
10 The Support Kit

Introduction	3
Class Information	5
The Class-Information Macros	5
Safe Casting	6
Participating in the System	7
If the Base Class Participates	7
If the Base Class Doesn't Participate	8
inherited	9
Caveats.	9
Debugging Tools	11
The DEBUG Compiler Variable.	11
The Debug Flag	11
Macros	12
BList	15
Overview	15
Constructor and Destructor	15
Member Functions.	16
Operators	20
Blocker	21
Overview	21
Constructor and Destructor	22
Member Functions.	23
BObject	25
Overview	25
Constructor and Destructor	25
BStopWatch	27
Overview	27
Constructor and Destructor	28

Functions, Constants, and	
Defined Types	.29
Functions and Macros	.29
Constants	.31
Defined Types	.32
Error Codes	.35
General Error Codes	.35
Application Kit Error Codes	.36
Debugger Error Codes	.36
Kernel Kit Error Codes	.36
Media Kit Error Codes	.37

10 The Support Kit

The Support Kit contains classes and utilities that any application can take advantage of—regardless of what kind of application it is or what it does. Among other things, it includes:

- The root BObject class
- The BList class
- A system for getting class information at run time
- Debugging tools including the BStopWatch class
- Common defined types, macros, and error codes

Class Information

Declared in: `<support/ClassInfo.h>`

The class-information system is a set of macros that can supply information at run time about an object's class, such as its name and whether it derives from some other class. There are two parts to the system:

- The macros themselves, and
- The things you need to do to allow classes you design to participate in the system.

The following sections explore these two topics.

Note: Notable by its absence from these discussions is the `BClassInfo` class. This class is the mechanism behind the class-information system—every class that participates in the system is given a `BClassInfo` object that supplies information about the class. An important feature of the system, however, is that you never have to instantiate (or otherwise locate) a `BClassInfo` object to take advantage of the information that it provides. So while you should be aware that the class exists (its declaration is, by necessity, public), you needn't be concerned with it.

The Class-Information Macros

An object of a class that participates in the class-information system (and this includes *almost* all the classes supplied by Be) can supply three kinds of information about itself:

- What the name of its class is,
- Whether it's an instance of a particular class, and
- Whether its class derives from some other class (or perhaps *is* the other class).

These three capabilities are embodied in the following macros,

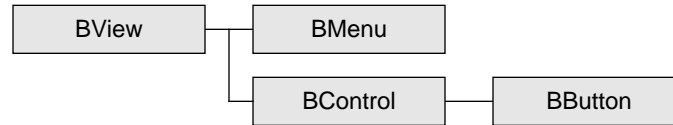
```
const char *class_name(object)
bool is_instance_of(object, class)
bool is_kind_of(object, class)
```

where *object* is a pointer to any type of object and *class* is a class designator—it's *not* a string name (for example, you would use `BView`, not “BView”).

The `class_name()` macro returns the name of the object's class. `is_instance_of()` returns `TRUE` if *object* is an instance of *class*, and `FALSE` otherwise. `is_kind_of()` returns `TRUE` if

object is an instance of a class that inherits from *class* or an instance of *class* itself, and **FALSE** if not.

For example, given this slice of the inheritance hierarchy from the Interface Kit,



and code like this that creates an instance of the BButton class,

```
BButton *anObject = new BButton(...);
```

these three macros would work as follows:

- The `class_name()` macro would return the string “BButton”:

```
const char *s = class_name(anObject);
```

- The `is_instance_of()` macro would return **TRUE** only if the *class* passed to it is BButton. In the following example, it would return **FALSE**, and the message would not be printed. Even though BButton inherits from BView, the object is an instance of the BButton class, not BView:

```
if ( is_instance_of(anObject, BView) )
    printf("The object is an instance of BView.\n");
```

- The `is_kind_of()` macro would return **TRUE** if *class* is BButton or any class that BButton inherits from. In the following example, it would return **TRUE** and the message would be printed. A BButton is a kind of BView:

```
if ( is_kind_of(anObject, BView) )
    printf("The object is a kind of BView.\n");
```

Note that class names are not passed as strings.

Safe Casting

An object whose class participates in the class-information system will permit itself to be cast to that class or to any class that it inherits from. The agent for this kind of “safe casting” is the following macro,

```
class *cast_as(object, class)
```

where *object* is an object pointer and *class* is a class designator.

`cast_as()` returns a pointer to *object* cast as a pointer to an object of *class*, provided that *object* is a kind of *class*—that is, provided that it’s an instance of a class that inherits from

class or is an instance of *class* itself. If not, *object* cannot be safely cast as pointer to *class*, so `cast_as()` returns `NULL`.

This macro is most useful when you have a pointer to a generic object and you want to treat it as a pointer to a more specific class.

Suppose, for example, that you retrieve a `BWindow` from a `BView`:

```
BWindow *window = myView->Window();
```

Furthermore, let's say that you suspect that the object that was returned is really an instance of a `BWindow`-derived class called `MyWindow`. If it is, you want to cast the object to be a `MyWindow` pointer. The `cast_as()` macro accomplishes this in one step:

```
MyWindow *mine;

if ( mine = cast_as(window, MyWindow) )
    /* mine is cast as a MyWindow object. */
else
    /* mine is set to NULL. */
```

Participating in the System

To take part in the class-information system, your classes need to include a pair of lines:

- A *declaration* line that goes in the class declaration (in the header file).
- A *definition* line that goes in the class implementation file.

There are two sets of both of these lines; the set you use depends on the (immediate) base class from which you're deriving your class.

If the Base Class Participates

If your class derives from a base class that participates in the class-information system, use this set:

```
B_DECLARE_CLASS_INFO(base)
B_DEFINE_CLASS_INFO(class, base)
```

where *base* is the name of the base class and *class* is the name of the new class you're defining.

For example, let's say you've created a `PaperView` class that derives from `BView`. `BView` participates in the class-information system (as do all Be classes, except where noted in

the class description), so `PaperView`'s declaration would look like this in the **PaperView.h** header file:

```
#include <interface/BView.h>

class PaperView : public BView
{
    B_DECLARE_CLASS_INFO(BView);
    /* Data and function declarations go here. */
}
```

And the implementation would follow this form in **PaperView.cpp**:

```
#include "PaperView.h"

B_DEFINE_CLASS_INFO(PaperView, BView);

PaperView::PaperView(...) : BView(...)
{
    ...
}
```

If the Base Class Doesn't Participate

If your class doesn't derive from a participating class, use these lines instead of those shown above:

```
B_DECLARE_ROOT_CLASS_INFO()
B_DEFINE_ROOT_CLASS_INFO(class)
```

The disposition of these lines is the same as before. In the example below, a class `TopDog` inherits from nobody. Notice that, in this case, the **TopDog.h** header must explicitly include **ClassInfo.h**:

```
#include <support/ClassInfo.h>

class TopDog
{
    B_DECLARE_ROOT_CLASS_INFO();
    /* Data and function declarations go here. */
}
```

The implementation file need only include the class header:

```
#include "TopDog.h"

B_DEFINE_ROOT_CLASS_INFO(TopDog);

TopDog::TopDog(...)
{
    ...
}
```

The only Be classes that *don't* participate in the class-information system are those that define global objects shared across address spaces (like the Application Kit's BRoster and BClipboard) and simple data containers that have been sheared of all extra baggage (like the Interface Kit's BRect and BPoint). If your class derives from any other Be class, you should use the first set of class-information lines.

inherited

Every class that participates in the class-information system, and has a base class that also participates, gets a private keyword—**inherited**—typed to its base class. This term comes with the **B_DECLARE_CLASS_INFO** declaration; it's designed to simplify references to base-class functions, especially when overriding a virtual function in order to add something to it. For example:

```
bool MyClass::DoSomething(long foolish)
{
    ...
    return inherited::DoSomething(foolish);
}
```

When every derived class has access to this same term, all classes can refer to inherited functions in the same way.

Caveats

The class-information system is accurate only for classes that explicitly participate. If, for example, the PaperView class didn't include the declaration and definition lines as shown above, the class-information macros would regard all PaperView objects as BView objects.

The macros fail entirely when applied to a class that not only doesn't participate in the system, but also doesn't derive (directly or indirectly) from any classes that do. The failure is reported as a compile-time error. Therefore, it's strongly suggested that your class derive from BObject if no other Be class is a fit base.

Finally, the class-information system doesn't accommodate multiple inheritance.

Debugging Tools

Declared in: `<support/Debug.h>`

The Support Kit provides a set of macros that help you debug your application. These tools let you print information to standard output or to the serial port, and conditionally enter the debugger.

To enable the Support Kit's debugging tools you have to do two things:

- Compile your code with the `DEBUG` compiler variable defined
- Turn on the debug flag in your code through the `SET_DEBUG_ENABLED()` macro

These two acts represent, respectively, a compile time and a run time decision about the effectiveness of the debugging tools. The compile time decision overrides the run time decision: Turning on the debug flag (`SET_DEBUG_ENABLED(TRUE)`) has no affect if the `DEBUG` variable isn't defined.

The DEBUG Compiler Variable

Defining the `DEBUG` compiler variable can be done by adding the following line to your **makefile**:

```
USER_DEBUG_C_FLAGS := -DDEBUG
```

(The MetroWerks IDE will undoubtedly supply a means for setting the `DEBUG` variable through its interface. Check your local papers.)

When you're through debugging your application, simply remove the `DEBUG` definition and all of the debugging macros will be compiled away—you don't have to actually go into the code and remove the macros or comment them out.

The Debug Flag

The `SET_DEBUG_ENABLED()` macro turns on (or off) the debug flag. When you call a debugging tool, the state of the debugger flag is checked; if it's turned on, the tool does what it's designed to do (and, in with some tools, you could end up in the debugger). If the flag is off, the tool is ignored.

Note: The debug flag is on by default. If you want to (initially, at least) turn it off, you should call `SET_DEBUG_ENABLED(FALSE)` as one of your first acts in `main()`.

The run time aspect of the debug flag is particularly convenient if your application is large and you want to concentrate on certain sections of the code. Note, however, that the scope of the flag is application-wide. You can't, for example, disable the debugging tools across an object's member functions by simply calling `SET_DEBUG_ENABLED(FALSE)` in the object's constructor.

Macros

DEBUGGER(), ASSERT()

`DEBUGGER(var_args)`

`ASSERT(condition)`

These macros cause your program to enter the debugger: `DEBUGGER()` always enters the debugger, `ASSERT()` enters if *condition* (which can be any normal C or C++ expression) evaluates to `FALSE`.

`DEBUGGER()` takes a `printf()`-style variable-length argument that must be wrapped inside a second set of parentheses; for example:

```
DEBUGGER(("What time is it? %f\n", system_time()));
```

The argument is evaluated and printed in the debugger's shell.

If `ASSERT()` enters the debugger, the following message is printed:

```
Assert failed: File: filename, Line: number. condition
```

Note that `ASSERT()`'s argument needn't be wrapped in a second set of parentheses.

HEAP_STATS()

`HEAP_STATS()`

Prints, to standard out, a message that gives statistics about your application's memory heap. The message appears in this format:

```
Heap Size: size bytes
Used blocks: count (size bytes)
Free blocks: count (size bytes)
```

PRINT(), SERIAL_PRINT()

```
PRINT(var_args)
SERIAL_PRINT(var_args)
```

These macros print the message given by *var_args*. The argument takes the variable argument form of a `printf()` call and must be wrapped inside a second set of parenthesis; for example:

```
PRINT(("The time is %f\n", system_time()));
```

`PRINT()` sends the message to standard out; `SERIAL_PRINT()` to serial port 4 (the bottom-most serial port on the back of the computer).

PRINT_OBJECT()

```
PRINT_OBJECT(object)
```

Prints information about the argument *object* (which must be a pointer to a C++ object) by calling the object's `PrintToStream()` function. The macro doesn't check to make sure that object actually implements the function, so you should use this macro with care.

Object information is always printed to standard out (there isn't a serial port version of the call).

SET_DEBUG_ENABLED(), IS_DEBUG_ENABLED()

```
SET_DEBUG_ENABLED(flag)
IS_DEBUG_ENABLED(void)
```

The `SET_DEBUG_ENABLED()` macro sets the state of the run time debug flag: A `TRUE` argument turns it on, `FALSE` turns it off. The utility of the other debugging macros depends on the state of the debug flag: When the flag is on, the macros work; when it's off, they're ignored. The debug flag is set to `TRUE` by default.

The debug flag is only meaningful if your code was compiled with the `DEBUG` compiler variable defined. Without the variable definition, the flag is always `FALSE`.

`IS_DEBUG_ENABLED()` returns the current state of the debug flag.

TRACE(), SERIAL_TRACE()

```
TRACE(void)
SERIAL_TRACE(void)
```

These macros print the name of the source code file that contains the currently executing code (in other words, the file that contains the `TRACE()` call itself), the line number of the code, and the `thread_id` of the calling thread. The information is printed in this form:

```
File: filename, Line: number, Thread: id
```

TRACE() sends the message to standard out; **SERIAL_TRACE()** to serial port 4 (the bottom-most serial port on the back of the computer).

BList

Derived from: public BObject
Declared in: <support/List.h>

Overview

A BList object is a compact, ordered list of data pointers. BList objects can contain pointers to any type of data, including—and especially—objects.

Items in a BList are identified by their ordinal position, or index, starting with index 0. Indices are neither arbitrary nor permanent. If, for example, you insert an item into the middle of a list, the indices of the items at the tail of the list are incremented (by one). Similarly, removing an item decrements the indices of the following items.

A BList stores its items as type `void *`, so it's necessary to cast an item to the correct type when you retrieve it. For example, items retrieved from a list of BBitmap objects must be cast as BBitmap pointers:

```
BBitmap *theImage = (BBitmap *)myList->ItemAt(anIndex);
```

Note: There's nothing to prevent you from adding a **NULL** pointer to a BList. However, functions that retrieve items from the list (such as `ItemAt()`) return **NULL** when the requested item can't be found. Thus, you can't distinguish between a valid **NULL** item and an invalid attempt to access an item that isn't there.

Constructor and Destructor

BList()

```
BList(long blockSize = 20)  
BList(const BList& anotherList)
```

Initializes the BList by allocating enough memory to hold *blockSize* items. As the list grows and shrinks, additional memory is allocated and freed in blocks of the same size.

The copy constructor creates an independent list of data pointers, but it doesn't copy the pointed-to data. For example:

```
BList *newList = new BList(oldList);
```

Here, the contents of *oldList* and *newList*—the actual data pointers—are separate and independent. Adding, removing, or reordering items in *oldList* won't affect the number or order of items in *newList*. But if you modify the data that an item in *oldList* points to, the modification will be seen through the analogous item in *newList*.

The block size of a BList that's created through the copy constructor is the same as that of the original BList.

~BList()

virtual ~BList(void)

Frees the list of data pointers, but doesn't free the data that they point to. To destroy the data, you need to free each item in an appropriate manner. For example, objects that were allocated with the **new** operator should be freed with **delete**:

```
void *anItem;
for ( long i = 0; anItem = myList->ItemAt(i); i++ )
    delete anItem;
delete myList;
```

See also: **MakeEmpty()**

Member Functions

AddItem()

bool AddItem(void *item, long index)
inline bool AddItem(void *item)

Adds an item to the BList at *index*—or, if no index is supplied, at the end of the list. If necessary, additional memory is allocated to accommodate the new item.

Adding an item never removes an item already in the list. If the item is added at an index that's already occupied, items currently in the list are bumped down one slot to make room.

If *index* is out-of-range (greater than the current item count, or less than zero), the function fails and returns **FALSE**. Otherwise it returns **TRUE**.

AddList()

```
bool AddList(BList *list, long index)
bool AddList(BList *list)
```

Adds the contents of another BList to this BList. The items from the other BList are inserted at *index*—or, if no index is given, they’re appended to the end of the list. If the index is out-of-range, the function fails and returns **FALSE**. If successful, it returns **TRUE**.

See also: **AddItem()**

CountItems()

```
inline long CountItems(void) const
```

Returns the number of items currently in the list.

DoForEach()

```
void DoForEach(bool (*func)(void *))
void DoForEach(bool (*func)(void *, void *), void *arg2)
```

Calls the *func* function once for each item in the BList. Items are visited in order, beginning with the first one in the list (index 0) and ending with the last. If a call to *func* returns **TRUE**, the iteration is stopped, even if some items have not yet been visited.

func must be a function that takes one or two arguments. The first argument is the currently-considered item from the list; the second argument, if *func* requires one, is passed to **DoForEach()** as *arg2*.

FirstItem()

```
inline void *FirstItem(void) const
```

Returns the first item in the list, or **NULL** if the list is empty. This function doesn’t remove the item from the list.

See also: **LastItem()**, **ItemAt()**

HasItem()

```
inline bool HasItem(void *item) const
```

Returns **TRUE** if *item* is in the list, and **FALSE** if not.

IndexOf()

```
long IndexOf(void *item) const
```

Returns the ordinal position of *item* in the list, or **B_ERROR** if *item* isn't in the list. If the item is in the list more than once, the index returned will be the position of its first occurrence.

IsEmpty()

```
inline bool IsEmpty(void) const
```

Returns **TRUE** if the list is empty (if it contains no items), and **FALSE** otherwise.

See also: **MakeEmpty()**

ItemAt()

```
inline void *ItemAt(long index) const
```

Returns the item at *index*, or **NULL** if the index is out-of-range. This function doesn't remove the item from the list.

See also: **Items()**, **FirstItem()**, **LastItem()**

Items()

```
inline void *Items(void) const
```

Returns a pointer to the BList's list. You can index directly into the list if you're certain that the index is in-range:

```
myType item = (myType)Items()[index];
```

Although the practice is discouraged, you can also step through the list of items by incrementing the list pointer that's returned by **Items()**. Be aware that the list isn't null-terminated—you have to detect the end of the list by some other means. The simplest method is to count items:

```
void *ptr = myList->Items();

for ( long i = myList->ItemCount(); i > 0; i-- )
{
    . . .
    *ptr++;
}
```

You should *never* use the list pointer to change the number of items in the list.

See also: **DoForEach()**, **SortItems()**

LastItem()

```
inline void *LastItem(void) const
```

Returns the last item in the list without removing it. If the list is empty, this function returns **NULL**.

See also: **RemoveLastItem()**, **FirstItem()**

MakeEmpty()

```
void MakeEmpty(void)
```

Empties the BList of all its items, without freeing the data that they point to.

See also: **IsEmpty()**, **RemoveItem()**

RemoveItem()

```
bool RemoveItem(void *item)
void *RemoveItem(long index)
```

Removes an item from the list. If passed an *item*, the function looks for the item in the list, removes it, and returns **TRUE**. If it can't find the item, it returns **FALSE**. If the item is in the list more than once, this function removes only its first occurrence.

If passed an *index*, the function removes the item at that index and returns it. If there's no item at the index, it returns **NULL**.

The list is compacted after an item is removed. Because of this, you mustn't try to empty a list (or a range within a list) by removing items at monotonically increasing indices. You should either start with the highest index and move towards the head of the list, or remove at the same index (the lowest in the range) some number of times. As an example of the latter, the following code removes the first five items in the list:

```
for ( long i = 0; i <= 4; i++ )
    myList->RemoveItem(0);
```

See also: **MakeEmpty()**

SortItems()

```
void *SortItems(int (*compareFunc)(const void *, const void *))
```

Rearranges the items in the list. The items are sorted using the *compareFunc* comparison function passed as an argument. This function should take two items as arguments. It should return a negative number if the first item should be ordered before the second, a

positive number if the second should be ordered before the first, and 0 if the two items should be ordered equivalently.

See also: `Items()`

Operators

`=` (assignment)

```
BList& operator =(const BList&)
```

Copies the contents of one BList object into another:

```
BList newList = oldList;
```

After the assignment, each object has its own independent copy of list data; destroying one of the objects won't affect the other.

Only the items in the list are copied, not the data they point to.

BLocker

Derived from: public BObject
Declared in: <support/Locker.h>

Overview

The BLocker class provides a locking mechanism that protects a section of code. The code that you want to protect should be placed between BLocker's `Lock()` and `Unlock()` calls:

```
BLocker *aLock = new BLocker();  
  
...  
aLock->Lock();  
/* Protected code goes here. */  
aLock->Unlock();
```

This disposition of calls guarantees that only one thread at a time will pass through the lock. After a thread has locked the BLocker object, subsequent attempts to lock by other threads are blocked until the first thread calls `Unlock()`.

BLocker keeps track of its lock's "owner"—the thread that's currently between `Lock()` and `Unlock()` calls. It lets the lock owner make nested calls to `Lock()` without blocking. Because of this, you can wrap a BLocker's lock around a series of functions that might, themselves, lock the same BLocker object.

For example, let's say you have a class called `BadDog` that's declared thus:

```
class MyObject : public BObject  
{  
public:  
    void DoThis();  
    void DoThat();  
    void DoThisAndThat();  
  
private:  
    BLocker lock;  
};
```

And let's implement the member functions as shown below:

```
void BadDog::DoThis()  
{  
    lock.Lock();
```

```

        /* Do this here. */
        lock.Unlock();
    }

    void BadDog::DoThat()
    {
        lock.Lock();
        /* Do that here. */
        lock.Unlock();
    }

    void BadDog::DoThisAndThat()
    {
        lock.Lock();
        DoThis();
        DoThat();
        lock.Unlock();
    }

```

Notice that **DoThisAndThat()** wraps the lock around its calls to **DoThis()** and **DoThat()**, both of which contain locks as well. A thread that gets past the **Lock()** call in **DoThisAndThat()** will be consider the lock's owner, and so it won't block when it calls the nested **Lock()** calls that it runs into in **DoThis()** and **DoThat()**.

Keep in mind that nested **Lock()** calls must be balanced by equally-nested **Unlock()** calls.

Constructor and Destructor

BLocker()

BLocker(void)

BLocker(const char **name*)

Sets up the object. The optional name is purely for diagnostics and debugging.

~BLocker()

virtual **~BLocker**(void)

Deletes the object. If there are any threads blocked waiting to lock the object, they're immediately unblocked.

Member Functions

CheckLock()

inline bool **CheckLock**(void) const

Checks to see whether the calling thread is the thread that currently owns the lock. If it is, **CheckLock()** returns **TRUE**. If it's not, **CheckLock()** returns **FALSE**

Lock(), Unlock()

void **Lock**(void)

void **Unlock**(void)

These functions lock and unlock the BLocker.

Lock() attempts to lock the BLocker and set the lock's owner to the calling thread. The function doesn't return until it has succeeded. While the BLocker is locked, non-owner calls to **Lock()** will block. The owner, on the other hand, can make additional, nested calls to **Lock()** without blocking.

Unlock() releases one level of nested locks and returns immediately. When the BLocker is completely unlocked—when all nested **Lock()** calls have been matched by calls to **Unlock()**—the lock's owner is “unset”, allowing some other thread to lock the BLocker. If there are threads blocked in **Lock()** calls when the lock is released, the thread that's been waiting the longest acquires the lock.

Although you're not prevented from doing so, it's not good form to call **Unlock()** from a thread that doesn't own the lock. For debugging purposes, you can call **CheckLock()** before calling **Unlock()** to make sure this doesn't happen in your code.

See also: **LockOwner()**

LockOwner()

inline thread_id **LockOwner**(void) const

Returns the thread that currently owns the lock, or **-1** if the BLocker isn't currently locked.

See also: **Lock()**

BObject

Derived from: *none*
Declared in: `<support/Object.h>`

Overview

BObject is the root class of the inheritance hierarchy. All Be classes (with just a handful of significant exceptions) are derived from it.

The primary reason for a single, shared base class is to provide common functionality to all objects. Currently, the BObject class is empty (except for its constructor and destructor), so there's no significant functionality to report. Subsequent releases will probably introduce new functions to the class; in anticipation of this, it's suggested that the classes you design derive from BObject (if no other Be class is a fit base).

In addition, when all objects are derived from BObject, the class can provide a generic type classification (`BObject *`) that simply means "an object." This can be a useful substitute for type `void *`.

As a further incentive, simply by deriving from BObject your classes will be recognized by the class-information mechanism. (They may not be recognized correctly, but at least a class-information query on your objects won't stop the compiler in its tracks. See "Class Information" on page 5 for details.)

Constructor and Destructor

BObject()

`BObject(void)`

Does nothing. Because the BObject class has no data members to initialize, the BObject constructor is empty.

~BObject()

`virtual ~BObject(void)`

Does nothing. Because the BObject class doesn't declare any data members, the BObject destructor has nothing to free.

BStopWatch

Derived from: public BObject
Declared in: <support/StopWatch.h>

Overview

The BStopWatch class is a debugging tool that you can use to time the execution of portions of your code. The class has no member functions or (public) member data. When a BStopWatch object is constructed, it starts its internal timer. When it's deleted it stops the timer and prints the elapsed time to standard out in this format:

StopWatch "*name*": *f* usecs.

Where *name* is the name that you gave to the object when you constructed it, and *f* is the elapsed time in microseconds reckoned to one decimal place.

For example ...

```
#include <StopWatch.h>
...
BStopWatch *myWatch = new BStopWatch("Timer 0");
/* The code you want to time goes here. */
delete myWatch;
...
```

... would produce, on standard out, a message that goes something like this:

```
StopWatch "Timer 0": 492416.3 usecs.
```

This would indicate that the timed code took about half a second to execute—remember, you're looking at microseconds.

BStopWatch objects are handy little critters. They're particularly useful if you want to get a general idea of where your cycles are going. But you shouldn't rely on them for painfully accurate measurements.

Important: Unlike the other debugging tools defined by the Support Kit, there's no run-time toggle to control a BStopWatch. Make sure you remove your BStopWatch objects after you're done debugging your code.

Constructor and Destructor

BStopWatch()

BStopWatch(const char **name*)

Creates a BStopWatch object, names it name, and starts its internal timer.

~BStopWatch()

virtual **~BStopWatch**(void)

Stops the object's timer, spits out a timing message to standard out, and then destroys the object and everything it believes in.

Functions, Constants, and Defined Types

This section lists the Support Kit’s general-purpose functions (including function-like macros), constants, and defined types. These elements are used throughout the Be application-programming interface.

Not listed here are constants that are used as error codes. These are listed in “Error Codes” on page 35.

Functions and Macros

`atomic_add()`, `atomic_and()`, `atomic_or()`

```
long atomic_add(long *atomic_variable, long add_value)
long atomic_and(long *atomic_variable, long and_value)
long atomic_or(long *atomic_variable, long or_value)
```

These functions perform the named operations (addition, bitwise AND, or bitwise OR) on the value found in *atomic_variable*, thus:

```
*atomic_variable += add_value
*atomic_variable &= and_value
*atomic_variable |= or_value
```

The functions return the previous value of **atomic_variable* (in other words, they return the value that *atomic_variable* pointed to before the operation was performed).

The significance of these functions is that they’re guaranteed to be *atomic*: If two threads attempt to access the same atomic variable at the same time (through these functions), one of the two threads will be made to wait until the other thread has completed the operation and updated the *atomic_variable* value.

class_name(), is_instance_of(), is_kind_of(), cast_as()

```

class_name(object)
is_instance_of(object, class)
is_kind_of(object, class)
cast_as(object, class)

```

These macros are part of the class information mechanism described in “Class Information” on page 5. In all cases, *object* is a pointer to an object, and *class* is a class designator (such as, literally, **BView** or **BFile**) and *not* a string (not “BView” or “BFile”).

class_name() returns a pointer to the name of *object*’s class.

is_instance_of() returns **TRUE** if *object* is a direct instance of *class*.

is_kind_of() returns **TRUE** if *object* is an instance of *class*, or if it inherits from *class*.

cast_as() if *object* is a kind of *class* (in the **is_kind_of()** sense), then **cast_as()** returns a pointer to *object* cast as an instance of *class*. Otherwise it returns **NULL**.

min(), max()

```

<support/SupportDefs.h>

min(a, b)
max(a, b)

```

These macros compare two integers or floating-point numbers. **min()** returns the lesser of the two (or *b* if they’re equal); **max()** returns the greater of the two (or *a* if they’re equal).

read_16_swap(), read_32_swap(), write_16_swap(), write_32_swap()

```

short read_16_swap(short *address)
long read_32_swap(long *address)

void write_16_swap(short *address, short value)
void write_32_swap(long *address, long value)

```

The **read...** functions read a 16- or 32-bit value from *address*, reverse the order of the bytes in the value, and return the swapped value directly.

The **write...** functions swap the bytes in *value* and write the swapped value to *address*.

`real_time_clock()`, `set_real_time_clock()`, `time_zone()`,
`set_time_zone()`

```
long real_time_clock(void)
void set_real_time_clock(long seconds)

long time_zone(void)
void set_time_zone(long seconds)
```

These functions measure and set time in seconds:

- `real_time_clock()` returns a measure of the number of seconds that have elapsed since the beginning of January 1st, 1970. `time_zone()` is a time-zone based offset, in seconds, that you can add to the value returned by `real_time_clock()` to get a notion of the actual (current) time of day.
- `set_real_time_clock()` and `set_time_zone()` set the values for the system's clock and time zone variables.

Warning: The `time_zone()` and `set_time_zone()` functions are currently unimplemented. If you call them, you will crash.

These functions aren't intended for scrupulously accurate measurement.

See also: `system_time()` in the Kernel Kit

`write_16_swap()` see `read_16_swap()`

`write_32_swap()` see `read_16_swap()`

Constants

Boolean Constants

<support/SupportDefs.h>

<u>Defined constant</u>	<u>Value</u>
<code>FALSE</code>	0
<code>TRUE</code>	1

These constants are used as values for `bool` variables (the `bool` type is listed in the next section).

Empty String

```
<support/SupportDefs.h>
const char *B_EMPTY_STRING
```

This constant provides a global pointer to an empty string (“”).

NULL and NIL

```
<support/SupportDefs.h>



| <u>Defined constant</u> | <u>Value</u> |
|-------------------------|--------------|
| <b>NIL</b>              | 0            |
| <b>NULL</b>             | 0            |


```

These constants represent “empty” values. They’re synonyms that can be used interchangeably.

Defined Types

bool

```
<support/SupportDefs.h>
typedef unsigned char bool
```

This is the Be version of the basic boolean type. The **TRUE** and **FALSE** constants (listed above) are defined as boolean values.

Function Pointers

```
<support/SupportDefs.h>
typedef int (*B_PFI)()
typedef long (*B_PFL)()
typedef void (*B_PFV)()
```

These types are pointers to functions that return **int**, **long**, and **void** values respectively.

Unsigned Integers

```
<support/SupportDefs.h>
typedef unsigned char uchar
typedef unsigned int uint
```

```
typedef unsigned long ulong  
typedef unsigned short ushort
```

These type names are defined as convenient shorthands for the standard unsigned types.

Volatile Integers

```
<support/SupportDefs.h>  
typedef volatile char vchar  
typedef volatile int vint  
typedef volatile long vlong  
typedef volatile short vshort
```

These type names are defined as shorthands for declaring volatile data.

Volatile and Unsigned Integers

```
<support/SupportDefs.h>  
typedef volatile unsigned char vuchar  
typedef volatile unsigned int vuint  
typedef volatile unsigned long vulong  
typedef volatile unsigned short vushort
```

These type names are defined as shorthands for specifying an integral data type to be both unsigned and volatile.

Error Codes

Error codes are returned by various functions to indicate the success or to describe the failure of a requested operation. All Be error constants except for **B_NO_ERROR** are negative integers; any function that returns an error code can thus be generally tested for success or failure by the following:

```
if ( funcCall() < B_NO_ERROR )
    /* failure */
else
    /* success */
```

Furthermore, all constants (except **B_NO_ERROR** and **B_ERROR**) are less than or equal to the value of the **B_ERRORS_END** constant. If you want to define your own negative-valued error codes, you should begin with the value (**B_ERRORS_END** + 1) and work your way toward 0.

General Error Codes

<support/Errors.h>

<u>Error Code</u>	<u>Meaning</u>
B_NO_MEMORY	There's not enough memory for the operation.
B_IO_ERROR	A general input/output error occurred.
B_PERMISSION_DENIED	The operation isn't allowed.
B_FILE_ERROR	A file error occurred.
B_FILE_NOT_FOUND	The specified file doesn't exist.
B_BAD_INDEX	The index is out of range.
B_BAD_VALUE	An illegal value was passed to the function.
B_MISMATCHED_VALUES	Conflicting values were passed to the function.
B_BAD_TYPE	An illegal argument type was named or passed.
B_NAME_NOT_FOUND	There's no match for the specified name.
B_NAME_IN_USE	The requested (unique) name is already used.
B_TIMED_OUT	Time expired before the operation was finished.
B_INTERRUPTED	A signal interrupted the operation.
B_ERROR = -1	This is a convenient catchall for general errors.
B_NO_ERROR = 0	Everything's OK.
B_ERRORS_END	Marks the end of all Be-defined error codes.

Application Kit Error Codes

<support/Errors.h>

Error Code

B_DUPLICATE_REPLY
B_BAD_REPLY
B_BAD_HANDLER

B_MESSAGE_TO_SELF
B_ALREADY_RUNNING
B_LAUNCH_FAILED

Meaning

A previous reply message has already been sent.
 The reply message is inappropriate and can't be sent
 The designated message handler isn't valid.

 A thread is trying to send a message to itself.
 The application can't be launched again.
 The attempt to launch the application failed.

These constants are defined for the messaging classes of the Application Kit. The messaging system also makes use of some of the general errors and kernel errors described above.

See also: **BMessage::Error()** and **BMessenger::Error()**

Debugger Error Codes

<support/Errors.h>

Error Code

B_DEBUGGER_ALREADY_INSTALLED

This constant signals that the debugger has already been installed for a particular team and can't be installed again.

Kernel Kit Error Codes

<support/Errors.h>

Error Code

B_BAD_THREAD_ID
B_BAD_THREAD_STATE
B_NO_MORE_THREADS

B_BAD_TEAM_ID
B_NO_MORE_TEAMS

B_BAD_PORT_ID
B_NO_MORE_PORTS

B_BAD_SEM_ID
B_NO_MORE_SEMS

Meaning

Specified thread identifier (**thread_id**) is invalid.
 The thread is in the wrong state for the operation.
 All thread identifiers are currently taken.

 Specified team identifier (**team_id**) is invalid.
 All team identifiers are currently taken.

 Specified port identifier (**port_id**) is invalid.
 All port identifiers have been taken.

 Semaphore identifier (**sem_id**) is invalid.
 All semaphores are currently taken.

B_BAD_IMAGE_ID Specified image identifier (image_id) is invalid.

These error codes are returned by functions in the Kernel Kit, and occasionally by functions defined in higher level kits.

Media Kit Error Codes

<support/Errors.h>

<u>Error Code</u>	<u>Meaning</u>
B_STREAM_NOT_FOUND	The attempt to locate the stream failed.
B_SERVER_NOT_FOUND	The attempt to locate the server failed.
B_RESOURCE_NOT_FOUND	The attempt to locate the resource failed.
B_RESOURCE_UNAVAILABLE	Permission to access the resource was denied.
B_BAD_SUBSCRIBER	The BSubscriber is invalid.
B_SUBSCRIBER_NOT_ENTERED	The BSubscriber hasn't entered the stream.
B_BUFFER_NOT_AVAILABLE	The attempt to acquire the buffer failed.

These error codes are defined for the Media Kit. See the classes and functions in that kit for an explanation of how they're used.

