

# Porting Applications to BREW®

Ray Rischpater, Chief Architect

*Rocket Mobile, Inc.*



# Our Agenda

- **Types of ports**
- **Common challenges**
- **Steps in porting**
- **Patterns in porting**
- **Making it work**

# My Background

- Writing for BREW since '01
- Ported > 250 kloc to BREW
- Participated in the launch of several commercial & embedded applications
- Author, Software Development for the QUALCOMM BREW Platform available in English and Korean



# Types of Ports

- **Porting C From Similar Platforms**
  - Embedded & handheld computing devices
- **Porting C from Dissimilar Platforms**
  - Libraries or applications from the desktop
  - Porting C++ from platforms with different frameworks
- **Porting from Other Platforms**
  - Bringing Java or .NET applications to BREW

# Porting Challenges

- **May be less risky than implementing from scratch**
- **But porting still has its risks**
  - Resource constraints
  - Execution model constraints
  - User interface constraints

# Porting Challenges

- **Other Platforms**

- Storage & Heap in Gigabytes
- Thousands of MIPS
- Multithreaded applications
- Synchronous
- Preemptive

- **BREW Applications**

- Storage & Heap in Megabytes
- Hundreds of MIPS
- Single (or few) threads in applications
- Asynchronous
- Cooperative

# Porting Challenges

- **Port only key algorithms**
- **Refactor code to meet constraints**
  - Examine & refactor storage use
  - Revise execution model
  - Leverage BREW APIs wherever possible
  - Leverage BREW Interfaces wherever possible

# Porting Challenges

- **Other Platforms**

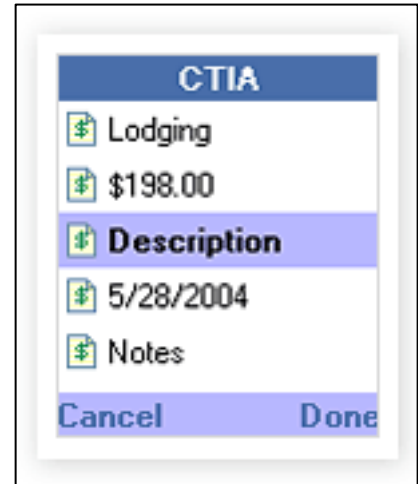
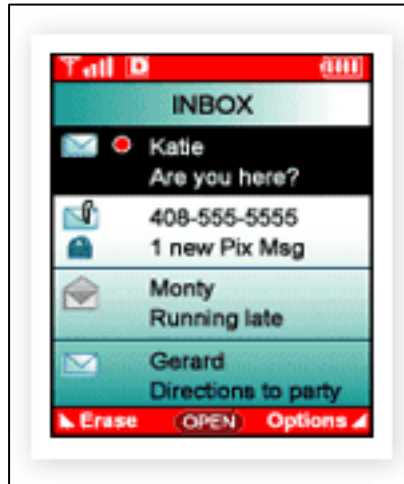
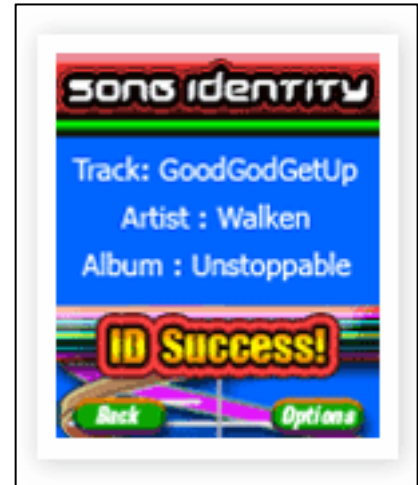
- Static content
- Big screen
- Rich interfaces
- Robust input mechanisms
- Data input is easy
- Users are stationary

- **BREW Applications**

- Dynamic content
- Small screen
- Constrained interfaces
- Limited input mechanisms
- Data input is hard
- Users are in motion

# Porting Challenges

- Your user interface must be concise
- Emphasize relevant data
- Simplify data input



# Steps in Porting

## Seven Steps for a Successful Port

- 1. Identify the object of the port**
- 2. Abstract platform dependencies**
- 3. Build a test harness**
- 4. Create a reference port**
- 5. Port platform dependencies to BREW**
- 6. Evaluate the port in your harness**
- 7. Create your application**

# Identifying the Object of the Port

- **Establish the business case**
- **Decide what to port**
  - Algorithms
  - Data structures
  - Business logic
  - Web services logic
- **Decide how to package your port**
  - Extensions are good containers for ported code

# Identifying the Object of the Port

- **Decide what not to port**
  - User interface
  - Redundant code (know your BREW APIs!)

# Abstracting Platform Dependencies

- Identify platform-specific interfaces
- Look for standard C library functions

**“Wrap using macros or functions...**

**...Don't just replace them!”**

# Abstracting Platform Dependencies

- **Identify run-time constraints**
  - Memory
  - Threading
  - Synchronous interfaces
- **Identify static constraints**
  - Endian issues
  - Structure packing
  - Word alignment
  - Code footprint

# Building a Test Harness

- **Exercise key porting layer APIs**
- **Exercise key application APIs**
- **Automate as much testing as possible**
- **Benchmark memory and performance of key APIs**

## Create a Reference Port

- **Port your code to a known platform**
- **Port dependencies to this platform**
- **Debug on your reference platform**

**“Debugging on the reference platform is easy...**

**...Debugging on the handset is harder.”**

# Port the Platform Dependencies to BREW

- **Establish your release environment**
  - Microsoft Visual Studio Projects & Solution
  - Make/gmake files for ARMCC or GCC
- **Implement the porting layer for BREW**
  - Leverage BREW interfaces wherever possible

# Evaluate the Port in Your Harness

- **Unit test in your harness**
  - Test on the emulator
  - Test on the handset
- **Monitor and review**
  - Memory use (heap & storage)
  - Application performance

# Creating your Application

- **Integrate your code and porting layer**
- **Create your user interface**
- **Validate your application**
- **Certify your application**
- **Distribute your application**

# Creating your Application

- **First you port to a platform-neutral configuration**
- **Next you port to your reference platform**
- **Finally you port to the BREW platform**
  - Simulator first
  - BREW-enabled handsets next

# Patterns in Porting

- **Standard library use**
- **Global & static variables**
- **Synchronous interfaces**
- **Multiple threads**
- **Packaging your port**

# Standard Library Use

- Use BREW's helper functions when you can
- Use macros:

```
#define strcmp(s1,s2) STRCMP(s1,s2)
```

- Or use functions:

```
__inline int strcmp(const char *s1, const char *s2) {  
    return STRCMP( s1, s2);  
}
```

# Global & Static Variables

- Place in application or extension context

```
typedef struct SMyExtension
{
    // Other stuff first, then globals
    int gMagicNumber;
} SMyExtension;
```

- Rewrite code to pass application or extension context for access to globals
- For applications, **GETAPPINSTANCE()** to access globals is an option

# Synchronous Interfaces

- **Refactor your code to work asynchronously**
  - How you do this is code-dependent
  - Break large computations into smaller functional chunks chained by asynchronous callbacks
- **Consider using IThread\***
  - Makes it easier to refactor code to behave asynchronously and yield to “main” application thread.
- **Use BREW’s callback mechanisms**

```
CALLBACK_Init  
CALLBACK_Cancel  
ISHELL_Resume
```

\* iThread is available on BREW Client 3.1 and newer devices

# Multiple Threads

- **Refactor your code to work asynchronously**
  - Model each thread as a state machine
  - Break large computations into smaller functional chunks chained by asynchronous callbacks
- **Use BREW's callback mechanisms**
- **Don't be tempted by IThread**
  - Cooperative, not preemptive

# Packaging Your Port

- **Wrapping your code as an extension**
  - Best when it's of use to multiple applications
  - Look to implement an existing interface
    - IMedia for media decoders
    - IHash for one-way hash algorithms
    - ICipher for cryptographic algorithms
- **Wrapping your code as a library**
  - Can be done in conjunction with an extension
  - Helps keep private things private
  - Can wrap your porting layer as a library too

# Making It Work

- **Heed compiler warnings**
  - Use the highest level of warnings available from the tool chain
  - Treat all warnings as errors
- **Leverage type checking**
  - Use const where appropriate
  - Avoid defeating the compiler's type checks

# Making It Work

- **Review data structures**
  - Look at data types and structure packing
  - Check your dependency on byte order
  
- **Review memory use**
  - Stack overflows
  - Misaligned data access
  - Memory leaks
  - Relying on the read/write segment (you don't have one!)
  - Consider a separate memory allocator

# Making It Work

- **Managing the porting effort**
  - Plan for iterative development
  - Keep iterations small
  - Use schedule checkpoints to provide proof of progress to stakeholders
  - Make time for prototyping & benchmarking
  - Invest in the reference port and test harness
  - Test early and test often in the simulator and on real handsets

## Don't Forget...

- **Interfaces may differ**
  - Some handsets don't have some BREW APIs
  - Handsets may have different capabilities (display, performance, heap, media decoders)
- **No read/write data segment**
- **No standard C library**
- **No blocking interfaces**
- **Byte order**
- **Structure packing**
- **Double-word aligned memory access**
- **They call it the BREW Simulator for a reason**