Tapwave® High Performance Games

Version 1.1a

Copyright

1.Overview

This guide provides an overview and reference for creating high performance games on the Tapwave platform. It is geared towards developers who are creating full-screen games and who don't need the standard Palm OS UI widgets.

The Tapwave Native Application (TNA) framework provides an environment for creating high performance, ARM-native, applications for Tapwave devices.

A Tapwave Native Application consists of two parts:

- A very small 68k stub that launches the ARM-native code on the device.
- The main application logic compiled into ARM-native code.

The Tapwave Native Application framework provides the following capabilities to enable the development of high-performance, interactive gaming applications:

- The fastest possible code execution.
- Source level debugging via the Palm OS Debugger.
- Extensive runtime and library support, including true global variables, runtime relocation, standard C/C++ library, floating point, math library, socket, and zlib.
- Binary compatibility with future Tapwave devices.

The Tapwave Native Application framework requires you to adhere to these parameter:

- Do not use any standard Palm OS UI widgets from ARM code.
- Do not call any Palm OS functions, except those defined in `TwGlue.h`.

2.Tapwave Native Application Framework

The Tapwave Native Application framework provides the easiest way to create a high-performance game on the Tapwave platform. By using this framework, an application can register to receive system events and can respond appropriately by processing input and drawing to the screen. The framework consists of the following APIs:

TwAppStart

Purpose	Initialize the Tapwave Native Application framework	
Prototype	`Err TwAppStart (Boolean (*handler)(EventType* event));`	
Parameters	`[in] handler`	The application event handler
Result	`ErrNone`	
Pre-Conditions	An application should not call any Palm OS user interface APIs before calling this function.	
Comments	This function creates a blank window, installs the event handler for the window, and makes the window the active window. The application can access the window by using `WinGetActiveWindow()` inside the event handler. Once the framework is started, the event handler must be ready to receive events from the system. If an application needs a specific display layout, it should perform the appropriate setup in advance to avoid unnecessarily redrawing the screen. **WARNING:** The event handler definition must include a special compiler attribute `SYSTEM_CALLBACK`, which is defined in `<TwDefs.h>`. This attribute specifies that the function needs a special prolog to restore the global pointer (GP). For Metrowerks CodeWarrior 9.2, the attribute is defined as `__declspec(pace_native_callback)`. If the compiler does not support this feature, you must write the proper thunk to handle this. The implementation details of this feature are beyond the scope of this document.	
Header	`TwRuntime.h (included by Tapwave.h)`	

Sample	`// using full screen landscape mode`
	`Int32 timeout = 0;`
	`SysSetOrientation(sysOrientationLandscape);`
	`PINSetInputAreaState(pinInputAreaHide);`
	`StatHide();`
	`TwAppStart(&AppHandleEvent);`
	`TwAppRun(&timeout);`
	`TwAppStop();`

TwAppStop

Purpose	Tear down Tapwave Native Application framework
Prototype	`Err TwAppStop (void);`
Result	`ErrNone`
Post-Conditions	An application should not call any Palm OS user interface APIs after calling this function.
Header	`TwApp.h (included by Tapwave.h)`

Sample	`Int32 timeout = 0;`
	`TwAppStart(&AppHandleEvent);`
	`TwAppRun(&timeout);`
	`TwAppStop();`

TwAppRun

Purpose	Run Tapwave Native Application framework	
Prototype	`Err TwAppRun (Int32 * timeout);`	
Parameters	`[in] timeout`	The pointer to a 32-bit integer that controls the event loop speed. The timeout is in milliseconds.
Result	`ErrNone`	
Pre-Conditions	An application should only call this API after `TwAppStart` and before `TwAppStop`.	
Comments	This function runs the standard Palm OS event loop. Alternately, you could write your own event loop, although this one is provided to help simplify your code. This function returns upon the first unhandled `appStopEvent`. **WARNING:** Although the timeout value is specified in milliseconds, the actual response rate is in 10ms increments due to various hardware and software limitations. The same rule also applies to `SysTaskDelay`.	
Header	`TwApp.h (included by Tapwave.h)`	

Sample	`Int32 timeout = 0;`
	`TwAppStart(&AppHandleEvent);`
	`TwAppRun(&timeout);`
	`TwAppStop();`	

The following illustrates a typical Tapwave Native Application event handler.

```
SYSTEM_CALLBACK Boolean
AppHandleEvent(EventType * event)
{
  switch (event->eType)
  {
```

```
  case winDisplayChangedEvent:
  {
    RectangleType bounds;
    // NOTE: update window bounds to match the new display bounds.
    WinGetBounds(WinGetDisplayWindow(), &bounds);
    WinSetBounds(WinGetActiveWindow(), &bounds);
    // TODO: update rest of application logic
    return true;
  }
  case winExitEvent:
    // NOTE: application should pause on this event,
    // such as stop background sound.  Control of the device
    // is about to switch to other code, and no events
    // will be received until the control is resumed.
    // WARNING: application has to return false here to give the
    // system a chance to fix the PINS state, or the application can
    // fix the PINS state by itself.
    return false;
  case winEnterEvent:
    // WARNING: application has to return false here to give the
    // system a chance to fix the PINS state, or the application can
    // fix the PINS state by itself.
    return false;
  case nilEvent:
    // NOTE: games should perform rendering on this event.
    // Timing-sensitive games should double-check
    // the wall clock time using TimGetTicks().
    return true;
  case frmUpdateEvent:
    // NOTE: system sends this event to force the application
    // to redraw itself.
    return true;
  case appStopEvent:
    // NOTE: save application state and get ready for quit.
    // If application refuses to quit, it should return true.
    return false;
  case keyDownEvent:
    // NOTE: this is a general key down event. Applications should
    // only return true if they handle the specific key in the event.
    return false;
  case keyUpEvent:
    // NOTE: this is a general key up event. Applications should
    // only return true if they handle the specific key in the event.
    return false;
  case penDownEvent:
    // NOTE: the application sees this event only if the pen is
    // down inside the application window, otherwise this pen is
    // handled by the operating system.
    return false;
```

```
  case penMoveEvent:
    // NOTE: application can use EvtGetPen() to poll the pen position
    // until the pen is up instead of waiting for penMoveEvent.
    return false;
  case penUpEvent:
    // NOTE: application sees this event only if it receives
    // a matching penDownEvent first. However, the application should
    // also guard against spurious wrong event dispatching.
    return false;
  default:
    // NOTE: TNA applications normally ignore other event types.
    return false;
  }
}
```

3.Tapwave Native Application APIs

Tapwave Native Applications have access to a subset of PalmOS 5.x APIs and most of the Tapwave APIs. These APIs are 32-bit native APIs, not PACE wrapper functions. They are all defined in the `TwGlue.h` header file.

In order to access these APIs, the header file must be compiled with `__PALMOS_ARMLET__` macro defined. The Metrowerks CodeWarrior 9.2 compiler does this automatically.

4.Creating a Tapwave Native Application for a Tapwave Device

To create a Tapwave Native Application, you need CodeWarrior 9.2 and the Tapwave SDK. If you don't have either, visit the Tapwave Developer Zone for more information.

The GameStarter sample code is a great place to start. This sample includes the following files:

Application.c	The 68k source file.
GameStarter.rcp	The application's resources. This example only includes the application's icons and a "ROM Incompatible" alert.
Application.rcp	This resource maps the entry point into the x86 code resource. This is only necessary for running/debugging your application on the Palm Simulator. The name of your DLL (minus the .dll extension) is included in this file. You'll need to change this name if you change the name of your DLL.
GameStarter.c	The source file for the ARM-native code.

Startup.cpp This file defines the real entry point into the ARM code. From here you can fine tune various hooks including relocation support and a floating point.

Application.c contains a small 68k stub whose only job is to launch the ARM code on the device or x86 code on the Palm Simulator. Below is a sample showing how to launch the ARM code resource:

```
UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
    UInt32 res = errNone;
    NativeFuncType* entry;

    if (cmd == sysAppLaunchCmdNormalLaunch) {
        if (!RomVersionCompatible(launchFlags)) {
            res = TwLoadModule(0, 0, 0, 1,
                               twLoadFlagTNA|twLoadFlagQuickRun,
&entry);
        }
    }

    return res;
}
```

GameStarter.c contains the main application event loop and the game engine. This example shows only the core code necessary to run properly:

```
SYSTEM_CALLBACK Boolean GameHandleEvent(EventType* eventP)
{
     Boolean         handled = false;
     RectangleType   bounds;

     switch (eventP->eType)
     {
          case winDisplayChangedEvent:
               WinGetBounds( WinGetDisplayWindow(), &bounds );
               WinSetBounds( WinGetActiveWindow(), &bounds );
               handled = true;
               break;

          case nilEvent:
          case frmUpdateEvent:
               // Put update code here
               handled = true;
               break;
     }
```

```
	return (handled);
}

UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
	Int32 timeOut = TimeUntillNextPeriod();

	if (cmd == sysAppLaunchCmdNormalLaunch)
	{
		// Get the screen to a state we want.
		SysSetOrientation(sysOrientationLandscape);
		PINSetInputAreaState(pinInputAreaHide);
		StatHide();

		// Set everything up.
		TwAppStart(&GameHandleEvent);

		// Run event loop.
		TwAppRun(&timeOut);

		// Tear everything down.
		TwAppStop();
	}

	return 0;
}
```

5.Tapwave Native Application Runtime

The Tapwave Native Application runtime provides simple and convenient APIs for managing Tapwave Native Application modules (or PACE Native Object modules). You should use these APIs from your 68k code to load your ARM code. The runtime consists of the following APIs:

TwLoadModule

Purpose	Load a PACE Native Object module	
Prototype	`Err TwLoadModule (UInt32 dbType, UInt32 dbCreator, UInt32 rsrcType, UInt32 rsrcID, UInt32 flags, NativeFuncType** entry);`	
Parameters	`[in] dbType`	Database type

	[in] dbCreator	Database creator
	[in] rsrcType	Resource type
	[in] rsrcID	Resource ID, which must be <= 0xFFFF
	[in] flags	Various flags that control the module loading twLoadFlagTNA load and initialize a TNA module twLoadFlagQuickRun load a PACE Native Object module, run it, and unload it
	[out] entry	Returns the entry point to the loaded PACE Native Object
Result	errNone sysErrParamErr memErrNotEnoughSpace memErrChunkNotLocked sysErrLibNotFound	

Comments	If both the `dbType` and `dbCreator` are zero, this function loads the module using `SysCurAppDatabase`. If `rsrcType` is zero, it is assumed to be 'ARMC' on ARM devices, and 'SIMC' on Palm OS Simulator. If the 'ARMC' code resource is missing, the runtime tries to load the 'ARMZ' code resource, which is assumed to be a gzipped version of the 'ARMC' resource. You can gzip your 'ARMC' resource and save about 50% of your storage space. The code in the resource is copied/expanded to the dynamic heap and relocation is performed. Larger code resources leave less memory available for the application's use. The runtime uses reference counting to manage loaded libraries. Multiple loads of the same library use the same entry point. The global state of a library is kept intact between multiple `PceNativeCall()` calls. If the debugger is present, this function also automatically notifies the debugger about the newly loaded module.
Header	`TwRuntime.h (included by Tapwave.h)`

Sample	`NativeFuncType* entry;`
	
	`TwLoadModule(0, 0, 0, 1,`
	`  twLoadFlagTNA|twLoadFlagQuickRun, &entry);`

TwUnloadModule

Purpose	Unload a PACE Native Object module	
Prototype	`Err TwUnloadModule (NativeFuncType* entry);`	
Parameters	`[in] entry`	PACE Native Object entry point, which must be previously returned by `TwLoadModule`.
Result	`errNone` `sysErrLibNotFound`	
Comments	The runtime uses reference counting to manage loaded modules. Applications should maintain balanced load and unload calls. If the debugger is present, this function also notifies the debugger regarding the unloaded module.	
Header	`TwRuntime.h (included by Tapwave.h)`	

Sample	`NativeFuncType* entry;`
	
	`TwLoadModule(0, 0, 0, 1, 0, &entry);`
	`PceNativeCall(entry, NULL);`
	`TwUnloadModule(entry);`	

TwFindModule

Purpose	Find PACE Native Object module for a given PC.	
Prototype	`Err TwFindModule (void* pc, TwModuleInfo* info, UInt32 size);`	
Parameters	`[in] pc`	Program counter to query.
	`[out] info`	Returns module info if any.
	`[in] size`	The `sizeof(TwModuleInfo)`.
Result	`errNone` `sysErrLibNotFound`	
Comments	This function only finds a module that was loaded using `TwLoadModule`. Applications should only use this API for debugging and profiling purpose. If you need to call `PceNativeCall`, then you must load the module using `TwLoadModule` first. If the size is smaller than `sizeof(TwModuleInfo)`, only partial data is returned. If the size is larger than `sizeof(TwModuleInfo)`, the extra space is filled with zeros.	
Header	`TwRuntime.h (included by Tapwave.h)`	

6.Creating a Tapwave Native Application for the Palm OS Simulator

You can also create a Tapwave Native Application for Palm OS Simulator. These are called *Simlets*. To create a Simlet, you'll need Visual C++ 6.0 with SP5 and the Tapwave SDK. A Simlet consists of two parts:

- The Tapwave Native Application for device
- A Windows DLL that contains the compiled x86 code.

The advantage of creating a Simlet is that you can debug your code quickly using the Palm OS Simulator. Note that you should still test your application thoroughly and frequently on the device because there are many differences between the Simlet and device runtime environments.

The GameStarter sample code includes the following files necessary to build the DLL:

GameStarter.dsw Visual Studio workspace file.

GameStarter.dsp Visual Studio project file.

7.Testing on a Tapwave device

Testing on a device is straightforward. Build the project in CodeWarrior 9.2 and transfer it to a device.

The easiest ways to transfer it to a device is to copy it to the /PALM/Launcher/ folder on an SD card. When you insert the card in the device, your application shows up in the Home screen. Just tap the icon to run it.

You can use the Palm Universal OS Debugger to perform source level debugging of your code on the device. The Palm Universal OS Debugger can also be used to transfer your PRC to the device.

8.Testing on the Palm OS Simulator

Testing on the Simulator is a bit more complicated but it allows you to perform source-level debugging of your code from Visual C++.

First, it's helpful to understand the debugging cycle on the Simulator. Debugging on the Simulator involves 4 components: the Palm OS Simulator, Visual C++, your application's PRC, and the Simlet DLL.

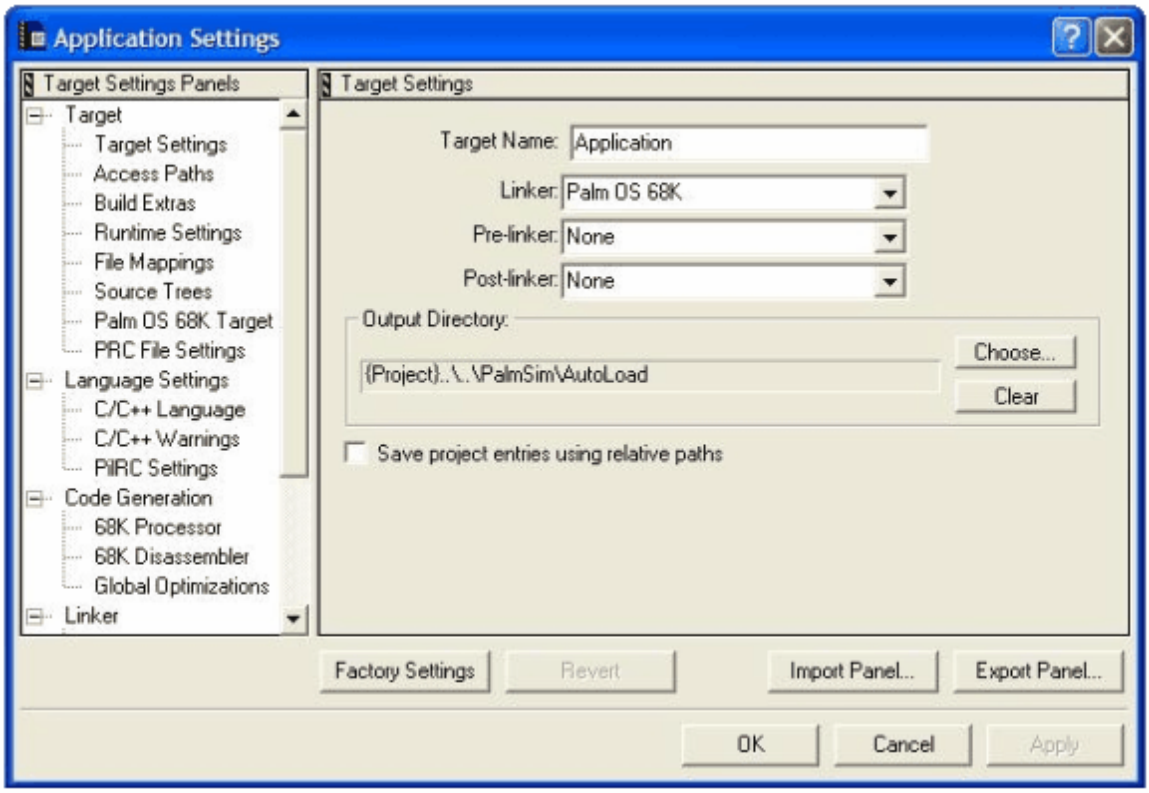
Traditionally you need to place your DLL in the Simulator folder and then load your PRC into a running instance of the Simulator. The PRC is necessary because it contains the 68k code that launches the x86 code in the DLL. The DLL is necessary because it contains the native x86 code that is executed when you launch your application. Note that you only need to re-compile your PRC under two conditions: when you change the 68k code or when you make changes to your resources. However, since most of your code changes will be to the DLL code (rather than the 68k code) your PRC will not change, in most cases. So you won't need to use CodeWarrior very often to build a Simlet.

You can automate the process of copying the DLL code and the PRC by using some tricks with CodeWarrior, the Palm OS Simulator, and Visual C++.

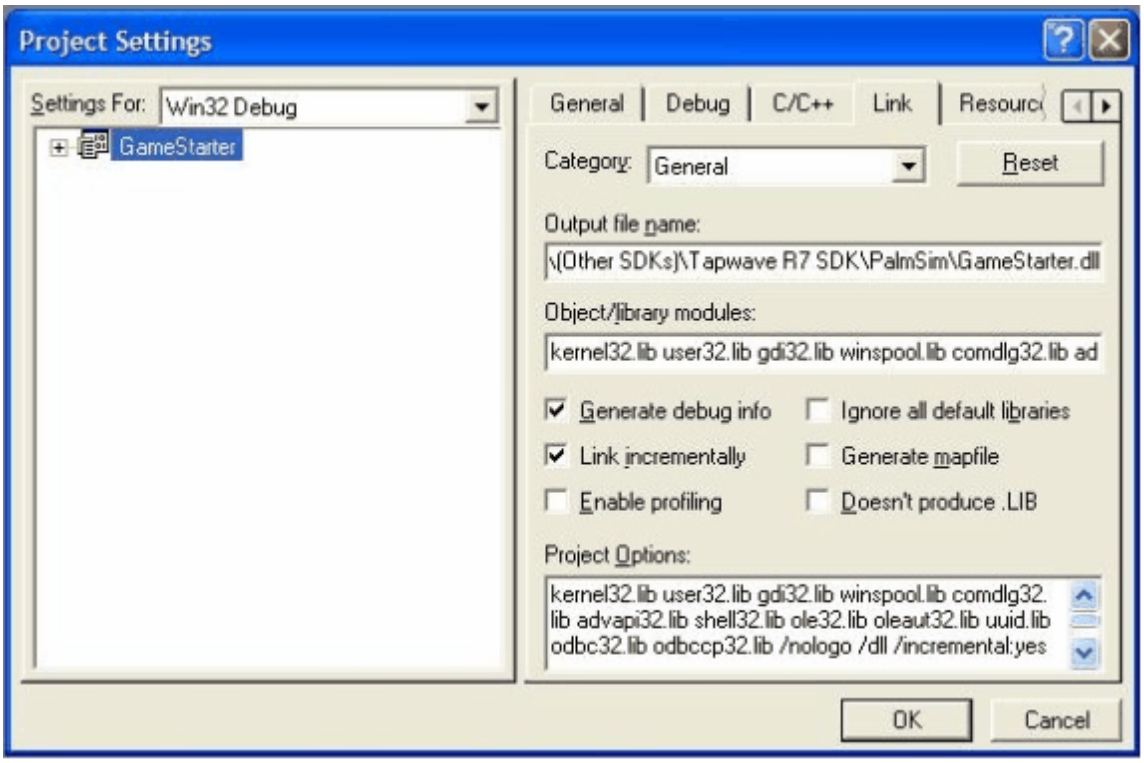
1 . Go to the Simulator\AutoLoad folder. If this folder doesn't exist, then create it.

The contents of this folder are loaded into the Simulator each time the Simulator is launched or reset. You should place your PRC into this folder.

2 Change the output directory in CodeWarrior so that your application builds are always placed in the AutoLoad folder. Open your project settings and under "Target Settings," change the "Output Directory" to the "AutoLoad" folder. Be sure to modify the 68k target and NOT the ARM target.



3 Change the DLL Output directory in Visual C++ to the Simulator folder. Go to "Project Settings", select "Link", select the "General" category, and change the "Output file name" to include the path of the Simulator folder.



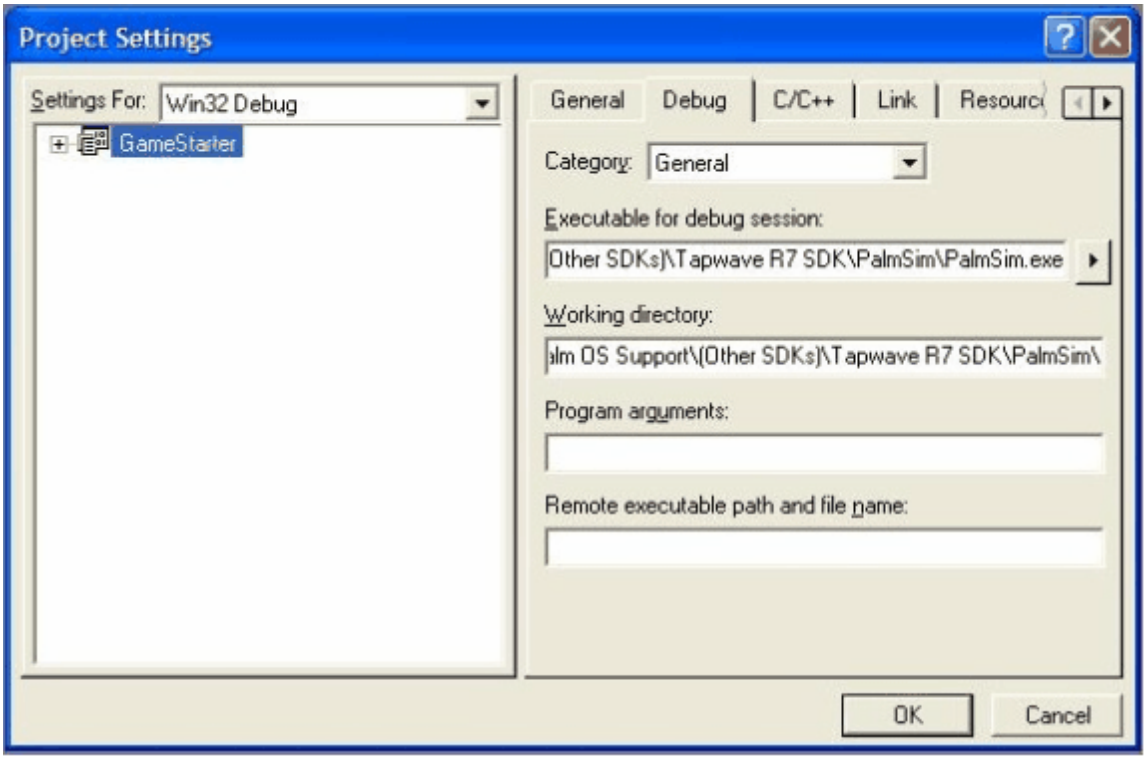
4 Update Visual C++ to enable debugging.

a Select “Settings” from the “Project” menu.

b Click the “Debug” tab and select the “General” category.

c Set the “Executable for debug session” path to the Simulator executable.

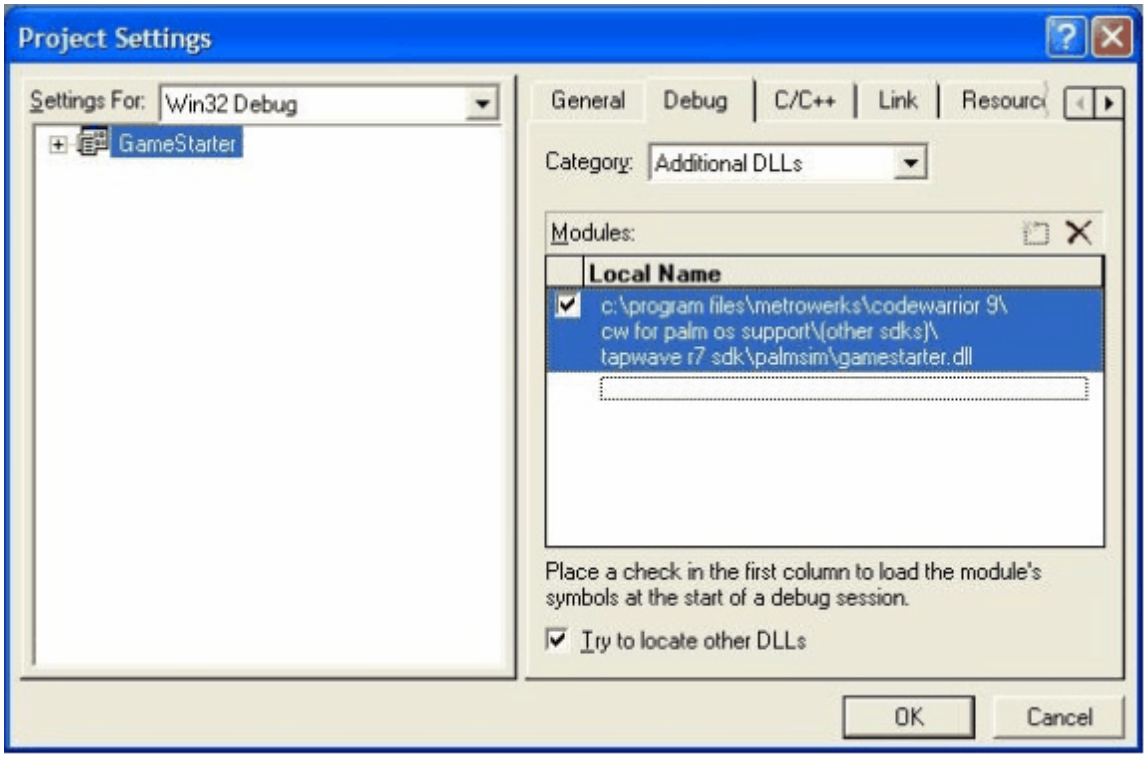
d Set your working directory path to the Simulator folder.



5 Make sure your project is set to load symbols for your DLL. By default, the VC++ debugger only loads symbols for the Simulator. By adding your DLL, you can debug your code.

a Select "Settings" from the "Project" menu.

b Click the "Debug" tab and select the "Additional DLLs" category.



You're now ready to debug!

Build your PRC in Codewarrior, then switch to Visual C++ and start debugging from the "Build" menu (F5). This builds your project and launches the simulator for you. Select your application from the launcher and you should be up and running!

9.Navigating on the Palm OS Simulator

To navigate through the Simulator you need to attach a game controller to your PC.

We recommend using the Microsoft Sidewinder Game Pad Pro. It has buttons for the analog controller, action keys, trigger keys, function, and home buttons. Most USB analog game pads should also work.

You can test the game controller's performance/configuration by launching the "Navigation" application. This application shows the analog controller location as well as the state of all other keys. If you have problems getting the analog controller working, or prefer using the keyboard, you can use the following keyboard mappings in the Simulator:

Key Combination	Action on the Simulator
Alt-A	Action A
Alt-B	Action B
Alt-C	Action C
Alt-D	Action D

Alt-L	Left Trigger
Alt-R	Right Trigger
Alt-F	Function Button
Alt-H	Home Button

Note that the Simulator supports multiple key presses and key-up.

10.Managing Game Data

There are two different models you can use to manage your game's data files, depending on the size of the files.

Note that the game data files should be marked as read/write in both models.

10.1.Model A

The first model is to embed all of your game data (sounds, bitmaps, etc.) directly in your application's PRC file or PDB file. The advantage of this model is that it keeps the download and installation of your application simple. We suggest developers use this model if their application is relatively small (i.e.: less than 500k).

10.2.Model B

The second model is to separate your game data into separate files so that your PRC only includes the game's compiled code. We suggest you use this model if you have a large amount of game data – this ensures the most efficient use of system memory.

If you use this model, you must also create an installer to ensure that files are installed in the correct location. See "Installing Your Game" for more information on installing files.

Use the following guidelines for managing your game's data files:

All game data should be stored on a VSF volume (either on an external memory card or on the internal card) using the following path: "/PALM/Programs/<prcdatabasename>-<creatorid>/", where <prcdatabasename> is the Data Manager's database name of the main application PRC file; the `appl` file.

For instance, if your application's database name is "Application" and the creator id is "Strt", then the path would be "/PALM/Programs/Application-Strt/". The following functions, defined in `TwOSAdditions.h`, are provided to help facilitate path creation: `TwGetDBDataDirectory` and `TwGetPRCDataDirectory`.

You can override the default path name by adding a `'Twdp'` resource with ID #0 to your main application file. This resource should specify the path to use in place of

"/PALM/Programs/<prcdatabasename>-<creatorid>/". You should only override this if you need to share data between applications.

Following this convention reduces clutter and allows the Tapwave launcher to copy game data correctly between the internal memory and an external memory card. The game data follows the game PRC if the PRC is moved from internal memory to external memory and visa versa. Note that all data in the application's directory is deleted when the application is deleted through the Tapwave launcher. This could have repercussions if you share data amongst applications.

Game preferences and saved game states should still be stored using the Palm OS Preferences or Data Manager APIs, the above information only applies to read-only game data such as graphics and sound files.

11.Installing Your Game

There are two ways to distribute and install your game depending on which model you use for managing your game's data files.

11.1.Model A

If you use Model A from above, then distributing and installing files is simple. You can distribute your PRCs and PDBs as separate files. In this case the user needs to double-click all install files to queue them for installation. Alternately, you can create an installer that calls the Install Aide API functions for installing PRCs and PDBs (e.g: `PltInstallFile`).

11.2.Model B

If you use Model B from above, then you must create an installer.

First, install your PRC and PDB files, as in Model A, using the appropriate Install Aide API (e.g: `PltInstallFile`). Then install your game's data files using the new `PlmSlotInstallFileToDir` API, which is included in the Tapwave desktop as part of Install Aide.

See the "PlmSlotInstallFileToDir" section in the *Tapwave Programmer's Reference* for more information. Also see the simple installer for samples that demonstrate how to use this API.

Appendix A - Important Runtime Features

There are several important runtime features developers must be aware of while developing TNA applications.

• SYSTEM_CALLBACK

The declarative macro SYSTEM_CALLBACK must be used for all callback function implementations. It is used to inform the compiler to generate the proper function prolog to set-up the global pointer register. Failing to do this will cause the callback function to crash immediately. For CW9.2 and CW9.3, this macro is expanded to __declspec(pace_native_callback).

• MSL

Tapwave SDK provides complete C/C++ runtime library support through Metrowerks Standard Library (MSL), which Tapwave licensed from Metrowerks. MSL provides standard C/C++ features, such as printf(), malloc(), fopen(), open(), opendir(), operator new(). There are some glue functions defined inside Startup.cpp which connect the MSL to the underlying PalmOS APIs. Advanced developers can customize this glue for their own purpose.

• File Path

The above mentioned MSL uses Windows file path to specify different volumes. For example, fopen("C:/foo", "r") opens file "foo" at the root directory of internal volume. "A:" stands for the first volume on the left slot of Zodiac. "B:" stands for the first volume on the right slot. "C:" standards for the volume on internal memory filesystem (MemFS).