# Block Implementation Specification

## History

2008/7/14 - created.
2008/8/21 - revised, C++.
2008/9/24 - add NULL isa field to __block storage.
2008/10/1 - revise block layout to use a static descriptor structure.
2008/10/6 - revise block layout to use an unsigned long int flags.
2008/10/28 - specify use of _Block_object_assign and _Block_object_dispose for all “Object” types in helper functions.
2008/10/30 - revise new layout to have invoke function in same place.
2008/10/30 - add __weak support.
2010/3/16 - rev for stret return, signature field.
2010/4/6 - improved wording.
2013/1/6 - improved wording and converted to rst.

This document describes the Apple ABI implementation specification of Blocks.

The first shipping version of this ABI is found in Mac OS X 10.6, and shall be referred to as 10.6.ABI. As of 2010/3/16, the following describes the ABI contract with the runtime and the compiler, and, as necessary, will be referred to as ABI.2010.3.16.

Since the Apple ABI references symbols from other elements of the system, any attempt to use this ABI on systems prior to SnowLeopard is undefined.

## High Level

The ABI of Blocks consist of their layout and the runtime functions required by the compiler. A Block consists of a structure of the following form:

```
struct Block_literal_1 {
    void *isa; // initialized to &_NSConcreteStackBlock or &_NSConcreteGlobalBlock
    int flags;
    int reserved;
    void (*invoke)(void *, ...);
    struct Block_descriptor_1 {
    unsigned long int reserved;         // NULL
        unsigned long int size;         // sizeof(struct Block_literal_1)
        // optional helper functions
        void (*copy_helper)(void *dst, void *src);     // IFF (1<<25)
        void (*dispose_helper)(void *src);             // IFF (1<<25)
        // required ABI.2010.3.16
        const char *signature;                         // IFF (1<<30)
    } *descriptor;
    // imported variables
};
```

The following flags bits are in use thusly for a possible ABI.2010.3.16:

```
enum {
    BLOCK_HAS_COPY_DISPOSE =  (1 << 25),
    BLOCK_HAS_CTOR =          (1 << 26), // helpers have C++ code
    BLOCK_IS_GLOBAL =         (1 << 28),
    BLOCK_HAS_STRET =         (1 << 29), // IFF BLOCK_HAS_SIGNATURE
    BLOCK_HAS_SIGNATURE =     (1 << 30),
};
```

In 10.6.ABI the (1<<29) was usually set and was always ignored by the runtime - it had been a transitional marker that did not get deleted after the transition. This bit is now paired with (1<<30), and represented as the pair (3<<30), for the following combinations of valid bit settings, and their meanings:

```
switch (flags & (3<<29)) {
  case (0<<29):      10.6.ABI, no signature field available
  case (1<<29):      10.6.ABI, no signature field available
  case (2<<29): ABI.2010.3.16, regular calling convention, presence of signature field
  case (3<<29): ABI.2010.3.16, stret calling convention, presence of signature field,
}
```

The signature field is not always populated.

The following discussions are presented as 10.6.ABI otherwise.

Block literals may occur within functions where the structure is created in stack local memory. They may also appear as initialization expressions for Block variables of global or static local variables.

When a Block literal expression is evaluated the stack based structure is initialized as follows:

1. A static descriptor structure is declared and initialized as follows:

    a. The invoke function pointer is set to a function that takes the Block structure as its first argument and the rest of the arguments (if any) to the Block and executes the Block compound statement.

    b. The size field is set to the size of the following Block literal structure.

    c. The copy_helper and dispose_helper function pointers are set to respective helper functions if they are required by the Block literal.

2. A stack (or global) Block literal data structure is created and initialized as follows:

    a. The isa field is set to the address of the external _NSConcreteStackBlock, which is a block of uninitialized memory supplied in libSystem, or _NSConcreteGlobalBlock if this is a static or file level Block literal.

    b. The flags field is set to zero unless there are variables imported into the Block that need helper functions for program level Block_copy() and Block_release() operations, in which case the (1<<25) flags bit is set.

As an example, the Block literal expression:

```
^ { printf("hello world\n"); }
```

would cause the following to be created on a 32-bit system:

```
struct __block_literal_1 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_1 *);
    struct __block_descriptor_1 *descriptor;
};

void __block_invoke_1(struct __block_literal_1 *_block) {
    printf("hello world\n");
}

static struct __block_descriptor_1 {
    unsigned long int reserved;
    unsigned long int Block_size;
} __block_descriptor_1 = { 0, sizeof(struct __block_literal_1), __block_invoke_1 };
```

and where the Block literal itself appears:

```
struct __block_literal_1 _block_literal = {
     &_NSConcreteStackBlock,
     (1<<29), <uninitialized>,
     __block_invoke_1,
     &__block_descriptor_1
};
```

A Block imports other Block references, const copies of other variables, and variables marked __block. In Objective-C, variables may additionally be objects.

When a Block literal expression is used as the initial value of a global or static local variable, it is initialized as follows:

```
struct __block_literal_1 __block_literal_1 = {
      &_NSConcreteGlobalBlock,
      (1<<28)|(1<<29), <uninitialized>,
      __block_invoke_1,
      &__block_descriptor_1
};
```

that is, a different address is provided as the first value and a particular (1<<28) bit is set in the flags field, and otherwise it is the same as for stack based Block literals. This is an optimization that can be used for any Block literal that imports no const or __block storage variables.

## Imported Variables

Variables of auto storage class are imported as const copies. Variables of __block storage class are imported as a pointer to an enclosing data structure. Global variables are simply referenced and not considered as imported.

### Imported const copy variables

Automatic storage variables not marked with __block are imported as const copies.

The simplest example is that of importing a variable of type int:

```
int x = 10;
void (^vv)(void) = ^{ printf("x is %d\n", x); }
x = 11;
vv();
```

which would be compiled to:

```
struct __block_literal_2 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_2 *);
    struct __block_descriptor_2 *descriptor;
    const int x;
};

void __block_invoke_2(struct __block_literal_2 *_block) {
    printf("x is %d\n", _block->x);
}

static struct __block_descriptor_2 {
    unsigned long int reserved;
    unsigned long int Block_size;
} __block_descriptor_2 = { 0, sizeof(struct __block_literal_2) };
```

and:

```
struct __block_literal_2 __block_literal_2 = {
      &_NSConcreteStackBlock,
      (1<<29), <uninitialized>,
      __block_invoke_2,
      &__block_descriptor_2,
      x
 };
```

In summary, scalars, structures, unions, and function pointers are generally imported as const copies with no need for helper functions.

## Imported const copy of Block reference

The first case where copy and dispose helper functions are required is for the case of when a Block itself is imported. In this case both a copy_helper function and a dispose_helper function are needed. The copy_helper function is passed both the existing stack based pointer and the pointer to the new heap version and should call back into the runtime to actually do the copy operation on the imported fields within the Block. The runtime functions are all described in **Runtime Helper Functions**.

A quick example:

```
void (^existingBlock)(void) = ...;
void (^vv)(void) = ^{ existingBlock(); }
vv();

struct __block_literal_3 {
   ...; // existing block
};

struct __block_literal_4 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_4 *);
    struct __block_literal_3 *const existingBlock;
};

void __block_invoke_4(struct __block_literal_2 *_block) {
   __block->existingBlock->invoke(__block->existingBlock);
}

void __block_copy_4(struct __block_literal_4 *dst, struct __block_literal_4 *src) {
     //_Block_copy_assign(&dst->existingBlock, src->existingBlock, 0);
     _Block_object_assign(&dst->existingBlock, src->existingBlock, BLOCK_FIELD_IS_BLOCK);
}

void __block_dispose_4(struct __block_literal_4 *src) {
     // was _Block_destroy
     _Block_object_dispose(src->existingBlock, BLOCK_FIELD_IS_BLOCK);
}

static struct __block_descriptor_4 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_4 *dst, struct __block_literal_4 *src);
    void (*dispose_helper)(struct __block_literal_4 *);
} __block_descriptor_4 = {
    0,
    sizeof(struct __block_literal_4),
    __block_copy_4,
    __block_dispose_4,
};
```

and where said Block is used:

```
struct __block_literal_4 _block_literal = {
      &_NSConcreteStackBlock,
      (1<<25)|(1<<29), <uninitialized>
      __block_invoke_4,
      & __block_descriptor_4
      existingBlock,
};
```

### Importing __attribute__((NSObject)) variables

GCC introduces __attribute__((NSObject)) on structure pointers to mean "this is an object". This is useful because many low level data structures are declared as opaque structure pointers, e.g. CFStringRef, CFArrayRef, etc. When used from C, however, these are still really objects and are the second case where that requires copy and dispose helper functions to be generated. The copy helper functions generated by the compiler should use the _Block_object_assign runtime helper function and in the dispose helper the _Block_object_dispose runtime helper function should be called.

For example, Block foo in the following:

```
struct Opaque *__attribute__((NSObject)) objectPointer = ...;
...
void (^foo)(void) = ^{  CFPrint(objectPointer); };
```

would have the following helper functions generated:

```
void __block_copy_foo(struct __block_literal_5 *dst, struct __block_literal_5 *src) {
     _Block_object_assign(&dst->objectPointer, src-> objectPointer, BLOCK_FIELD_IS_OBJECT);
}

void __block_dispose_foo(struct __block_literal_5 *src) {
     _Block_object_dispose(src->objectPointer, BLOCK_FIELD_IS_OBJECT);
}
```

## Imported __block marked variables

### Layout of __block marked variables

The compiler must embed variables that are marked __block in a specialized structure of the form:

```
struct _block_byref_foo {
    void *isa;
    struct Block_byref *forwarding;
    int flags;   //refcount;
    int size;
    typeof(marked_variable) marked_variable;
};
```

Variables of certain types require helper functions for when Block_copy() and Block_release() are performed upon a referencing Block. At the "C" level only variables that are of type Block or ones that have __attribute__((NSObject)) marked require helper functions. In Objective-C objects require helper functions and in C++ stack based objects require helper functions. Variables that require helper functions use the form:

```
struct _block_byref_foo {
    void *isa;
    struct _block_byref_foo *forwarding;
    int flags;   //refcount;
    int size;
    // helper functions called via Block_copy() and Block_release()
    void (*byref_keep)(void  *dst, void *src);
    void (*byref_dispose)(void *);
    typeof(marked_variable) marked_variable;
};
```

The structure is initialized such that:

a. The forwarding pointer is set to the beginning of its enclosing structure.

b. The size field is initialized to the total size of the enclosing structure.

c. The flags field is set to either 0 if no helper functions are needed or (1<<25) if they are.

d. The helper functions are initialized (if present).
e. The variable itself is set to its initial value.
f. The isa field is set to NULL.

## Access to __block variables from within its lexical scope

In order to “move” the variable to the heap upon a copy_helper operation the compiler must rewrite access to such a variable to be indirect through the structures forwarding pointer. For example:

```
int __block i = 10;
i = 11;
```

would be rewritten to be:

```
struct _block_byref_i {
  void *isa;
  struct _block_byref_i *forwarding;
  int flags;   //refcount;
  int size;
  int captured_i;
} i = { NULL, &i, 0, sizeof(struct _block_byref_i), 10 };

i.forwarding->captured_i = 11;
```

In the case of a Block reference variable being marked __block the helper code generated must use the _Block_object_assign and _Block_object_dispose routines supplied by the runtime to make the copies. For example:

```
__block void (voidBlock)(void) = blockA;
voidBlock = blockB;
```

would translate into:

```
struct _block_byref_voidBlock {
    void *isa;
    struct _block_byref_voidBlock *forwarding;
    int flags;   //refcount;
    int size;
    void (*byref_keep)(struct _block_byref_voidBlock *dst, struct _block_byref_voidBlock *src);
    void (*byref_dispose)(struct _block_byref_voidBlock *);
    void (^captured_voidBlock)(void);
};

void _block_byref_keep_helper(struct _block_byref_voidBlock *dst, struct _block_byref_voidBlock *src) {
    //_Block_copy_assign(&dst->captured_voidBlock, src->captured_voidBlock, 0);
    _Block_object_assign(&dst->captured_voidBlock, src->captured_voidBlock, BLOCK_FIELD_IS_BLOCK | BLOCK_BYREF_CALLER);
}

void _block_byref_dispose_helper(struct _block_byref_voidBlock *param) {
    //_Block_destroy(param->captured_voidBlock, 0);
    _Block_object_dispose(param->captured_voidBlock, BLOCK_FIELD_IS_BLOCK | BLOCK_BYREF_CALLER)}
```

and:

```
struct _block_byref_voidBlock voidBlock = {( .forwarding=&voidBlock, .flags=(1<<25), .size=sizeof(struct _block_byref_voidBlock *),
    .byref_keep=_block_byref_keep_helper, .byref_dispose=_block_byref_dispose_helper,
    .captured_voidBlock=blockA )};

voidBlock.forwarding->captured_voidBlock = blockB;
```

### Importing __block variables into Blocks

A Block that uses a __block variable in its compound statement body must import the variable and emit copy_helper and dispose_helper helper functions that, in turn, call back into the runtime to actually copy or release the byref data block using the functions _Block_object_assign and _Block_object_dispose.

For example:

```
int __block i = 2;
functioncall(^{ i = 10; });
```

would translate to:

```
struct _block_byref_i {
    void *isa;  // set to NULL
    struct _block_byref_voidBlock *forwarding;
    int flags;   //refcount;
    int size;
    void (*byref_keep)(struct _block_byref_i *dst, struct _block_byref_i *src);
    void (*byref_dispose)(struct _block_byref_i *);
    int captured_i;
};


struct __block_literal_5 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_5 *);
    struct __block_descriptor_5 *descriptor;
    struct _block_byref_i *i_holder;
};

void __block_invoke_5(struct __block_literal_5 *_block) {
   _block->forwarding->captured_i = 10;
}

void __block_copy_5(struct __block_literal_5 *dst, struct __block_literal_5 *src) {
     //_Block_byref_assign_copy(&dst->captured_i, src->captured_i);
     _Block_object_assign(&dst->captured_i, src->captured_i, BLOCK_FIELD_IS_BYREF | BLOCK_BYREF_CALLER);
}

void __block_dispose_5(struct __block_literal_5 *src) {
     //_Block_byref_release(src->captured_i);
     _Block_object_dispose(src->captured_i, BLOCK_FIELD_IS_BYREF | BLOCK_BYREF_CALLER);
}

static struct __block_descriptor_5 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_5 *dst, struct __block_literal_5 *src);
    void (*dispose_helper)(struct __block_literal_5 *);
} __block_descriptor_5 = { 0, sizeof(struct __block_literal_5) __block_copy_5, __block_dispose_5 };
```

and:

```
struct _block_byref_i i = {( .isa=NULL, .forwarding=&i, .flags=0, .size=sizeof(struct _block_byref_i), .captured_i=2 )};
struct __block_literal_5 _block_literal = {
      &_NSConcreteStackBlock,
      (1<<25)|(1<<29), <uninitialized>,
      __block_invoke_5,
      &__block_descriptor_5,
      &i,
};
```

### Importing __attribute__((NSObject)) __block variables

A __block variable that is also marked __attribute__((NSObject)) should have byref_keep and byref_dispose helper functions that use _Block_object_assign and _Block_object_dispose.

### __block escapes

Because Blocks referencing __block variables may have Block_copy() performed upon them the underlying storage for the variables may move to the heap. In Objective-C Garbage Collection Only compilation environments the heap used is the garbage collected one and no further action is required. Otherwise the compiler must issue a call to potentially release any heap storage for __block variables at all escapes or terminations of their scope. The call should be:

```
_Block_object_dispose(&_block_byref_foo, BLOCK_FIELD_IS_BYREF);
```

### Nesting

Blocks may contain Block literal expressions. Any variables used within inner blocks are imported into all enclosing Block scopes even if the variables are not used. This includes const imports as well as __block variables.

## Objective C Extensions to Blocks

### Importing Objects

Objects should be treated as __attribute__((NSObject)) variables; all copy_helper, dispose_helper, byref_keep, and byref_dispose helper functions should use _Block_object_assign and _Block_object_dispose. There should be no code generated that uses *-retain or *-release methods.

### Blocks as Objects

The compiler will treat Blocks as objects when synthesizing property setters and getters, will characterize them as objects when generating garbage collection strong and weak layout information in the same manner as objects, and will issue strong and weak write-barrier assignments in the same manner as objects.

### __weak __block Support

Objective-C (and Objective-C++) support the __weak attribute on __block variables. Under normal circumstances the compiler uses the Objective-C runtime helper support functions objc_assign_weak and objc_read_weak. Both should continue to be used for all reads and writes of __weak __block variables:

```
objc_read_weak(&block->byref_i->forwarding->i)
```

The __weak variable is stored in a _block_byref_foo structure and the Block has copy and dispose helpers for this structure that call:

```
_Block_object_assign(&dest->_block_byref_i, src-> _block_byref_i, BLOCK_FIELD_IS_WEAK | BLOCK_FIELD_IS_BYREF);
```

and:

```
_Block_object_dispose(src->_block_byref_i, BLOCK_FIELD_IS_WEAK | BLOCK_FIELD_IS_BYREF);
```

In turn, the block_byref copy support helpers distinguish between whether the __block variable is a Block or not and should either call:

```
_Block_object_assign(&dest->_block_byref_i, src->_block_byref_i, BLOCK_FIELD_IS_WEAK | BLOCK_FIELD_IS_OBJECT | BLOCK_BYREF_CALLER);
```

for something declared as an object or:

```
_Block_object_assign(&dest->_block_byref_i, src->_block_byref_i, BLOCK_FIELD_IS_WEAK | BLOCK_FIELD_IS_BLOCK | BLOCK_BYREF_CALLER);
```

for something declared as a Block.

A full example follows:

```
__block __weak id obj = <initialization expression>;
functioncall(^{ [obj somemessage]; });
```

would translate to:

```
struct _block_byref_obj {
    void *isa;  // uninitialized
    struct _block_byref_obj *forwarding;
    int flags;   //refcount;
    int size;
    void (*byref_keep)(struct _block_byref_i *dst, struct _block_byref_i *src);
    void (*byref_dispose)(struct _block_byref_i *);
    id captured_obj;
};

void _block_byref_obj_keep(struct _block_byref_voidBlock *dst, struct _block_byref_voidBlock *src) {
    //_Block_copy_assign(&dst->captured_obj, src->captured_obj, 0);
    _Block_object_assign(&dst->captured_obj, src->captured_obj, BLOCK_FIELD_IS_OBJECT | BLOCK_FIELD_IS_WEAK | BLOCK_BYREF_CALLER);
}

void _block_byref_obj_dispose(struct _block_byref_voidBlock *param) {
    //_Block_destroy(param->captured_obj, 0);
    _Block_object_dispose(param->captured_obj, BLOCK_FIELD_IS_OBJECT | BLOCK_FIELD_IS_WEAK | BLOCK_BYREF_CALLER);
};
```

for the block byref part and:

```
struct __block_literal_5 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_5 *);
    struct __block_descriptor_5 *descriptor;
    struct _block_byref_obj *byref_obj;
};

void __block_invoke_5(struct __block_literal_5 *_block) {
   [objc_read_weak(&_block->byref_obj->forwarding->captured_obj) somemessage];
}

void __block_copy_5(struct __block_literal_5 *dst, struct __block_literal_5 *src) {
     //_Block_byref_assign_copy(&dst->byref_obj, src->byref_obj);
     _Block_object_assign(&dst->byref_obj, src->byref_obj, BLOCK_FIELD_IS_BYREF | BLOCK_FIELD_IS_WEAK);
}

void __block_dispose_5(struct __block_literal_5 *src) {
     //_Block_byref_release(src->byref_obj);
     _Block_object_dispose(src->byref_obj, BLOCK_FIELD_IS_BYREF | BLOCK_FIELD_IS_WEAK);
}

static struct __block_descriptor_5 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_5 *dst, struct __block_literal_5 *src);
    void (*dispose_helper)(struct __block_literal_5 *);
} __block_descriptor_5 = { 0, sizeof(struct __block_literal_5), __block_copy_5, __block_dispose_5 };
```

and within the compound statement:

```
truct _block_byref_obj obj = {( .forwarding=&obj, .flags=(1<<25), .size=sizeof(struct _block_byref_obj),
                 .byref_keep=_block_byref_obj_keep, .byref_dispose=_block_byref_obj_dispose,
                 .captured_obj = <initialization expression> )};

truct __block_literal_5 _block_literal = {
     &_NSConcreteStackBlock,
     (1<<25)|(1<<29), <uninitialized>,
     __block_invoke_5,
     &__block_descriptor_5,
     &obj,        // a reference to the on-stack structure containing "captured_obj"
};


functioncall(_block_literal->invoke(&_block_literal));
```

## C++ Support

Within a block stack based C++ objects are copied into const copies using the copy constructor. It is an error if a stack based C++ object is used within a block if it does not have a copy constructor. In addition both copy and destroy helper routines must be synthesized for the block to support the Block_copy() operation, and the flags work marked with the (1<<26) bit in addition to the (1<<25) bit. The copy helper should call the constructor using appropriate offsets of the variable within the supplied stack based block source and heap based destination for all const constructed copies, and similarly should call the destructor in the destroy routine.

As an example, suppose a C++ class FOO existed with a copy constructor. Within a code block a stack version of a FOO object is declared and used within a Block literal expression:

```
{
    FOO foo;
    void (^block)(void) = ^{ printf("%d\n", foo.value()); };
}
```

The compiler would synthesize:

```
struct __block_literal_10 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_10 *);
    struct __block_descriptor_10 *descriptor;
    const FOO foo;
};

void __block_invoke_10(struct __block_literal_10 *_block) {
   printf("%d\n", _block->foo.value());
}

void __block_literal_10(struct __block_literal_10 *dst, struct __block_literal_10 *src) {
     FOO_ctor(&dst->foo, &src->foo);
}

void __block_dispose_10(struct __block_literal_10 *src) {
     FOO_dtor(&src->foo);
}

static struct __block_descriptor_10 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_10 *dst, struct __block_literal_10 *src);
    void (*dispose_helper)(struct __block_literal_10 *);
} __block_descriptor_10 = { 0, sizeof(struct __block_literal_10), __block_copy_10, __block_dispose_10 };
```

and the code would be:

```
{
  FOO foo;
  comp_ctor(&foo); // default constructor
  struct __block_literal_10 _block_literal = {
    &_NSConcreteStackBlock,
    (1<<25)|(1<<26)|(1<<29), <uninitialized>,
    __block_invoke_10,
    &__block_descriptor_10,
   };
   comp_ctor(&_block_literal->foo, &foo);  // const copy into stack version
   struct __block_literal_10 &block = &_block_literal;  // assign literal to block variable
   block->invoke(block);    // invoke block
   comp_dtor(&_block_literal->foo); // destroy stack version of const block copy
   comp_dtor(&foo); // destroy original version
}
```

C++ objects stored in __block storage start out on the stack in a block_byref data structure as do other variables. Such objects (if not const objects) must support a regular copy constructor. The block_byref data structure will have copy and destroy helper routines synthesized by the compiler. The copy helper will have code created to perform the copy constructor based on the initial stack block_byref data structure, and will also set the (1<<26) bit in addition to the (1<<25) bit. The destroy helper will have code to do the destructor on the object stored within the supplied block_byref heap data structure. For example,

```
__block FOO blockStorageFoo;
```

requires the normal constructor for the embedded blockStorageFoo object:

```
FOO_ctor(& _block_byref_blockStorageFoo->blockStorageFoo);
```

and at scope termination the destructor:

```
FOO_dtor(& _block_byref_blockStorageFoo->blockStorageFoo);
```

Note that the forwarding indirection is *NOT* used.

The compiler would need to generate (if used from a block literal) the following copy/dispose helpers:

```
void _block_byref_obj_keep(struct _block_byref_blockStorageFoo *dst, struct _block_byref_blockStorageFoo *src) {
     FOO_ctor(&dst->blockStorageFoo, &src->blockStorageFoo);
}

void _block_byref_obj_dispose(struct _block_byref_blockStorageFoo *src) {
     FOO_dtor(&src->blockStorageFoo);
}
```

for the appropriately named constructor and destructor for the class/struct FOO.

To support member variable and function access the compiler will synthesize a const pointer to a block version of the this pointer.

## Runtime Helper Functions

The runtime helper functions are described in /usr/local/include/Block_private.h. To summarize their use, a Block requires copy/dispose helpers if it imports any block variables, __block storage variables, __attribute__((NSObject)) variables, or C++ const copied objects with constructor/destructors. The (1<<26) bit is set and functions are generated.

The block copy helper function should, for each of the variables of the type mentioned above, call:

```
_Block_object_assign(&dst->target, src->target, BLOCK_FIELD_<apropos>);
```

in the copy helper and:

```
_Block_object_dispose(->target, BLOCK_FIELD_<apropos>);
```

in the dispose helper where <apropos> is:

```
enum {
    BLOCK_FIELD_IS_OBJECT   =  3,  // id, NSObject, __attribute__((NSObject)), block, ...
    BLOCK_FIELD_IS_BLOCK    =  7,  // a block variable
    BLOCK_FIELD_IS_BYREF    =  8,  // the on stack structure holding the __block variable

    BLOCK_FIELD_IS_WEAK     = 16,  // declared __weak

    BLOCK_BYREF_CALLER      = 128, // called from byref copy/dispose helpers
};
```

and of course the constructors/destructors for const copied C++ objects.

The block_byref data structure similarly requires copy/dispose helpers for block variables, __attribute__((NSObject)) variables, or C++ const copied objects with constructor/destructors, and again the (1<<26) bit is set and functions are generated in the same manner.

Under ObjC we allow __weak as an attribute on __block variables, and this causes the addition of BLOCK_FIELD_IS_WEAK orred onto the BLOCK_FIELD_IS_BYREF flag when copying the block_byref structure in the Block copy helper, and onto the BLOCK_FIELD_<apropos> field within the block_byref copy/dispose helper calls.

The prototypes, and summary, of the helper functions are:

```
/* Certain field types require runtime assistance when being copied to the
   heap.  The following function is used to copy fields of types: blocks,
   pointers to byref structures, and objects (including
   __attribute__((NSObject)) pointers.  BLOCK_FIELD_IS_WEAK is orthogonal to
   the other choices which are mutually exclusive.  Only in a Block copy
   helper will one see BLOCK_FIELD_IS_BYREF.
*/
void _Block_object_assign(void *destAddr, const void *object, const int flags);

/* Similarly a compiler generated dispose helper needs to call back for each
   field of the byref data structure.  (Currently the implementation only
   packs one field into the byref structure but in principle there could be
   more).  The same flags used in the copy helper should be used for each
   call generated to this function:
*/
void _Block_object_dispose(const void *object, const int flags);
```

## Copyright