

Game mobility portability

Writing code that is easily portable across
and operating systems

by

Guido Henkel, CEO

ghenkel@g3studios.com

G3 Studios - <http://www.g3studios.com>

Introduction

Mobile developers are faced with a variety of different platforms, operating systems, languages and technical specifications when creating games

Being able to deliver and adapt applications quickly to all existing platforms is vital in this market

Portable code is the key to fast and problem-free application turn-around

What is portable code?

- Portable code is “good code” – well designed, well structured, well documented and well implemented
- Portable code makes it easy and quick to understand its functionality
- Portable code can easily be adapted to other environments

Portable code does not necessarily
eradicate the need for code
changes. Its purpose is to
minimize them!

Think ahead

- No mobile game exists in a vacuum and by nature has to serve a large number of environments
- Do not limit your thinking to *your* platform of choice
- Keep all potential environments in mind
- Keep all potential requirements in mind

Think ahead

- Chances are, your code is not for your eyes only
- Comment your code to make future changes and ports easy and pain-free
- Comment your code, because it's good practice!

Take pride in your code

Unlike graphic artists and musicians, programmers often do not take pride in their actual code. The result are shoddy implementations, dirty tricks, illegible code, unintelligible implementations that hide bugs, and code that is simply not up to snuff.

It is part of the reason why there are so many buggy games on store shelves!

The quality of your source code is just as important as your executable code!

The kinds of portability

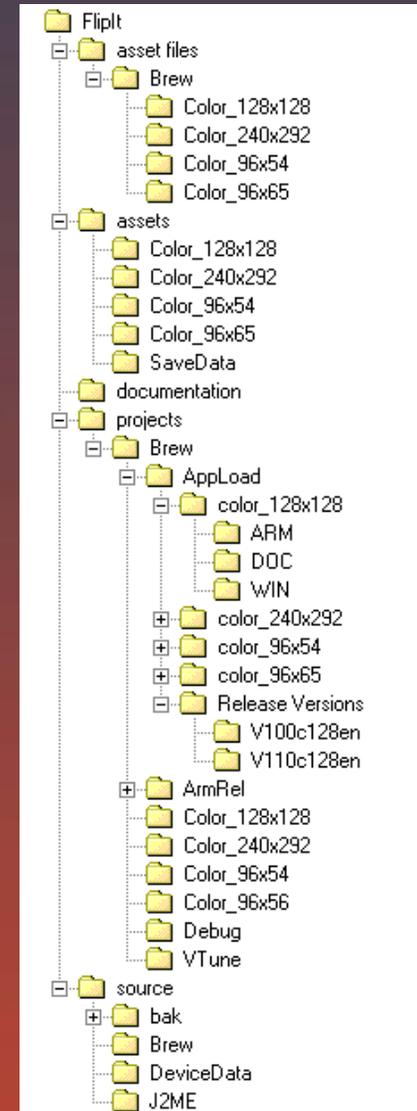
- Across platforms (Brew, Symbian, Windows CE)
 - Needs to abstract platform dependencies, such as File IO, User Interface, Bootstrap, etc.
- Across languages (C, C++, J2ME)
 - Needs to assimilate language features
- Across handsets (t720, vx6000,...)
 - Needs to abstract hardware differences, such as display resolutions, audio capabilities, firmware bugs, etc.

Project organisation

Directory structure

- Group directory
 - Platform/language directory
 - Handset-specific directory

The deeper the directory structure goes, the more specialized the items within get



Directory structure

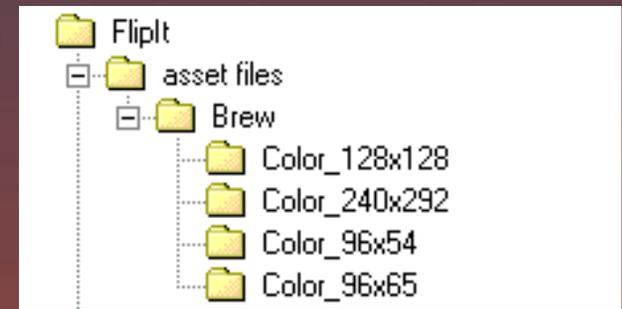
The "Asset Files" directory contains the discrete asset data, such as maps, images, audio files, data files, scripts, etc.

Data become more specialized the deeper we get in the hierarchy

```
asset files/brew/color_128x128
    contains data only relevant to
    that particular environment
```

```
asset files/brew contains data for
    all Brew handsets
```

```
asset files contains data that are
    relevant to all environments
```

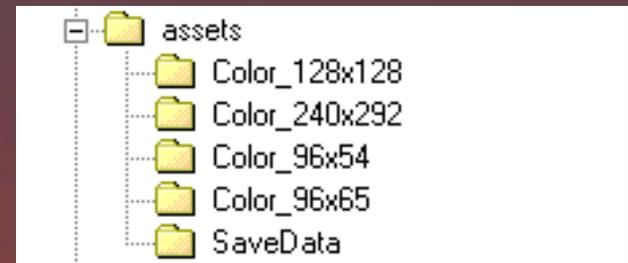


Directory structure

The "Assets" directory contains the prepared asset data, such as maps, images, audio files, data files, scripts, etc. in the form of BAR files, data library files, or any format of choice

Data in these directories are typically tool-generated and represent the data in the form the application can use

It also contains the untouched, original save game files, etc.



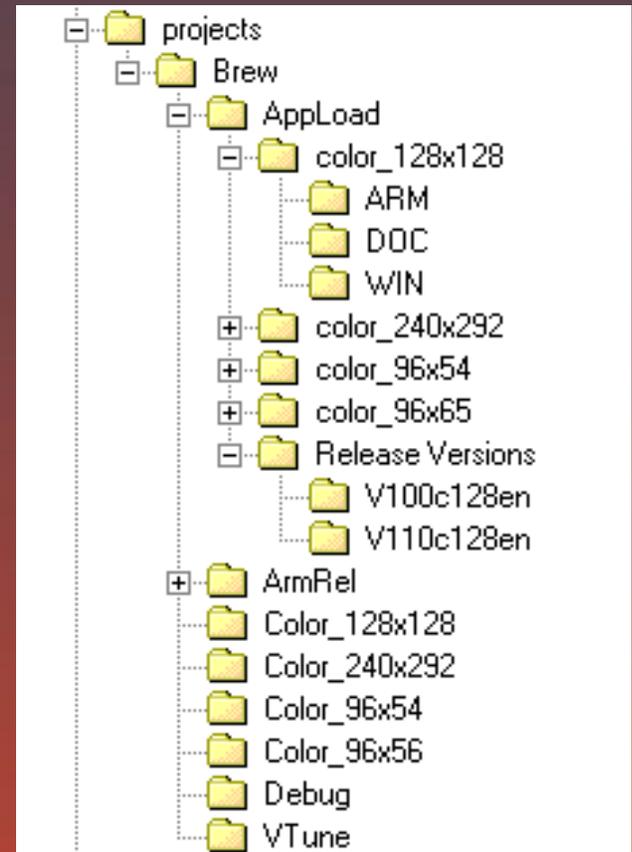
Directory structure

The "Projects" directory contains Visual Studio solution files, EVC projects, make-files, as well as all compiled program code, complete builds and archived versions of previous production builds

"AppLoad" and "ArmRel" are used to separate intermediate builds from final builds

```
.../brew/appload/color_128x128  
contains the complete  
submission/release package
```

Game Developers Conference 2004
for that particular
application



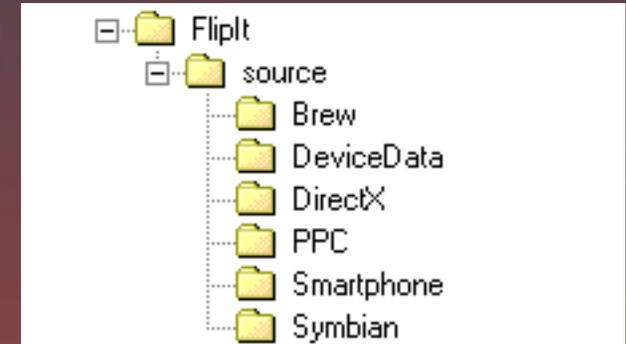
Directory structure

The "Source" directory contains all the source code for the application

source/brew contains all the platform dependent code parts for the Brew builds

source/ppc contains the platform dependent code for the Pocket PC version, and so forth...

source/devicedata contains the necessary device-specific source code for particular environments



Directory structure

The key to a good project
directory structure

- Keep it extensible
- Keep it clear
- Keep it structured
- Keep out the clutter

With this sort of directory structure we have created an easy way to physically separate device-dependent code from general game code

In the project we now simply create wrappers to load the platform specific implementations during compile time

Create wrappers for both header and source code files

Examples for typical platform dependent implementations are application startup, memory manager, display handling, input handling, file IO, and various utility functions, such as sprite blitting, debug

```
//  
// UTILITY.CPP  
//  
// Author: Henkel  
// Date: 04/17/03  
//  
// This file contains various wrappers and  
// utility functions for the various platforms  
//  
  
#ifdef BREW_VERSION  
    #include "BREW/Utility.cpp"  
#elif PPC_VERSION  
    #include "PPC/Utility.cpp"  
#elif SYMBIAN_VERSION  
    #include "Symbian/Utility.cpp"  
#elif SMARTPHONE_VERSION  
    #include "Smartphone/Utility.cpp"  
#endif
```

Designing the framework

To make all this work together
effortlessly we need to create a
framework that accommodates this data
design in a way that is transparent
and unobtrusive

The way to do this is by using "hooks"

Create an event driven application
design using a finite state machine
and use these hooks to drive the core
of the application

```

static boolean
EventHandler( . . . Parameters . . . )
{
    switch ( eventCode )
    {
        case EVT_APP_START:
            ClearScreen( curApp )
            MemInit( curApp );      // Initialize memory manager
            GameStart( curApp );
            GamePostfix( curApp );
            break;

        case EVT_APP_STOP:
            GameEnd( curApp );
            MemExit( curApp );      // Exit the memory manager
            break;

        case EVT_APP_EVENT:        // Special treatment may be
            GameEvent( curApp );    // required here depending on
            break;                  // the actual platform

        case EVT_APP_SUSPEND:
            SuspendGame( curApp );
            break;

        case EVT_APP_RESUME:
            ResumeGame( curApp );
            break;

        case EVT_KEY_PRESS:
            keyPressed( curApp, parameter );
            break;

        case EVT_KEY_RELEASE:
            keyReleased( curApp, parameter );
            break;
    }
}

```

These are the hooks
to platform
independent code
that drive the game
engine

```
static boolean  
EventHandler( . . . Parameters . . . )  
{  
    switch ( eventCode )  
    {  
        case EVT_APP_START:  
            ClearScreen( curApp )  
            MemInit( curApp );           // Initialize memory manager  
            GameStart( curApp );  
            GamePostfix( curApp );  
            break;  
  
        case EVT_APP_STOP:  
            GameEnd( curApp );  
            MemExit( curApp );          // Exit the memory manager  
            break;  
  
        case EVT_APP_EVENT:             // Special treatment may be  
            GameEvent( curApp );        // required here depending on  
            break;                       // the actual platform  
  
        case EVT_APP_SUSPEND:  
            SuspendGame( curApp );  
            break;  
  
        case EVT_APP_RESUME:  
            ResumeGame( curApp );  
            break;  
  
        case EVT_KEY_PRESS:  
            keyPressed( curApp, parameter );  
            break;  
  
        case EVT_KEY_RELEASE:  
            keyReleased( curApp, parameter );  
            break;  
    }  
}
```

These are the hooks
to platform
independent code
that drive the
game engine

With this interface
we abstract the
core platform
dependencies

This core can be
created for
virtually every
environment today

And it can easily
be extended to
add
functionality,
such as stylus
input, etc.

```
static boolean  
EventHandler( . . . Parameters . . . )  
{  
    switch ( eventCode )  
    {  
        case EVT_APP_START:  
            ClearScreen( curApp )  
            MemInit( curApp );      // Initialize memory manager  
            GameStart( curApp );  
            GamePostfix( curApp );  
            break;  
  
        case EVT_APP_STOP:  
            GameEnd( curApp );  
            MemExit( curApp );     // Exit the memory manager  
            break;  
  
        case EVT_APP_EVENT:        // Special treatment may be  
            GameEvent( curApp ); // required here depending on  
            break;                 // the actual platform  
  
        case EVT_APP_SUSPEND:  
            SuspendGame( curApp );  
            break;  
  
        case EVT_APP_RESUME:  
            ResumeGame( curApp );  
            break;  
  
        case EVT_KEY_PRESS:  
            keyPressed( curApp, parameter );  
            break;  
  
        case EVT_KEY_RELEASE:  
            keyReleased( curApp, parameter );  
            break;  
    }  
}
```

Think about it

The key to successfully creating portable code is to look for and find a common denominator

This applies to both, platforms and operating systems, as well as language features

The more you know about your targets, the easier you will find it to set up these common denominators

Creating a common base

J2ME doesn't have a pre-processor,
so it is advisable to use C/C++
greater flexibility to
assimilate J2ME behavior

Since Brew doesn't have a global variable
name space we can create a macro to
"simulate" this name space if we have to.
Use this feature judiciously, however!

```
#define ScreenWidth (Applet->ScreenWidth)
```

Creating a common base

J2ME has hard-wired `keyPressed()`,
`keyReleased()`, and `paint()`
interfaces for certain objects

You will make your life much easier
if you stick to that naming
convention in C/C++ as well, even
if it breaks with your own naming
convention

Creating a common base

J2ME does not support operator
overloading or templates

While these are powerful and tempting
features for every C++ programmer
to use, you may want to stay away
from them in preference of higher
portability

Creating a common base

J2ME does not have pointers, only
arrays

Given today's CPU architecture,
pointers are also no longer
necessarily faster - especially on ARM
processors in Thumb mode

```
for (i=0; i<100; i++)  
{  
    *dest++ = *src++;  
}
```

The problem with this loop is that it will have to be re-written for the J2ME version and in terms of
Game Developers Conference 2004
performance, that the processor has to increment the
iterator *i* and the two pointers during every loop

Creating a common base

```
for (i=0; i<100; i++)  
{  
    dest[i] = src[i];  
}
```

This version is portable AND more powerful on today's CPU's because only the iterator needs to be incremented, and because existing opcodes allow to pre-multiply the index as an addressing mode, making array access in a single instruction possible even with variable array element sizes.

Creating a common base

Brew does not support exception
handling

Make sure you use it judiciously in
your other versions and try to
instead rely on other means of
error detection

Creating a common base

Brew does not support static class variables or global-scope variables

If you have to create persistent class variables, place them in the global applet structure, and maybe use a special naming convention to indicate their class-ownership

Alternatively, create a "friend" super-class that holds only those elements that need to remain static. Keep in mind however that in Java, each class has a memory overhead that can be severe in tight environments

Creating a common base

Stay away from dirty programming
tricks!

Using J2ME's out-of-bounds exception
handling to detect array sizes is
very bad practice and barely
portable



General tips and good habits

Good habits

Many programmers seem reluctant to make full use of the tools that modern compilers put at their fingertips. There are a few things that intrinsically make your life as a programmer easier, and help making your code more stable and safer

One should think that applying some of these methods is common sense, but still they are applied far too infrequently in my experience. Here are some suggestions...

Good habits

Source code is cheap, so use it!

Good programming style occasionally requires you to write more source code than otherwise, but the payoff is usually enormous

There is no room for laziness in programming!

Good habits

- Only one command per line
- Properly indent and bracket your code
- Work with a sensible style guide and stick to it
- Apply some common sense and don't try to be a "cool" coder. Ultimately you're making your own life unnecessarily harder
- Never add a call to a particular OS-specific function to your game engine code - write a custom version instead, that simply forwards the call to the OS if need be

Lose the #define macros

The single-most abused feature of C and C++ is the #define pre-processor command

Use const to create constant values

- const has scope and a name space. It is type-safe and allows the compiler to optimize your code much better
- const can quickly be turned into a variable without much code change. Great for experimentation before locking down final values
- Despite Brew's inability to create global variables, const can be used to create

Lose the #define macros

There are valid uses for #define.

Make sure you use it only in such cases, and keep in mind that even simple macros like the one below can turn into deadly traps

```
#define max( a, b )    ((a) > (b) ? (a) : (b))
```

Lose the #define macros

```
#define max( a, b )    ((a) > (b) ? (a) : (b))
```

Now, imagine this..

```
int a = 5, b = 0;
```

```
max( ++a, b );
```

```
max( ++a, b+10 );
```

The results of this little exercise are fatal

Lose the #define macros

A much better way to solve this problem is to use an inline function, such as this

```
inline int max( int a, int b) { return a > b ? a : b; }
```

This may not be perfect, but since we can't use templates, it's certainly the next best thing

Say "No" to Hungarian Notation

- It is unwieldy and error-prone
- It is counterproductive and hard to maintain
- It reveals implementation specifics as part of a class interface

Finite State Machines

- Use of Finite State Machines can greatly increase the portability of a code base
 - Inherently iterative
 - Easy to understand and extend
 - Non-blocking

Heed your compiler warnings

Always set your compiler to highest warning levels –
Level 4 in Visual Studio

Make sure your project compiles with 0 warnings

Do not use #pragma to turn off warnings as it is
compiler-specific and may silently turn off vital
warnings in another environment

If a warning is unavoidable, comment the line of code
in question, explaining why the warning cannot be
circumvented

Warnings are usually a sign of sloppy laziness, which
has no place in professional game development

Warnings often reveal actual bugs that are discovered
as you inspect your code for the cause of the

warning

Putting together a game

The framework

- Write an application framework that is reusable, portable and extensible
- Include implementations for application startup, memory manager, display handling, input handling, file IO, and various utility functions, such as sprite blitting, debug logging, handset detection, etc.

The framework

- Memory manager

- Make sure to overload new and delete as well as new[] and delete[]

```
void *operator new( size_t size );  
void operator delete( void *ptr );  
void *operator new[]( size_t size );  
void operator delete[]( void *ptr );
```

- Add debugging tools if you wish, such as bounds-checking, leak detection, and other statistics
- Having good new and delete implementations I found that malloc() became virtually redundant

The framework

- Display handling
 - Create an abstract interface - I am typically using a GAPI-style interface because it is small and very tight, providing only the essentials
 - If you work with direct framebuffer access, be prepared to write your routines for RGB444, RGB555 and RGB565 color triplets in 16-bit modes. You may not have to implement them immediately - just keep in mind that there are devices with these configurations that you may run into some time down the line

The framework

- File IO
 - Despite providing basic file IO such as `open()`, `close()`, `read()` and `write()` I found it much more useful in mobile applications to provide specific loading routines
 - `LoadImage()`
 - `LoadData()`
 - `ReadFile()`, `WriteFile()`
 - This usually removes repetitive coding and error-checking, and provides you with tighter calls in your game engine that are platform-independent

The framework

- File IO

- Never access files directly by name. Use a constant instead. That way you keep it portable between platforms where files are loaded by name and those where they are loaded by ID from a resource file

- J2ME

```
static final char FileName[] = "datafile.dat";
```

- Brew

```
const int FileName          = DAT_DATAFILE;
```

The implementation of the file IO functions helps to hide the difference in data types used for loading, as they provide the respective prototypes for the game engine and the resulting call remains the same

```
LoadDataFile( FileName );
```

The framework

- Sprite blitting

- Blitting functions should be basic and platform independent

```
void Blit( applet *curApp, G3word xPos, G3word yPos, G3word width,  
          G3word height, void *data, G3word xSrc, G3word ySrc, G3RasterOp  
          mode )
```

- Provide image clipping
- Especially in mobile development U/V-mapping of sprites in a large source-bitmap is advisable to save memory
- Make sure to always, always clip your sprites against the screen dimensions

Example of a high-level blitting implementation in Brew

```
void
Blit( applet *curApp, G3word xPos, G3word yPos, G3word width, G3word height, void *data,
      G3word xSrc, G3word ySrc, G3RasterOp mode )
{
  if ( xPos < curApp->ClipRect.left )
  {
    width += xPos - curApp->ClipRect.left;
    xSrc   -= xPos - curApp->ClipRect.left;
    xPos   = curApp->ClipRect.left;
  }
  else if ( ( xPos + width ) > ( curApp->ClipRect.left + curApp->ClipRect.right ) )
  {
    width = width - ( xPos + width - ( curApp->ClipRect.left + curApp->ClipRect.right ) );
  }

  if ( yPos < curApp->ClipRect.top )
  {
    height += yPos - curApp->ClipRect.top;
    ySrc    -= yPos - curApp->ClipRect.top;
    yPos    = curApp->ClipRect.top;
  }
  else if ( ( yPos + height ) > ( curApp->ClipRect.top + curApp->ClipRect.bottom ) )
  {
    height = height - ( yPos + height - ( curApp->ClipRect.top + curApp->ClipRect.bottom ) );
  }

  if ( width > 0 && height > 0 )
  {
    IDISPLAY_BitBlt( curApp->a.m_pIDisplay, xPos, yPos, width, height, data, xSrc, ySrc, mode );
  }
}
```

The framework

- Debug Logging
 - Create a basic set of functions to log, protocol or dump data into file
 - Add functionality to log, protocol or dump data to the debugger output window
 - Debug functions can be performance drains, so make sure you wrap them so that the calls can be easily removed in your release builds

The framework

- Stdlib functions
 - Even though most functionality is available across platforms, its performance varies dramatically
 - Brew does not allow support stdlib functions and instead utilizes substitute functions
 - When compared to their library counterparts, memcpy() for example can be implemented twice as fast in a simple C-loop in Brew, while it is virtually unbeatable in Windows CE
 - Create an abstraction layer to isolate all calls to external library functions for flexibility. That way you can simply redirect to the stdlib function or write your own implementation when necessary

The framework

- Custom UI

- Create your own UI class so you won't have to rely on UI features and implementations on actual handsets
- Create functionality for text display, menuing using your own custom fonts
 - Eliminates problems stemming from firmware bugs
 - Eliminates problems from non-existing UI features across platforms, such as menus with icon, etc.
 - Creates a consistent look and handling across handsets and platforms
 - It is expandable and you have full control over it

The framework

- Handset detection
 - Create a module that is dedicated to handset detection based on platform Ids, OEM Strings, or the like
 - Make sure to also read-out the OS version
 - Retrieve handset specific information in this module and make it available to the application

Example of handset detection in Brew

```
void
DetectHandset( applet *curApp )
{
    handsetID      deviceID;
    AEEDeviceInfo  ddi = { 0 };
    const platformID *ptr;

    deviceID      = INVALID_DEVICE;           // Retrieve device information
    ddi.wStructSize = sizeof( ddi );         // from the handset
    ISHELL_GetDeviceInfo( curApp->a.m_pIShell, &ddi );
    ScreenWidth   = ddi.cxScreen;
    ScreenHeight  = ddi.cyScreen;
    Language      = ddi.dwLang;

    if ( 0 != ddi.dwPlatformID )             // If the handset returns an ID
    {                                         // things are fairly easy.
        ptr = PlatformIDTable;
        while ( NULL != ptr->OEMID )
        {
            if ( ptr->OEMID == ddi.dwPlatformID )
            {
                deviceID = (handsetID) ptr->DeviceID;
                break;
            }
            ptr++;
        }
    }

    if ( INVALID_DEVICE == deviceID )       // If it does not return a unique ID
    {                                         // we will have to use other means...
        ptr = PlatformIDTable;
        while ( NULL != ptr->OEMID )
        {
            if ( ptr->Width == ScreenWidth && ptr->Height == ScreenHeight )
            {
                deviceID = (handsetID) ptr->DeviceID;
                break;
            }
            ptr++;
        }
    }
    curApp->HandsetID = deviceID;
}
```

Example of handset detection in Windows CE, Pocket PC and Smartphone

```
void
DetectHandset( applet *curApp )
{
    G3ustring string[64] = { 0 };           // Unicode string required!

    if ( 0 != SystemParametersInfo( SPI_GETOEMINFO, sizeof( string ), &string[0], 0 ) )
    {
        ptr = PlatformIDTable;
        while ( NULL != ptr->DeviceID )
        {
            if ( 0 == wcsicmp( ptr->OEMString, string ) )
            {
                deviceID = (handsetID) ptr->DeviceID;
                break;
            }
            ptr++;
        }
    }
}
```

- Use the information from the handset detection to drive your application

I am typically using a structure with all the relevant variables for a number of key handsets

```
typedef struct
{
    G3word16  SoundIdleDelay;
    G3bool    DeviceAllowsSoundPause;
    G3word16  ScreenQuadX;
    G3word16  ScreenQuadY;
    G3word16  ScreenLogoX;
    G3word16  ScreenLogoY;
} deviceConfig;

static const deviceConfig DeviceConfigs[] =
{
    { 1,  TRUE, 64, 65, 24, 30 };           // Motorola t720
    { 40, TRUE, 60, 57, 22, 24 };         // LGE VX6000
    NULL
};
```

Frequently, certain handsets can safely use the exact same configurations, so we then identify the key handset from the obtained handset information

```
switch ( curApp->HandsetID )
{
    case AUDIOVOX_CDM8400:
    case AUDIOVOX_CDM8410:
    case AUDIOVOX_CDM8600:
        curApp->KeyDeviceID = USE_AUDIOVOX_CDM8600;
        break;
}
```

Now all we have to do is, create a pointer to the correct device configuration data for the key handset that we determined

```
curApp->DevCfg = &DeviceConfigs[ curApp->KeyDeviceID ];
```

We now have easy access to all sorts of variables that can be hand-optimized for particular handsets. These can contain delay values, flags to avoid certain function calls and circumvent firmware bugs, placement coordinates, filenames to allow for swapping out images or sounds or anything else you can think of

```
void  
SoundSuspend( applet *curApp )  
{  
    if ( NULL != curApp->Sound )  
    {  
        if ( TRUE == curApp->DevCfg->DeviceAllowsSoundPause )  
        {  
            ISOUNDPLAYER_Pause( curApp->Sound );  
        }  
        else  
        {  
            ISOUNDPLAYER_Stop( curApp->Sound );  
        }  
    }  
}
```

This data-driven architecture minimizes code-changes especially for handset ports and substitutes them with easy to edit data structures in one place

Accommodating screen resolutions

Screen resolutions on mobile phones can be a nightmare and range from anywhere between 96x54 to 240x320 pixels and above

Using the device configuration table described before will make dealing with these resolutions much easier, especially when combined with a little trick

Splicing

Splicing takes different graphic elements and draws them in a way that creates a seamless image



Splicing

Through different positioning it allows us to create backgrounds in various sizes on the fly



FlipIt! – The example



All 36 Brew handset versions of "FlipIt!" are using the exact same code base and all changes to resolutions and screen layouts were achieved through the data-driven architecture and

Some final thoughts

- Always apply common sense and try to think ahead
- Always document your code and use variable and function names that are sensible and self-explanatory
- Use Assertions liberally. They can not only detect errors in your program logic, but also reveal porting issues between handsets and platforms, such as incorrect endianness, alignment issues, corrupt data, etc.
- Consciously look for and isolate platform and device dependencies, even if it means some extra work

Some final thoughts

- If your data structures change while you're developing a later handset, make sure to make the correct adjustments in ALL your previous configurations. Never leave your code open-ended or prone to bugs
- While you are doing the initial handset implementations, make as many as possible, even if they're not immediately required. It is much easier to make handset ports while the code is still fresh in your mind than having to go back again later
- Once you are done with the application, lock down your game engine code so it will not be broken by future handset ports. You can use revision control systems to do that or simply

Some final thoughts

- Properly document the steps necessary to build the application
- Document the steps that are necessary to implement a new handset version
- Never #ifdef your way through code. It is the least portable solution - a hack in essence - and typically only introduces errors in versions and builds that were working fine previously
- In general, try to touch as little code as possible when doing ports. The less code you change, the smaller the chance of introducing new bugs!

Book recommendations

- Writing Solid Code, by Steve Maguire
Microsoft Press, ISBN 1556155514
- Code complete, by Steve McConnell
Microsoft Press, ISBN 1556154844
- Code Reading, by Diomidis Spinellis
Addison-Wesley, ISBN 0201799405
- Effective C++, by Scott Meyers
Addison-Wesley, ISBN 0201924889