1 APR::Bucket - Perl API for manipulating APR Buckets

1.1 Synopsis

```
use APR::Bucket ();
my $ba = $c->bucket_alloc;

$b1 = APR::Bucket->new($ba, "aaa");
$b2 = APR::Bucket::eos_create($ba);
$b3 = APR::Bucket::flush_create($ba);

$b2->is_eos;
$b3->is_flush;

$len = $b1->length;
$len = $b1->read($data);
$type = $b1->type;

$b1->insert_after($b2);
$b1->insert_before($b3);
$b1->remove;
$b1->destroy;

$b2->delete; # remove+destroy

$b4 = APR::Bucket->new($ba, "to be setaside");
$b4->setaside($pool);
```

1.2 Description

`APR::Bucket` allows you to create, manipulate and delete APR buckets.

You will probably find the various insert methods confusing, the tip is to read the function right to left. The following code sample helps to visualize the operations:

```
my $bb = APR::Brigade->new($r->pool, $ba);
my $d1 = APR::Bucket->new($ba, "d1");
my $d2 = APR::Bucket->new($ba, "d2");
my $f1 = APR::Bucket::flush_create($ba);
my $f2 = APR::Bucket::flush_create($ba);
my $e1 = APR::Bucket::eos_create($ba);
                        # head->tail
$bb->insert_head(  $d1); # head->d1->tail
$d1->insert_after( $d2); # head->d1->d2->tail
$d2->insert_before($f1); # head->d1->f1->d2->tail
$d2->insert_after( $f2); # head->d1->f1->d2->f2->tail
$bb->insert_tail(  $e1); # head->d1->f1->d2->f2->e1->tail
```

1.3 API

`APR::Bucket` provides the following functions and/or methods:

1.3.1 delete

Tell the bucket to remove itself from the bucket brigade it belongs to, and destroy itself.

```
$bucket->delete();
```

- **obj: $bucket (APR::Bucket object)**
- **ret: no return value**
- **since: 2.0.00**

If the bucket is not attached to any bucket brigade then this operation just destroys the bucket.

`delete` is a convenience wrapper, internally doing:

```
$b->remove;
$b->destroy;
```

Examples:

Assuming that `$bb` already exists and filled with buckets, replace the existing data buckets with new buckets with upcased data;

```
for (my $b = $bb->first; $b; $b = $bb->next($b)) {
   if ($b->read(my $data)) {
        my $nb = APR::Bucket->new($bb->bucket_alloc, uc $data);
        $b->insert_before($nb);
        $b->delete;
        $b = $nb;
    }
}
```

1.3.2 destroy

Free the resources used by a bucket. If multiple buckets refer to the same resource it is freed when the last one goes away.

```
$bucket->destroy();
```

- **obj: $bucket (APR::Bucket object)**
- **ret: no return value**
- **since: 2.0.00**

A bucket needs to be destroyed if it was removed from a bucket brigade, to avoid memory leak.

If a bucket is linked to a bucket brigade, it needs to be removed from it, before it can be destroyed.

Usually instead of calling:

```
$b->remove;
$b->destroy;
```

it's better to call `delete` which does exactly that.

1.3.3 eos_create

Create an *EndOfStream* bucket.

```
$b = APR::Bucket::eos_create($ba);
```

- **arg1: `$ba` (`APR::BucketAlloc object`)**

  The freelist from which this bucket should be allocated

- **ret: `$b` (`APR::Bucket object`)**

  The new bucket

- **since: 2.0.00**

This bucket type indicates that there is no more data coming from down the filter stack. All filters should flush any buffered data at this point.

Example:

```
use APR::Bucket ();
use Apache2::Connection ();
my $ba = $c->bucket_alloc;
my $eos_b = APR::Bucket::eos_create($ba);
```

1.3.4 flush_create

Create a flush bucket.

```
$b = APR::Bucket::flush_create($ba);
```

- **arg1: `$ba` (`APR::BucketAlloc object`)**

  The freelist from which this bucket should be allocated

- **ret: `$b` (`APR::Bucket object`)**

  The new bucket

- **since: 2.0.00**

This bucket type indicates that filters should flush their data. There is no guarantee that they will flush it, but this is the best we can do.

1.3.5 `insert_after`

Insert a list of buckets after a specified bucket

```
$after_bucket->insert_after($add_bucket);
```

- **obj: `$after_bucket` (`APR::Bucket object`)**

  The bucket to insert after

- **arg1: `$add_bucket` (`APR::Bucket object`)**

  The buckets to insert. It says buckets, since `$add_bucket` may have more buckets attached after itself.

- **ret: no return value**
- **since: 2.0.00**

1.3.6 `insert_before`

Insert a list of buckets before a specified bucket

```
$before_bucket->insert_before($add_bucket);
```

- **obj: `$before_bucket` (`APR::Bucket object`)**

  The bucket to insert before

- **arg1: `$add_bucket` (`APR::Bucket object`)**

  The buckets to insert. It says buckets, since `$add_bucket` may have more buckets attached after itself.

- **ret: no return value**
- **since: 2.0.00**

1.3.7 `is_eos`

Determine if a bucket is an EOS bucket

```
$ret = $bucket->is_eos();
```

- **obj: `$bucket` (`APR::Bucket object`)**
- **ret: `$ret` (boolean)**
- **since: 2.0.00**

1.3.8 is_flush

Determine if a bucket is a FLUSH bucket

```
$ret = $bucket->is_flush();
```

- **obj: $bucket (APR::Bucket object)**
- **ret: $ret (boolean)**
- **since: 2.0.00**

1.3.9 length

Get the length of the data in the bucket.

```
$len = $b->length;
```

- **obj: $b (APR::Bucket object)**
- **ret: $len (integer)**

  If the length is unknown, $len value will be -1.

- **since: 2.0.00**

1.3.10 new

Create a new bucket and initialize it with data:

```
$nb = APR::Bucket->new($ba, $data);
$nb =          $b->new($ba, $data);
$nb = APR::Bucket->new($ba, $data, $offset);
$nb = APR::Bucket->new($ba, $data, $offset, $len);
```

- **obj: $b (APR::Bucket object or class)**
- **arg1: $ba (APR::BucketAlloc object)**
- **arg2: $data (string)**

  The data to initialize with.

  Important: in order to avoid unnecessary data copying the variable is stored in the bucket object. That means that if you modify $data after passing it to new() you will modify the data in the bucket as well. To avoid that pass to new() a copy which you won't modify.

- **opt arg3: $offset (number)**

  Optional offset inside $data. Default: 0.

- **opt arg4: $len (number)**

Optional partial length to read.

If `$offset` is specified, then:

```
length $buffer - $offset;
```

will be used. Otherwise the default is to use:

```
length $buffer;
```

- **ret: `$nb` (`APR::Bucket object`)**

  a newly created bucket object

- **since: 2.0.00**

Examples:

- Create a new bucket using a whole string:

  ```
  use APR::Bucket ();
  my $data = "my data";
  my $b = APR::Bucket->new($ba, $data);
  ```

  now the bucket contains the string *'my data'*.

- Create a new bucket using a sub-string:

  ```
  use APR::Bucket ();
  my $data   = "my data";
  my $offset = 3;
  my $b = APR::Bucket->new($ba, $data, $offset);
  ```

  now the bucket contains the string *'data'*.

- Create a new bucket not using the whole length and starting from an offset:

  ```
  use APR::Bucket ();
  my $data   = "my data";
  my $offset = 3;
  my $len    = 3;
  my $b = APR::Bucket->new($ba, $data, $offset, $len);
  ```

  now the bucket contains the string *'dat'*.

1.3.11 `read`

Read the data from the bucket.

```
$len = $b->read($buffer);
$len = $b->read($buffer, $block);
```

- **obj: `$b` (`APR::Bucket object`)**

  The bucket to read from

- **arg1: `$buffer` (SCALAR)**

  The buffer to fill. All previous data will be lost.

- **opt arg2: `$block` (`APR::Const :read_type constant`)**

  optional reading mode constant.

  By default the read is blocking, via `APR::Const::BLOCK_READ constant`.

- **ret: `$len` (number)**

  How many bytes were actually read

  `$buffer` gets populated with the string that is read. It will contain an empty string if there was nothing to read.

- **since: 2.0.00**
- **excpt: `APR::Error`**

It's important to know that certain bucket types (e.g. file bucket), may perform a split and insert extra buckets following the current one. Therefore never call `$b->remove`, before calling `$b->read`, or you may lose data.

Examples:

Blocking read:

```
my $len = $b->read(my $buffer);
```

Non-blocking read:

```
use APR::Const -compile 'NONBLOCK_READ';
my $len = $b->read(my $buffer, APR::Const::NONBLOCK_READ);
```

1.3.12 `remove`

Tell the bucket to remove itself from the bucket brigade it belongs to.

```
$bucket->remove();
```

- **obj: `$bucket` (`APR::Bucket object`)**
- **ret: no return value**
- **since: 2.0.00**

If the bucket is not attached to any bucket brigade then this operation doesn't do anything.

When the bucket is removed, it's not not destroyed. Usually this is done in order to move the bucket to another bucket brigade. Or to copy the data way before destroying the bucket. If the bucket wasn't moved to another bucket brigade it must be destroyed.

Examples:

Assuming that `$bb1` already exists and filled with buckets, move every odd bucket number to `$bb2` and every even to `$bb3`:

```
my $bb2 = APR::Brigade->new($c->pool, $c->bucket_alloc);
my $bb3 = APR::Brigade->new($c->pool, $c->bucket_alloc);
my $count = 0;
while (my $bucket = $bb->first) {
    $count++;
    $bucket->remove;
    $count % 2
        ? $bb2->insert_tail($bucket)
        : $bb3->insert_tail($bucket);
}
```

1.3.13 `setaside`

Ensure the bucket's data lasts at least as long as the given pool:

```
my $status = $b->setaside($pool);
```

- **obj: `$b` (`APR::Bucket object`)**
- **arg1: `$pool` (`APR::Pool object`)**
- **ret: (`APR::Const status constant`)**

  On success, `APR::Const::SUCCESS` is returned. Otherwise a failure code is returned.

- **excpt: `APR::Error`**

  when your code deals only with mod_perl buckets, you don't have to ask for the return value. If this method is called in the `VOID` context, i.e.:

  ```
  $b->setaside($pool);
  ```

  mod_perl will do the error checking on your behalf, and if the return code is not `APR::Const::SUCCESS`, an `APR::Error exception` will be thrown.

  However if your code doesn't know which bucket types it may need to setaside, you may want to check the return code and deal with any errors. For example one of the possible error codes is `APR::Const::ENOTIMPL`. As of this writing the pipe and socket buckets can't `setaside()`, in which case you may want to look at the `ap_save_brigade()` implementation.

- **since: 2.0.00**

Usually setaside is called by certain output filters, in order to buffer socket writes of smaller buckets into a single write. This method works on all bucket types (not only the mod_perl bucket type), but as explained in the exceptions section, not all bucket types implement this method.

When a mod_perl bucket is setaside, its data is detached from the original perl scalar and copied into a pool bucket. That allows downstream filters to deal with the data originally owned by a Perl interpreter, making it possible for that interpreter to go away and do other things, or be destroyed.

1.3.14 `type`

Get the type of the data in the bucket.

```
$type = $b->type;
```

- **obj: `$b` (`APR::Bucket object`)**
- **ret: `$type` (`APR::BucketType object`)**
- **since: 2.0.00**

You need to invoke `APR::BucketType` methods to access the data.

Example:

Create a flush bucket and read its type's name:

```
use APR::Bucket ();
use APR::BucketType ();
my $b = APR::Bucket::flush_create($ba);
my $type = $b->type;
my $type_name =  $type->name; # FLUSH
```

The type name will be *'FLUSH'* in this example.

1.4 Unsupported API

`APR::Socket` also provides auto-generated Perl interface for a few other methods which aren't tested at the moment and therefore their API is a subject to change. These methods will be finalized later as a need arises. If you want to rely on any of the following methods please contact the the mod_perl development mailing list so we can help each other take the steps necessary to shift the method to an officially supported API.

1.4.1 `data`

```
$data = $b->data;
```

Gives a C pointer to the address of the data in the bucket. I can’t see what use can be done of it in Perl.

- **obj: `$b` (`APR::Bucket object`)**
- **ret: `$data` (C pointer)**
- **since: subject to change**

1.4.2 `start`

```
$start = $b->start;
```

It gives the offset to when a new bucket is created with a non-zero offset value:

```
my $b = APR::Bucket->new($ba, $data, $offset, $len);
```

So if the offset was 3. `$start` will be 3 too.

I fail to see what it can be useful for to the end user (it’s mainly used internally).

- **obj: `$b` (`APR::Bucket object`)**
- **ret: `$start` (offset number)**
- **since: subject to change**

1.5 See Also

mod_perl 2.0 documentation.

1.6 Copyright

1.7 Authors

The mod_perl development team and numerous contributors.

Table of Contents: