Secret ~~Source~~ Sauce

Using or Duplicating Apple's Coolest Internal Functionality

Jim Dovey
iTeam Lead Developer
Kobo Inc.
@alanQuatermain

http://alanquatermain.net/

What this talk is *not* about

- OMG private APIs w00t!
- I can be a l337 h4xx0rz t00
- Process invasion
- Function patching

What this talk *is* about

- Crash Reporting
  - How crashes are reported
  - How to interpret crash messages
  - How to remotely backtrace a process
- Time Machine
  - Who will use this?
  - Concepts and design
    - UI elements, events
  - Public API (a whole *two* functions!)
  - Private API
  - An example Cocoa controller class

There will be code…

Crash Reporting

Mach

Mach IPC

- The building blocks for the Mach microkernel
- Used at the system level
  - Virtual memory pager
  - IOKit messaging
  - Task/thread kernel APIs
- Used at the user level
  - Distributed Objects system
  - CoreGraphics/Quartz
  - Many many many public APIs and services
- Based around *ports*
  - Think of them as like BSD sockets and you'll be golden

Mach Interface Generator

- Domain-specific language
- Used to define IPC routines and custom types
- Command-line tool to generate C code
  - Separate code is generated for both user and server

MIG Code Sample

```
routine run_command
(
    helper                  : mach_port_t;
    authorization           : authInfo_t;
    arguments               : propList_t;
    arguments_ool           : pointer_t, Dealloc;
out xml_data                : xmlData_t;
out xml_data_ool            : pointer_t, Dealloc
);
```

C Implementation

- User side is all done for you
  - MIG generates headers and source code
- Server side is partially complete
  - MIG generates message parsers
  - You must implement actual IPC endpoints

C Function Names

```
userprefix fcsmig_;
serverprefix fcsmig_do_;
```

Generated User C Functions:

```
kern_return_t fcsmig_run_command( ... );
```

Required Server C Functions:

```
kern_return_t fcsmig_do_run_command( ...);
```

A Real-Life Example: Exceptions

```
ServerPrefix catch_;
routine exception_raise(
        exception_port  : mach_port_t;
        thread          : mach_port_t;
        task            : mach_port_t;
);
routine exception_raise_state(
        exception_port  : mach_port_t;
        exception       : exception_type_t;
        code            : exception_data_t, const;
inout   flavor          : int;
        old_state       : thread_state_t, const;
  out   new_state       : thread_state_t);
```

- No UserPrefix
  - Users call `exception_raise(...);`
  - Servers implement `catch_exception_raise(...);`

All very nice, but…

What does this have to do with crashes?

- Low-level exceptions can be handled
- UNIX Signals
  - SIGABRT, SIGBUF, SIGILL
  - Sent to process which triggered them
- Mach Exceptions
  - Three routines:
    - exception_raise, exception_raise_state, exception_raise_state_identity
  - Sent to the designated *exception handler port* for the triggering process
    - It's just a port, so it can be received by *any* process

What decides on the exception ports?

- By default the kernel handles them
  - Sends a signal to the triggering process
- Anyone who asks
  - Can be triggering process (in a dedicated thread)
  - Can be something else
- On OS X, launchd sets up an exception handler for everything

Launchd exception handlers

- Special flag in launchd job property lists
  - `MachExceptionHandler = true`
- Causes launchd to register an exception handler server and launch this process to forward them on
- Default on OS X is setup by com.apple.ReportCrash.Root.plist
- Actual crash reports are created and logged by ReportCrash app in /System/Library/CoreServices

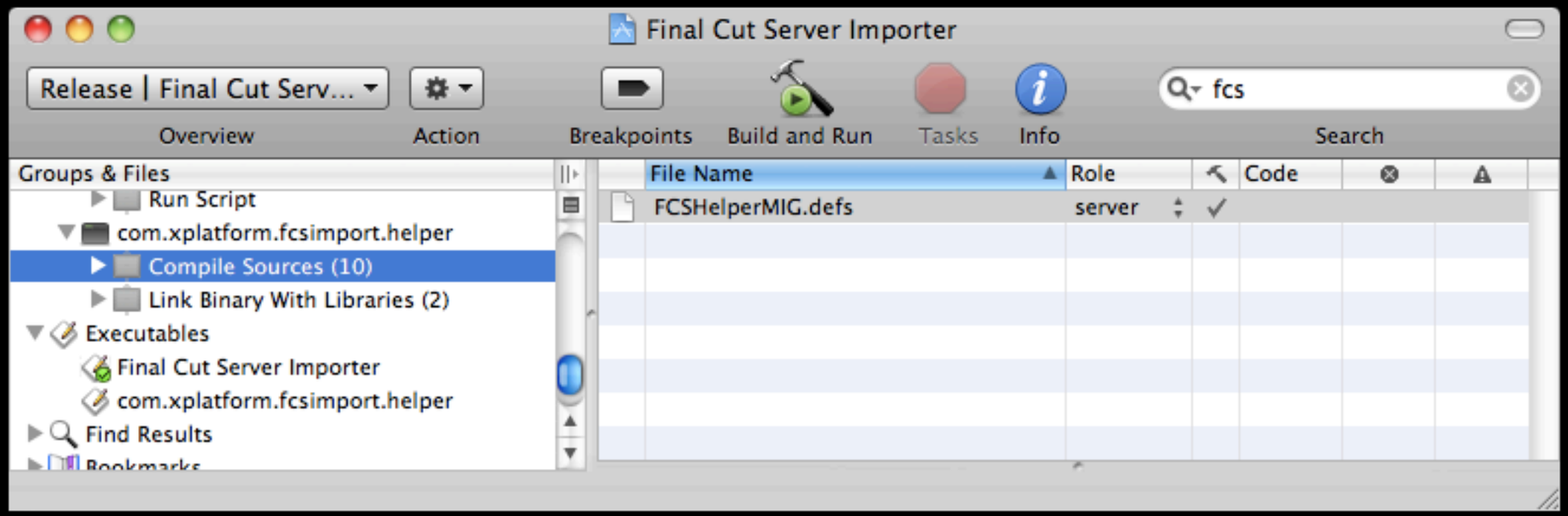
Rolling Your Own

In your .app Bundle:

- Create a new command-line tool
- Call it OMG_FAIL
  - It's funny, laugh
- Launch it from your .app when that launches
- Pass it your process ID in its parameters
  - Get this from -`[NSProcessInfo processIdentifier]`
- This tool will now register to receive exceptions for your .app
- BUT:
  - Make your tool setgid 'procmod' — or else it can't access your .app's task-port to attach the handlers

Creating the exception server

- Copy in exc.defs
- Set to build as ‘server’ in the target’s Compile Sources section



Create a Service Mach Port

```
#include <mach/message.h>
#include <mach/mach_error.h>
#include <mach/task.h>
#include <mach/port.h>

kern_return_t kr = KERN_SUCCESS;
mach_port_t exc_catcher_port = MACH_PORT_NULL;

kr = mach_port_allocate(mach_task_self(),
MACH_PORT_RIGHT_RECEIVE, &exc_catcher_port);
if ( kr != KERN_SUCCESS ) {
    // print mach_error_string(kr)
    return;
}
```

Make It Writable

```
kr = mach_port_insert_right(mach_task_self(),
exc_catcher_port, exc_catcher_port,
MACH_MSG_TYPE_MAKE_SEND);
if ( kr != KERN_SUCCESS ) {
    // print mach_error_string(kr)
    return;
}
```

Receive Exception Messages

```
dispatch_source_t exc_source =
dispatch_source_create(
  DISPATCH_SOURCE_TYPE_MACH_RECV,
  exc_catcher_port, 0,
  dispatch_get_main_queue() );

// handle new messages when they arrive
dispatch_source_set_event_handler(
  exc_source, ^{
    ...
  });
```

Handle Exceptions

```
mach_msg_header_t *msg, *reply;
kern_return_t krc;

#define MSG_SIZE 512
msg = alloca(MSG_SIZE);
reply = alloca(MSG_SIZE);

// read the incoming message
krc = mach_msg( msg, MACH_RCV_MSG, MSG_SIZE,
MSG_SIZE, exc_catcher_port, 0, MACH_PORT_NULL );
// check error for failure
```

Handle Exceptions (continued)

```
// demux and dispatch the message to the
// appropriate handler
// exc_server is generated for us by MIG
if ( exc_server(msg, reply) == false ) {
        // print error
        return;     // returns from the block
    }
```

Handle Exceptions (continued)

```
// we've got the reply from our handler function
// now we need to send it back
(void) mach_msg( reply, MACH_SEND_MSG, reply-
>msgh_size, 0, msg->msgh_local_port, 0,
MACH_PORT_NULL );
```

- Our dispatch source is called when a message arrives
- We read it using `mach_msg(..., MACH_RCV_MSG, ...)`
- We pass the message and a reply buffer to the MIG-generated demux routine `exc_server()`
- If that succeeded, we need to send the reply back to the sender

Registering to Receive Exceptions

```
// define our data structure to keep hold of
// old exception handler chain
#define HANDLERS 64
typedef struct _ExceptionPorts {
    mach_msg_type_number_t  maskCount;
    exception_mask_t        masks[HANDLERS];
    exception_handler_t     handlers[HANDLERS];
    exception_behavior_t    behaviors[HANDLERS];
    thread_state_flavor_t   flavors[HANDLERS];
} ExceptionPorts;
static ExceptionPorts * gOldHandlerData = NULL;
```

Registering (continued)

```
pid_t procID = ...;   // get target process ID
task_t task = MACH_PORT_NULL;
kern_return_t kr = KERN_SUCCESS;
// fetch target’s task-port
kr = task_for_pid( mach_task_self(), procID,
&task );
if ( kr != KERN_SUCCESS )
{
    // don’t have privileges to get task ports
    // this is why we need setgid ‘procmod’
    return;
}
```

Registering (continued)

```
// copy old exception handlers
gOldHandlerData = calloc(...);
gOldHandlerData->maskCount = HANDLERS;
kr = task_get_exception_ports(
    task, EXC_MASK_ALL,
    gOldHandlerData->masks,
    gOldHandlerData->maskCount,
    gOldHandlerData->behaviors,
    gOldHandlerData->flavors );
if ( kr != KERN_SUCCESS ) {
    // handle error
}
```

Registering (continued)

```
// install new ports
kr = task_set_exception_ports( task,
    EXC_MASK_ALL &
        ~(EXC_MASK_MACH_SYSCALL|
          EXC_MASK_SYSCALL|EXC_MASK_RPC_ALERT),
    exc_catcher_port, EXCEPTION_DEFAULT,
    THREAD_STATE_NONE );

if ( kr != KERN_SUCCESS ) {
    // handle error
}
```

Cleanup

- When the app we're monitoring quits or crashes, we want to reset the old exception ports, so we don't hose the system
- In dispatch source parlance, we can do this with a source cancellation block
- The old ports may be empty, they may be many and complex: but it's easy enough to reset them all.

Source Cancellation Handler

```
int i;
for (i = gOldHandlerData->maskCount-1; i>=0 i++)
{
    if ( gOldHandlerData->handlers[i] ==
MACH_PORT_NULL )
        break;
    task_set_exception_ports( task,
        gOldHandlerData->masks[i],
        gOldHandlerData->handlers[i],
        gOldHandlerData->behaviors[i],
        gOldHandlerData->flavors[i] );
}
// destroy our exception handler port
mach_port_destroy( mach_task_self(),
    exc_catcher_port );
```

One more piece of groundwork

- Determining when our target has quit
- We can use the target's task-port for this
- Ask the server to send us a dead-name notification
  - Tells us when the task has become invalid, i.e. been destroye

Task Death Notifications

- Create a mach port, just like we did earlier; no need to make it writable
- Wrap it in a dispatch source
- In the dispatch source's cancellation method, we destroy the notification port we just created
- In the source handler method, we cancel both this source and the exception source.

Task Death Notification Request

```
mach_port_t death_port = MACH_PORT_NULL;
mach_port_t old_port = MACH_PORT_NULL;
kr = mach_port_allocate( mach_task_self(),
    MACH_PORT_RIGHT_RECEIVE, &death_port );
kr = mach_port_request_notification(
    mach_task_self(), theTask,
    MACH_NOTIFY_DEAD_NAME, 0, death_port,
    MACH_MSG_TYPE_MAKE_SEND_ONCE, &old_port );
```

Task Death Source Cancellation

```
// when cancelled, reset the old port
// (this may be MACH_PORT_NULL)
mach_port_request_notification(
    mach_task_self(), task,
    MACH_NOTIFY_DEAD_NAME, 0, old_port,
    0, NULL );

// the source owns our port, so we'll destroy it
mach_port_destroy( mach_task_self(),
    death_port );
```

Task Death Source Handler

```
mach_msg_header_t * msg;
msg = alloca(MSG_SIZE);

// consume the message
(void) mach_msg( msg, MACH_RCV_MSG, MSG_SIZE,
    MSG_SIZE, death_port, 0, MACH_PORT_NULL );

// the task has gone-- log the crash report
// we only do this once the app is gone, in case
// a logged exception isn't fatal
backtrace_log();

// the task we're watching is gone: quit now
dispatch_source_cancel( exc_source );
dispatch_source_cancel( death_source );
```

Running it

- Dispatch sources are created in a suspended state, so we must call `dispatch_resume()` on them all
- This is a vanilla C app, so we run the event loop using `dispatch_main()`
- There is no step three.

Exception Processing

Required Tasks

- Implement the three exception handler routines
  - catch_exception_raise
  - catch_exception_raise_state
  - catch_exception_raise_state_identity
- Implement an exception forwarding routine
  - We don’t actually handle the exceptions, we just want to know when they happen

Exception Handler Types

- catch_exception_raise
  - Smallest. Receives task, thread, exception code & data
- catch_exception_raise_state
  - Adds state of the thread causing the exception, removes task and thread ports
- catch_exception_raise_state_identity
  - All of the above

Forwarding Exceptions

- Iterate through the old exception handlers to find the first which wants to handle this one
- Check what type of method it wants
  - Could be any of the previous three
  - This is where the headache appears
    - Means our forward_exception implementation should take ALL possible parameters
- Build parameters for that method
- Call exception_raiseXXX passing those parameters, with the handler port as the first parameter

forward_exception()

```
kern_return_t forward_exception(
    thread_t thread,
    mach_port_t task,
    exception_type_t exception,
    exception_data_t code,
    mach_msg_type_number_t codeCount,
    int *flavor,
    thread_state_t old_state,
    mach_msg_type_number_t old_stateCnt,
    thread_state_t new_state,
    mach_msg_type_number_t *new_stateCnt )
```

```
forward_exception()

for ( portIndex = 0;
      portIndex < gOldHandlerData->maskCount;
      portIndex++ ) {
    if ( gOldHandlerData->masks[portIndex] &
         (1 << exception) )
    {
        // This handler wants the exception
        break;
    }
}

if ( portIndex >= gOldHandlerData->maskCount ) {
    // nothing wanted it
    return ( KERN_FAILURE );
}
```

forward_exception()

```
case EXCEPTION_DEFAULT:
    kr = exception_raise( port, thread, task,
            exception, code, codeCount );
    break;
case EXCEPTION_STATE:
    kr = exception_raise_state( port, exception,
            code, codeCount, flavor,
            old_state, old_stateCnt,
            new_state, new_stateCnt );
    break;
case EXCEPTION_STATE_IDENTITY:
    kr = exception_raise_state_identity( port,
            thread, task, exception, code,
            codeCount, flavor, old_state,
            old_stateCnt, new_state,
            new_stateCnt );
```

forward_exception()

```
if ( behaviour != EXCEPTION_DEFAULT )
{
    kr = thread_set_state( thread, *flavor,
                           new_state, *new_stateCnt );
}
return ( kr );
```

Finally: The Exception Handlers

```
kern_return_t catch_exception_raise(
    mach_port_t exception_port,
    thread_t thread,
    mach_port_t task,
    exception_type_t exception,
    exception_data_t code,
    mach_msg_type_number_t codeCnt
)
{
    kern_return_t kr;
    backtrace_task(task, thread, code, codeCnt);
    kr = forward_exception( thread, task,
                            exception, code, codeCnt,
                            NULL, NULL, 0, NULL, 0 );
    return ( kr );
}
```

Remote Backtracing

Backtracing Tasks

- Walk up the stack
- The prior stack frame address is in a known place
  - Register (address)
  - Register + offset (also an address)
- The return-to address is stored next to this on the stack
- Grab the process counter (current instruction address) then:
  - Read the return-to address
  - Read the prior frame address
  - Go up to prior frame
  - Rinse & repeat until you see a value of 0 or -1

Local Tracing Is (Now) Easy

- In Mac OS X 10.6, Apple opened the `backtrace()` function
  - It's in libSystem.dylib, so you link it for free
  - Declared in `<execinfo.h>`
- That function returns addresses
- Companion function—`backtrace_symbols()`—to fetch function names (symbols) for each address

Not so easy for remote values

- Get process counter value from a Mach `thread_state_t`; this contains the registers saved by the kernel for context-swapping
- The addresses contained there can't be directly read— no guarantee the same data is at the same address in your app
- Frame addresses particularly— your stack is going to be *completely* different, but its start address will likely be the same as it's set by convention
- We need to read data from another process
  - But how?

Mach Virtual Memory APIs

- The Mach system has public functions which will let you read from any process's memory address space
- You need a valid task-port to access that (e.g. setgid procmod)
- A few functions we would use:
  - `vm_map()` will take a chunk of real memory used by another process and put it into your address space (at a different address)
  - `vm_read()` enables you to read page-sized chunks of memory from another process into a buffer allocated for you
  - Using `vm_map()` will NOT copy— it acts like regular paged virtual memory, so you can ask for a lot

HOLY SWEET &@^#

THAT'S A LOT

…which is why we won’t do that (today)

Symbolication.framework

(Private APIs w00t!)

Our Backtracer

```
if ( gBacktraceLog == nil )
    gBacktraceLog = [NSMutableString new];
else
    [gBacktraceLog setString: @""];
```

Our Backtracer

```
VMUSymbolicator * symbolicator =
[VMUSymbolicator symbolicatorForTask: task];

NSArray * samples = [VMUSampler
          sampleAllThreadsOfTask: task
                    withSymbolicator: symbolicator];
```

Our Backtracer

```
NSUInteger i = 0;
for ( VMUBacktrace * backtrace in samples )
{
    [gBacktraceLog appendFormat: @"Thread %d
(%#x)", i, [backtrace thread]];
    if ( [backtrace thread] == exc_thread )
        [gBacktraceLog appendString: @"
 Crashed"];
    [gBacktraceLog appendString: @":\n"];
```

Our Backtracer

```
pointer_t * trace = [backtrace backtrace];
for ( int j = 0;
      j < [backtrace backtraceLength];
      j++ )
{
    VMUSymbol * symbol = [symbolicator
                  symbolForAddress: trace[j]];
        VMUSymbolOwner * owner = [symbolicator
                  symbolOwnerForAddress: trace[j]];
        [gBacktraceLog appendFormat: @"%d\t%-30s
                  %p : %@\n", j,
                  [[owner name] UTF8String],
                  trace[j], [symbol name]];
}
```

That's It?

Yes.

Here's what you get: (minus `\n`)

```
Thread 7 (0x3f03):
0   libSystem.B.dylib                0x0000008885efca :
__semwait_signal
1   libSystem.B.dylib                0x00000088862de1 :
_pthread_cond_wait
2   JavaScriptCore                   0x00000082f8d1a0 :
WTF::ThreadCondition::timedWait(WTF::Mutex&, double)
3   WebCore                          0x00000080bd4dd1 :
WebCore::LocalStorageThread::threadEntryPoint()
4   libSystem.B.dylib                0x0000008885d536 :
_pthread_start
5   libSystem.B.dylib                0x0000008885d3e9 :
thread_start
```

It's Also Open-Source

- Going up on github *real soon now*
- Two command-line tools:
  - **backtracer**— Use `backtracer -ProcID <process_id>`. Generated the sample you just saw.
  - **CrashMonitor**— designed to drop into your own apps to do your own crash reporting. Have it launch another app to send that crash report to you.

Integration with Time Machine

What we won't cover

- Snapshots
  - Used by Xcode ?
  - Useful for managing non-bundled collections of discrete files
- Triggering backups programatically
  - BUBackUpNow( ) function

Who will use this?

- Applications managing collections of data
  - Address Book, Mail, iPhoto
  - iTunes, iCal
  - Library, Ledgers, CRM
- Apps with a desire to handle partial dataset restorations
- CoreData

Concepts and Design

Time Machine User Interface

- One large fullscreen window
- A collection of images
  - Time Machine ‘windows’ aren’t (necessarily) actual windows
  - Each instance is an image, usually taken from a simple window via CGContextXXX() functions.
- Time Machine scrolls through these windows for you
- Your app is alerted when a real window is required, and your app handles display & input for that window.

Events and Callbacks

Time Machine handles the interface for you— you only have to provide some callback routines.

```
BURegisterStartTimeMachineFromDock(...);
BURegisterRequestSnapshotImage(...);
BURegisterTimeMachineDismissed(...);
BURegisterTimeMachineRestore(...);
```

The ‘events’ posted by Time Machine include the startup request, actions, dismissal (cancel), restore (one or all), activate/deactivate snapshot windows, and requests for snapshot or thumbnail images.

API

Public API

Apple has released two functions:

```
CSBackupIsItemExcluded(CFURLRef item, Boolean * byPath);
CSBackupSetItemExcluded(CFURLRef item, Boolean exclude,
Boolean byPath);
```

These routines allow you to inform the backup system of cache files or other oft-changed data which need not be backed up.

Anything further than this requires that we resort to accessing the private API…

Private API

- Request notification of Time Machine invocation
- Provide callbacks for the Time Machine engine, then start Time Machine itself
  - If in a non-applicable state, don't start time machine
  - Modal loops, active document is untitled/unsaved
- Answer callbacks to provide snapshot window images corresponding to backup data
- Handle activation and deactivation of individual snapshots
- Restore if so requested, or else revert to prior state upon dismissal.

Startup

- When your app starts, call BURegisterStartTimeMachineFromDock();
  - Your callback returns nothing and takes no arguments.
- The callback will fire when the user clicks the Time Machine icon in the dock. It's still up to you to launch the Time Machine UI, however.

```
typedef void (*BUStartTimeMachineCallBack)(void);
void BURegisterStartTimeMachineFromDock(BUStartTimeMachineCallBack
        cb);
void BUStartTimeMachine(int windowNumber, CFURLRef urlForWindow,
        BUAction flags);
```

Data Callbacks

- Upon receiving the startup call, you register your other callbacks, to provide data and handle events
- Time Machine provides request callbacks for window snapshots and for thumbnail images, but we'll just use snapshots.
- To generate a snapshot image, create a window for the data at the given URL, and call BUUpdateSnapshotImage(), providing the CG window number (using -[NSWindow windowNumber]) and the provided URL as parameters.

```
typedef void (*BURequestSnapshotImageCallBack)(void * token,
        CFURLRef backupURL);
void BURegisterRequestSnapshotImage(void * token,
        BURequestSnapshotCallBack callback);
void BUUpdateSnapshotImage(int windowNumber, CFURLRef url);
```

Snapshot Events

- You must provide callbacks to be notified when snapshots are focussed or blurred.
- When these callbacks are called, the application must display or remove a window at the given coordinates.
- When done processing, call BUActivatedSnapshot() or BUDeactivatedSnapshot() as appropriate.

```
typedef void (*BUActivateSnapshotCallBack)(void * token, CFURLRef
        backupURL, CGRect workingBounds);
typedef void (*BUDeactivateSnapshotCallBack)(void * token, CFURLRef
        backupURL);
void BURegisterActivateSnapshot(void * token,
        BUActivateSnapshotCallBack callback);
void BURegisterDeactivateSnapshot(void * token,
        BUDeactivateSnapshotCallBack callback);
void BUActivatedSnapshot(int windowNumber, CFURLRef url);
void BUDeactivatedSnapshot(int windowNumber, CFURLRef url);
```

Action Callbacks

- Two main actions: restore and dismiss
- Restore provides a flag to indicate whether to restore all items or just a selection.
- Dismissal only triggers *after* the Time Machine UI has gone away.
- To programatically dismiss, call `BUTimeMachineAction(1);`

```
typedef void (*BUTimeMachineDismissedCallBack)(void * token);
typedef void (*BUTimeMachineRestoreCallBack)(void * token, CFURLRef
        backupURL, CFURLRef liveURL, Boolean restoreAll,
        CFDictionaryRef userInfo);
void BURegisterTimeMachineDismissed(void * token,
        BUTimeMachineDismissedCallBack callback);
void BURegisterTimeMachineRestore(void * token,
        BUTimeMachineRestoreCallBack calback);
void BUTimeMachineAction(BUAction action);
```

Cocoa Controller

AQTimeMachineController

- Implemented in Objective-C 2.0
- Singleton class
- Designed to handle most of the work for you
  - You shouldn't need to call BUxxxx() methods yourself
- You implement a delegate to provide application-specific data
  - Ideally this delegate should be concerned only with Time Machine, and should be your *only* Time Machine-handling class

Properties

- @property(assign) id<AQTimeMachineDelegate> __weak delegate;
  - Synchronized access, non-retaining
- @property NSRect workingBounds;
  - The current snapshot bounds set by Time Machine
- @property BOOL changedItemsOnly;
  - YES if the UI should only show changed items
- @property BOOL inTimeMachine;
  - Check to see if Time Machine actions should be performed

General Functions

- + (AQTimeMachineController *) timeMachineController;
  - Fetch the singleton instance
- - (BOOL) canEnterTimeMachine;
  - A simple check, will call the delegate
- - (IBAction) browseBackups: (id) sender;
  - When you want your own Time Machine button
- - (void) dismissTimeMachine;
  - Close down the Time Machine UI
- - (void) invalidateSnapshotImages;
  - When your UI has changed, updates snapshots

Controller Tasks

- Handles Time Machine startup notifications
  - *Requires a delegate to be set prior to this*
- Stores the window state of the initial window, and restores this state when Time Machine is dismissed
  - Miniaturized, visible
- Maintains a list of window controller to URL mappings, one for each snapshot window
- Handles updates to snapshot images
- Activates and deactivates snapshots, notifying delegate
- Calls delegate when a restore action is requested

AQTimeMachineController Code

Delegate Tasks

- Determines whether the app can enter Time Machine
- Creates and returns controllers and data paths for the live window and any snapshot windows requested
- Implements data restoration
- Optionally:
  - Performs setup before & after entering Time Machine
  - Performs actions before & after snapshot activation/deactivation
  - Makes any changes required for ‘show changed items only’
  - Any app-specific cleanup when Time Machine is dismissed

An NSDocument-based Delegate

Useful Data

- Keep a record of all snapshot NSDocuments, indexed by path or URL
- Keep track of the current document
- Store any document user-interface state which is likely to change while in Time Machine
  - Search box contents, list selections
- Ensure that no documents are editable while in Time Machine

-canEnterTimeMachine

- Check for modal panels:
  - [[[NSRunLoop mainRunLoop] currentMode] isEqualToString: NSModalPanelRunLoopMode]
- Check for an open & stored current document:
  - [[NSDocumentController sharedDocumentController] currentDocument]
- Document must have window controllers
- No sheet should be attached:
  - [[ctrl window] attachedSheet]

Snapshot window controllers

- You can create NSDocuments for backup snapshots, but it's a good idea to limit them a little
  - Create using -[NSDocumentController makeDocumentWithContentsOfURL:ofType:error:]
  - Use -makeWindowControllers to setup the controllers, rather than letting NSDocument put itself onscreen

Updating snapshots

- Implement the optional notification handlers to store and set data at appropriate times
- Store UI state:
  - Before Time Machine activates
  - When deactivating snapshots
- Set UI state:
  - When activating snapshots
  - When restoring or dismissing Time Machine
- Also install your own handlers to invalidate & update snapshots in response to user activity
  - Notifications, delegates, KVO

Example Delegate Code

I Want Friends!

- **Twitter**— @alanQuatermain
  - NB: *My name is NOT Alan*
- **Web**— http://quatermain.tumblr.com/
  - This contains an email link, LinkedIn, Facebook (sigh), etc.