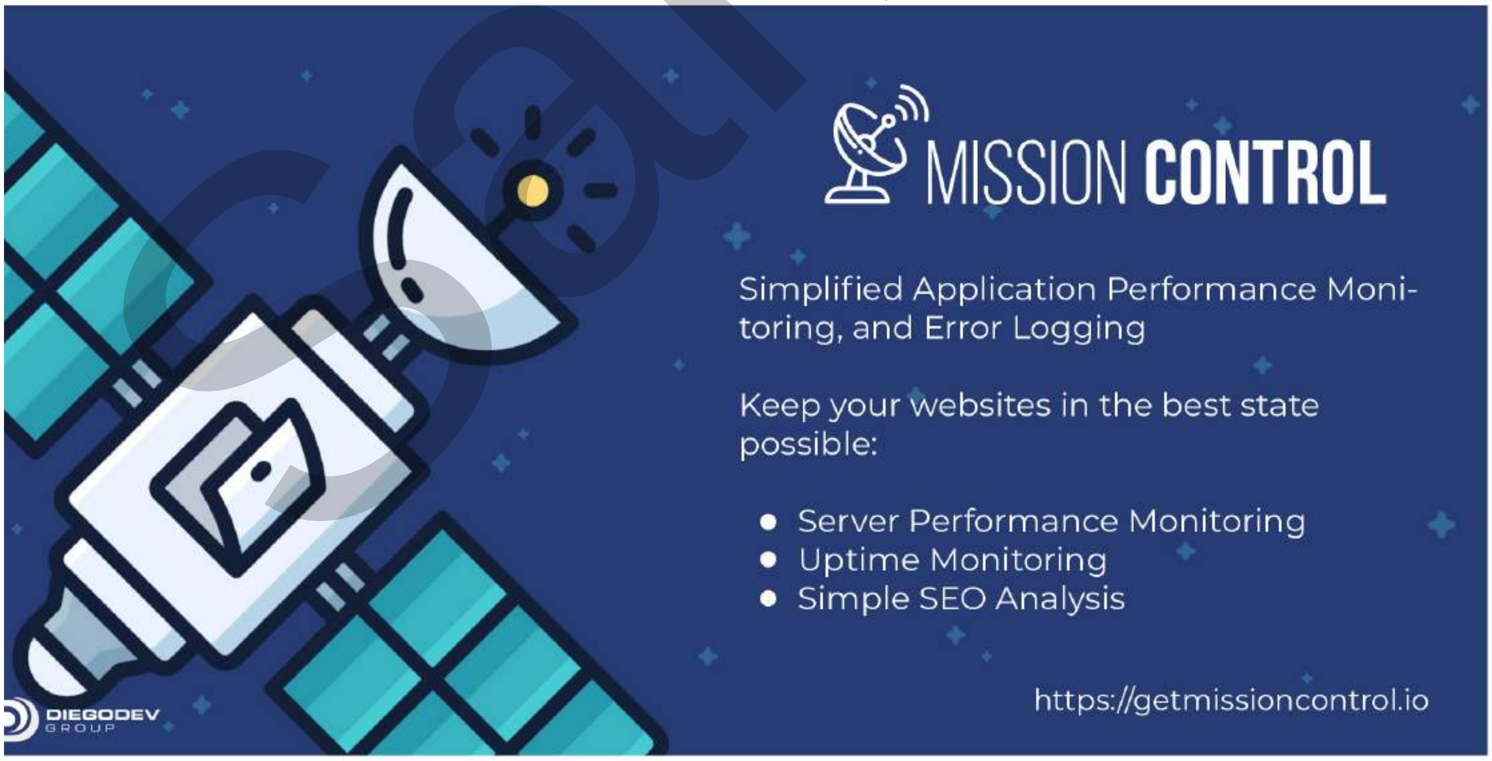
`sizeof(Bucket)` provides us the number of bytes in a bucket. Multiply that by *n* and we have the byte address for bucket *n* (counting from zero). Add that to the base address of the memory slab, and we have the address of bucket *n*.

By defining this as a macro, the result becomes inline code wherever the macro is used. We don't incur the overhead of a function call. The compiler may be able to optimize the code further using the surrounding context. The source code contains many such macros.

In Listing 1 line 5, `arData` is our pointer to that slab of memory. `p` points to the individual bucket inside that slab of memory.

More Macros

The next line of Listing 1 is `IS_CONSISTENT(ht)`. The macro is defined in `Zend/zend_hash.c`. Why would it be defined in the `.c` file and not the header file? That, in effect, means it's private.


MISSION CONTROL
Simplified Application Performance Monitoring, and Error Logging
Keep your websites in the best state possible:
Server Performance Monitoring
Uptime Monitoring
Simple SEO Analysis
https://getmissioncontrol.io
DIEGODEV GROUP

That macro is not available outside the one file. This macro definition is a useful pattern to recognize, so let's take a look, see Listing 3.

A glance at the code indicates we're doing some sanity checking when deleting (destroying) the hash table and releasing its memory. PHP garbage collection can potentially have several phases, but that's a discussion for another time. Right now we're looking for the pattern.

Line 1 is `#if ZEND_DEBUG`, and the last four lines contain the `else` and `endif`. It is a compile-time check. The choice is made when compiling PHP. We can't run PHP with checking on one time and off another. We'd need to recompile PHP with `ZEND_DEBUG` defined or not defined.

The function _zend_is_inconsistent only gets compiled with the debug flag on. Without `ZEND_DEBUG`, the function won't exist at all. With debugging on, the `IS_CONSISTENT(ht)` macro does the checking, passing in the file name and line number for display purposes.

With debug off, the macro compiles to nothing. The macro becomes empty white space. Our line of code gets reduced to the semicolon `;`. Fortunately, an empty statement is legal C code. Multiple empty statements `;;;` are perfectly legal, just like with PHP.

Lines 30-32 show another common idiom: `do { ... } while (0)`. Here the macro is introducing a block of code. A `do` block executes at least once, just like with PHP. The `while(0)` ensures it executes only once.

However, unlike PHP, a variable can be declared inside that `do` block, and its scope is only inside that block. In PHP, the variable is available anywhere inside the function/method, but in C it's limited to the nearest enclosing set of braces. In a macro, the `do` block technique allows the macro to stick multiple lines of C code pretty much anywhere, complete with temporary variables that disappear at the end of the macro.

These code insertion techniques can be maddening to follow, but they provide a consistent way of source code generation. The compiler then takes the ugly result and compiles it into the most efficient code it can.

In a `#define`, the backslash \ at the end of the line says the definition continues on the next line.

Again, the detail of the `IS_CONSISTENT(ht)` macro doesn't matter. What's important is recognizing the pattern so you can understand what's happening without tripping over the details. The PHP virtual machine, C code, gets generated by a PHP script. So, unlike standard PHP source code, much of the Zend Engine (runtime PHP compiler/interpreter) isn't handcrafted like we might expect.

Listing 3. consistent.c

```
 1. #if ZEND_DEBUG
 2.
 3. #define HT_OK                     0x00
 4. #define HT_IS_DESTROYING          0x01
 5. #define HT_DESTROYED              0x02
 6. #define HT_CLEANING               0x03
 7.
 8. static void _zend_is_inconsistent(const HashTable *ht, const char *file, int line)
 9. {
10.     if ((HT_FLAGS(ht) & HASH_FLAG_CONSISTENCY) == HT_OK) {
11.         return;
12.     }
13.     switch (HT_FLAGS(ht) & HASH_FLAG_CONSISTENCY) {
14.         case HT_IS_DESTROYING:
15.             zend_output_debug_string(1, "%s(%d) : ht=%p is being destroyed", file, line, ht);
16.             break;
17.         case HT_DESTROYED:
18.             zend_output_debug_string(1, "%s(%d) : ht=%p is already destroyed", file, line, ht);
19.             break;
20.         case HT_CLEANING:
21.             zend_output_debug_string(1, "%s(%d) : ht=%p is being cleaned", file, line, ht);
22.             break;
23.         default:
24.             zend_output_debug_string(1, "%s(%d) : ht=%p is inconsistent", file, line, ht);
25.             break;
26.     }
27.     zend_bailout();
28. }
29. #define IS_CONSISTENT(a) _zend_is_inconsistent(a, __FILE__, __LINE__);
30. #define SET_INCONSISTENT(n) do { \
31.         HT_FLAGS(ht) = (HT_FLAGS(ht) & ~HASH_FLAG_CONSISTENCY) | (n); \
32.     } while (0)
33. #else
34. #define IS_CONSISTENT(a)
35. #define SET_INCONSISTENT(n)
36. #endif
```

Reference Count

Listing 1 line 8 reads `HT_ASSERT_RC1(ht);`. We know the pattern, so we can make some guesses. The answers, if you're interested, are at `Zend/zend_hash.c` near line 30. "Asserts" tend to be active only in debug mode. Sure enough, without debug mode, this macro compiles down to empty space.

The PHP compiler uses reference counts for various entities. When the reference count drops to zero, perhaps as a result of `unset($variable)` in your PHP code, the item's memory can be released and reused (in the "garbage collection" phase). In PHP, for example, if you have two variables both pointing to the same object instance, that object might have a reference count of two. You'd have to unset both variables before that object can safely disappear.

With `HT_ASSERT_RC1(ht)`, the `HT_` prefix is used all over the hash table feature code. The pointer to the hash table is often a variable named `ht`. So we can pretty well guess this is a sanity check (used in debug mode only) that our hash table has a reference count of one. The hash table is the internal representation of one specific array. The hash table code is written in such a way that the reference count is *always* one. So the sanity check makes sense.

Packed Hash Table

An ordered list of values such as `[1, 1, 2, 3, 5, 8]` is a special case. The array keys are consecutive integers beginning with 0. PHP (written in C) stores the array as a packed hash table. The array value is stored in that `Bucket` we've been discussing. We either store the value directly if it's small enough (`int`, `float`, or `bool`), or we store the pointer to the separate slab of memory containing that value.

Remember, the array is actually the ordered set of key-value pairs:

```
[0 => 1, 1 => 1, 2 => 2, 3 => 3, 4 => 5, 5 => 8]
```

We store key/value *n* in bucket *n*. `$array[0]` is in bucket 0; `array[4]` is in bucket 4, and so on. It's simple and efficient.

If the array is not a "packed" array, it's more difficult to find which bucket contains our array element. We transform the array key through a hashing function to find a possible bucket number. More than one key could hash to the same bucket number. So we check that *possible* bucket to see if it is the right key. If not, we follow its link to the next possible bucket, and so on down the collision chain until we find the matching key or fail completely (the array key does not exist).

Listing 1 line 9 reads `ZEND_ASSERT(!(HT_FLAGS(ht) & HASH_FLAG_PACKED));`. The `ASSERT` is no doubt a sanity check only active in debug mode. The answer is in `Zend/zend_portability.h` near line 100. What's interesting to us is the `FLAG` pattern. It's important to recognize and understand.

However, first, let's note the overall intent of this line. We're asserting that the "packed" flag is not set. This function (line 1 of the listing) aims to find a bucket given the array key. The array key is a string, so we definitely should not be dealing with a packed hash table. The packed hash table is only used for that particular case of continuous numeric keys beginning with zero.

Line 9 includes a Boolean NOT, a bitwise AND, and a hidden bitwise left shift. It's a typical pattern throughout the source code, so let's dive down that rabbit hole.

Flags

The `HT_FLAGS` macro is near line 43 of `Zend/zend_hash.h`.

```
#define HASH_FLAG_CONSISTENCY   ((1<<0) | (1<<1))
#define HASH_FLAG_PACKED        (1<<2)
#define HASH_FLAG_UNINITIALIZED (1<<3)
#define HASH_FLAG_STATIC_KEYS   (1<<4) /* long and interned strings */
#define HASH_FLAG_HAS_EMPTY_IND (1<<5)
#define HASH_FLAG_ALLOW_COW_VIOLATION (1<<6)

#define HT_FLAGS(ht) (ht)->u.flags
```

We know from the function signature (listing 1 line 1) that `ht` is a pointer of type `HashTable`. The next listing to consider is from `Zend/zend_types.h` near line 250.

Well, now. We have quite a rabbit hole. Again, this is typical. As you learn the code base, you'll be chasing down one topic after another. The style of code, though, is consistent—and that's the point. As you come to recognize the patterns, terse as they are, you'll recognize the code's structure in those patterns.

That's the whole point of this month's article: recognizing and understanding those patterns. That's your key to not being intimidated by the terseness and opaqueness of the code. Your comfort level, and your confidence level will increase. Meanwhile, we have this rabbit hole to explore.

Listing 4 Zend/zend_types extract

```
typedef struct _zend_array HashTable;

struct _zend_array {
    zend_refcounted_h gc;
    union {
        struct {
            ZEND_ENDIAN_LOHI_4(
                zend_uchar    flags,
                zend_uchar    _unused,
                zend_uchar    nIteratorsCount,
                zend_uchar    _unused2)
        } v;
        uint32_t flags;
    } u;
    uint32_t          nTableMask;
    Bucket           *arData;
    uint32_t          nNumUsed;
    uint32_t          nNumOfElements;
    uint32_t          nTableSize;
    uint32_t          nInternalPointer;
    zend_long         nNextFreeElement;
    dtor_func_t       pDestructor;
};
```

Line 1 says that `HashTable` is a synonym for `_zend_array`. I'm guessing a lot of other things are also synonyms for `_zend_array`. No doubt that's why the `typedef` and the `struct` are declared separately.

We only care about `u.flags`, but let's take a peek at some other things while we're here. `gc` stands for "garbage collection." It includes reference counts. "Endian" is a macro allowing for the fact that different computers have different endianness[10]. Some computers store stuff in memory front-to-back, and other computers store stuff inside-out and backward. The `ZEND_ENDIAN` macros ensure that things get laid-out in memory consistently across various architectures.

In C, a `union` is what you'd expect. Fields `v` and `flags` are accessed at the same memory location. Remember, in C, a variable (or field) is just a pointer to a memory location. C is perfectly happy to treat those 32 bits as a whole set of flags, or something with an "iterators count" stuffed in the middle. If you want the 8-bit `nIteratorsCount`, access it as `u.v.nIteratorsCount`. If you want the 32 bits of flags, access them as `u.flags`.

Note the rest of the fields (other than `arData`) are plain values, not pointers. The plain value might be a pointer, such as `pDestructor`, but that's the programmer's issue and not ours at the moment.

`arData` will point to that slab of memory we've been describing. For bucket *n*, the memory offset is *n* times `sizeof(Bucket)` in bytes. `arData` plus that memory offset is a pointer to bucket *n*. As we saw last month, that slab of memory also contains hash results at a negative offset. `arData` actually points into the middle of that slab of memory.

`HASH_FLAG_PACKED` is `(1<<2)` which is 4. Our `FLAGS` macros are dealing with bit fields packed into a single variable. We extract the single flag with the bitwise AND (`&`). We test for zero/nonzero as true/false. The NOT operator negates the result.

Why do we use `(1<<2)` instead of just saying 4? This operation is common with bit fields all defined in the same variable. The number is the bit number being used. We can see at a glance that bits 0..6 (counting the rightmost, least-significant, bit as bit 0) are in use. If someone needed to define another bit field, clearly `(1<<7)` is the next available field.

Isn't specifying the left shift less efficient than just specifying the value 1, 2, 4, 8, and so on? No, it isn't. The compiler does the calculation at compile time. It performs the substitution for us. It's better to do it this way for readability's sake.

When we're dealing with, say, 24 bits, it's easy to make a mistake. Which bit is 131072? What if we accidentally wrote it as 131272? Would a reviewer catch that? Using hexadecimal might avoid that problem. 0x40000 has only one bit set, but which bit, counting from zero? We've been talking about bit 16, 17, 18—which bit did we mean? In the context of a sequence of bit-flag definitions, `(1<<17)` makes sense—once we recognize and understand the pattern.

Finally, take a look at listing 1 line 18, `HT_FLAGS(ht) &= ~HASH_FLAG_STATIC_KEYS;`. We know from the header file `HASH_FLAG_STATIC_KEYS` is `(1<<4)`, which is 16, but let's call it bit 4. The `~` operator works just like PHP. Flip all the bits. All bits that are 1 become 0, and all that are 0 become 1. Therefore `~16` means that all bits *except* bit 4 are set to 1.

"Anything" ANDed with 1 is the same "anything." In this case `ht->u.flags` ANDed with `~16` leaves all of the flags intact except bit 4. Anything ANDed with 0 is 0. Therefore what this is doing is clearing bit 4 to zero and leaving everything else as-is. We don't know, at this point, why we're clearing the static keys flag, but we can now recognize the pattern for clearing a bit-flag. When you see this pattern, the code is clearing a flag (or a subfield within a field).

The opposite pattern for setting a bit flag looks like this:

```
HT_FLAGS(ht) |= HASH_FLAG_STATIC_KEYS;
```

Any value ORed with 0 is the same value. Anything ORed with 1 is 1. So we're leaving the other flags as-is and setting the static keys flag to 1.

The above line of code appears to have a function call as the left side of an assignment statement. That wouldn't be legal PHP code, and it's not legal C code either. We're looking at a macro evaluated at compile time. The fact that it's all upper case is our clue that it's a macro. A macro is purely a text substitution.

After substitution, the C compiler will see:

```
(ht)->u.flags |= (1<<4);
```

We can now see the line of perfectly legal C code. We can see what it does. On the other hand, now that we know the pattern, the "macro" form more clearly expresses the intent. The macros aim to show us what the code is trying to accomplish.

Summary and Looking Ahead

We learned about how the C code used macros, structs, and typedefs. PHP (written in C) often stores information (flags) as bit fields. We saw examples of the macros examining, setting, and clearing those bit fields.

We learned more of the C code implementing hash tables as we focused on learning programming patterns common throughout the C code.

Next month we'll begin looking at memory management. We'll first see how stacks and heaps work with runtime C code. We'll then be prepared to learn how that C code handles variables and stack frames for PHP.

10 endianness: https://en.wikipedia.org/wiki/Endianness

Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others. @ewbarnard