



ZOOM OUT BREW 2008 CONFERENCE

Dispatcher, Events, Callbacks and Threads

Rob Walker, Principal Engineer,
Qualcomm Incorporated

QUALCOMM



Agenda

- Event-driven programming
- BREW[®] Events
- BREW Callbacks
- Using AEECallback to receive notification
- Notifying via AEECallback
- IThread



“Sequential Programming”

- A sequence of operations are listed to be performed one after another. A thread of execution performs one action after another, waiting as necessary on external events:
 - read userid & password
 - connect to TCP port
 - read protocol message
 - write protocol message
 - ...
- More than one operation proceeding "simultaneously" requires multiple threads.



Event-Driven Model

- Complex UIs and protocols demand a different approach
 - The sequence of events is not dictated by the programmer
 - User can choose one or many options
 - Even simple protocols must deal with cancellation
- Event-driven
 - Decompose system into objects
 - Objects are invoked when events occur
 - Object can trigger other events



Event-Driven Issues

- In event-driven environments, with lots of objects calling each other recursively, some problems can arise.
 - 1. Stack exhaustion
 - Very deep nesting
 - 2. Re-entry data consistency problems
 - The caller or caller's caller (and so on) could be using data you need to modify.
 - 3. Long-lived operations
 - Something may take a long time; don't want to pre-empt other operations
 - Interferes with appearance of concurrency



Avoiding Re-entry Problems

- We can avoid this by asynchronous dispatching (versus synchronous calling)
 - Event queues
 - Queued callbacks
- Where A calls B synchronously, B does not call A synchronously
 - Call flow is a directed acyclic graph
 - Example:
 - Sockets: client calls Read(), Write() ... gets notified via callbacks



Standard GUI Model

- Events are placed on a queue
- Events contain information about what happened
 - User input (key presses, etc.)
 - Application-specific information
- A thread dispatches events

```
while (GetEvent(&e)) {  
    // dispatch event e  
}
```

- Multiple objects can execute in the same thread



BREW Event Queue

- There is one global event queue
- Input devices generate EVT_KEY, etc.
- Applications can “post” events onto the queue
 - EVT_USER : base for range of events that an application can define for its own purposes
- BREW implements the dispatch loop
 - Applets do not implement dispatch loop
 - IApplet is passed (an event target)



Send & Post

- ISHELL_PostEvent()
 - Places events on the queue
- ISHELL_SendEvent()
 - Sends events *synchronously*
 - Destination app is determined just as when BREW dispatches queued events
- Restrictions
 - Event codes 0...7 cannot be sent/posted by apps



BREW Callbacks

- AEECallback holds function & context pointers
 - Notification is function call “directly” to an object
- No allocation for queuing
 - AEECallback pre-allocated space for queueing data
 - No payload
- No dangling pointers
 - Each scheduled operation can be canceled synchronously.



BREW Event Management

- BREW maintains queue of AEECallback structs
- One event queue for events and callbacks
 - Events are actually queued as individual AEECallbacks
- *All applets* share the same thread and queue
- System dispatch loop tracks 'current app'
 - When an event is delivered, IAPPLET_HandleEvent() is called in the appropriate “app context”
 - Callbacks scheduled by an app are called in the context of the apps that requested the callback



AEECallback Semantics

- When a function accepts AEECallback* ...
 - Passing AEECallback means “call me back later”
 - One-shot semantics
 - After callback, re-register if still interested in notification
 - Plus cancellation semantics...
- Two different categories of AEECallback calls:
 - “Notify me when something changes”
 - Cancel → prevent callback from happening
 - “Start this operation and notify when done”
 - Cancel → prevent callback and cease the asynch operation



AEECallback Semantics (2)

- Functions to request notification or start an operation return immediately
- “void” return values
 - Errors are typically delivered asynchronously
 - ISHELL_Resume() can be relied on to succeed
 - In callback function, client can get error status
 - No point in checking in two different places
 - Caller may have no way to deal with synchronous error
 - Caller may not be able to perform operation synchronously; must have some mechanism that guarantees success

Using AEECallback: 1

- Allocate AEECallbacks within your object

```
struct MyApplet {  
    IShell *piShell;  
    AEECallback cbkWeb;      ←  
    AEECallback cbkPosDet;  ←  
    ...  
};
```

- Each can support one ongoing operation at a time



Using AEECallback: 2

A) Initialize callbacks in your constructor

```
CALLBACK_Init(&me->cbkNetwork, MyApplet_WebCB, me);  
CALLBACK_Init(&me->cbkPosDet, MyApplet_PosDetCB,  
me);
```

(This assumes zero-initialized memory as with MALLOC().)

B) Cancel callbacks in your destructor

```
CALLBACK_Cancel (&me->cbkWeb);  
CALLBACK_Cancel (&me->cbkPosDet);
```

This avoids dangling pointers. Your object will not be called back after Cancel().

Using AEECallback: 3

- Start your operations...

```
IWEB_GetResponse(pIWeb, (pIWeb,  
    &me->resp,  
    &me->cbkWeb,  
    pszURL,  
    WEBOPT_USERAGENT,          "Spaminator/1.0",  
    WEBOPT_IDLECONNTIMEOUT,    120,  
    WEBOPT_END) );
```

- Wait for the callback function to be called

Using AEECallback: 4

- Cancel operations when necessary...

...

```
case EVT_SUSPEND:
```

```
    CALLBACK_Cancel (&me->cbkWeb);
```

```
    CALLBACK_Cancel (&me->cbkPosDet);
```

...

- It is safe to call Cancel() at any time

Implementing Asynch Notifications

- To notify asynchronously...
 - Implement a function that accepts an AEECallback
- Accepting AEECallback requires:
 - Queueing the request
 - Keep track of the fact that notification was requested
 - AEECallback structure helps here
 - Implementing a cancellation function
 - Take “ownership” of the callback while it is on your queue
 - Intercept client’s CALLBACK_Cancel() calls so you can release ownership and dequeue from your queue

AEECallback

```
AEECallback {
    AEECallback*      pNext;          // PRIVATE
    void*             pmc;            // PRIVATE
    CallbackCancelFunc* pfnCancel;   // Cancel
    void*             pCancelData;    // PRIVATE
    CallbackNotifyFunc* pfnNotify;   // User data
    void*             pNotifyData;    // User data
    void*             pReserved;     // PRIVATE
};
```

- “User data” is set by user via `CALLBACK_Init()`
- “PRIVATE” is for use by notifying object
 - Undefined contents (specific to implementation of notifying object)
- “Cancel” is used by `CALLBACK_Cancel()`
 - Managed (set/reset) by notifying object
 - Has pre-defined semantics



Asynch Notifications: Details (1)

- Take ownership
 - Cancel
 - Before doing anything else, cancel the callback just in case it is owned by some other notifying object
 - Enqueue
 - Keep track so you can notify when the time comes
 - May allocate data at this stage
 - Use pNext or other PRIVATE fields
 - Supply cancel function
 - pfnCancel is called when the client cancels the request
 - pfnCancel must de-queue (and free any allocated data)
 - Use pCancelData or other PRIVATE fields to recover context



Asynch Notifications: Details (2)

- When done
 - Call ISHELL_Resume()
 - Resume() will cancel the callback (calling your pfnCancel)
- On error
 - Make note of error condition in your object state
 - Schedule callback immediately using ISHELL_Resume()
 - After client's callback function is called, client will call to accept results
 - This works for errors at any time, including errors during the initial call that requests the notification.



Porting multi-threaded code

- Instead of sequential style “blocking” operations, must use callbacks instead
- Stack variables -> heap
 - Variables can remain on stack during blocking calls
 - While awaiting a callback, only heap-allocated variables persist
- Code must deal with cancellation
 - Code may not have originally allowed for interruption



IThread

- IThread supports user-level threading
 - Allows “sequential” style of programming
- Allows multiple threads of execution
 - Each stack is allocated from BREW heap
- Cooperative multitasking
 - All “IThreads” run in the same kernel thread as BREW
 - For each asynch callback-based API, there can be a equivalent threaded blocking function
 - Execution within the thread is driven by callback



IThread – differences from other thread APIs

- Cooperative (non-preemptive)
- Cancellation
 - IThread object may need to be killed synchronously
 - “Kill” may happen during any blocking call
 - Code in the thread must be ready to be canceled
 - Main problem is freeing allocated resources
 - Stack-based pointers are problematic



Resource Pooling

- Each IThread has a resource pool
 - Inherits from IRscPool
- Resources allocated from (or registered with) the pool will be automatically freed
 - IThread_Malloc/Free : comes from pool
 - IThread_HoldRsc: holds an interface pointer
- Eliminates most cleanup hassles associated with canceling threads
 - And can simplify lots of other allocation issues



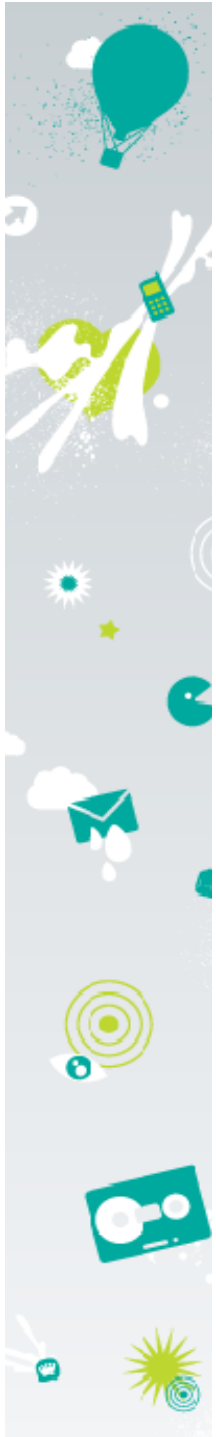
Using IThread

- Create IThread via ISHELL_CreateInstance()
- Start the thread
 - Specify stack size, start function, etc.
- When running in an IThread...
 - Allocate or group resources using IRscPool methods
 - Call supplied IThread-based blocking calls
 - Use examples provided in SDK under src/thrdutil
 - Roll your own
 - These functions accept an IThread* as a context
 - Useful as resource pool
 - Distinguishes them from “truly” blocking calls



IThread blocking calls

- Each blocking call does the following:
 - Obtain an AEECallback that will resume the thread
 - ITHREAD_GetResumeCBK() provides this
 - Request notification from some callback-based API
 - Supplying the IThread's resume callback
 - Suspend the thread
 - This transitions back to the main stack from the IThread's stack
 - Back in the main stack, we return control to the BREW event loop
- When the event fires:
 - The AEECallback is called from the BREW event loop
 - When called, it switches back to the IThread's stack and resumes execution where it was suspended



THANK YOU