Features

The Palm OS Emulator

Keith Rollin
Senior Software Engineer
3Com Palm Computing Division
krollin@palm.com

The Palm OS Emulator (formerly Copilot) is a Windows and Macintosh application for emulating the execution of ROMs from Palm Computing devices such as the Pilot 1000, Pilot 5000, PalmPilot, and Palm III. It has a number of features for accurate emulation of the Palm Computing hardware platform, plus additional features for debugging Palm OS applications.

The Palm OS Emulator is primarily a developer's tool, although anyone can run it. This means you can, for example, play your favorite Palm OS game right on your Windows or Macintosh desktop.

In The Beginning

The Palm OS Emulator began life as the UN*X Amiga Emulator (UAE), written by Bernd Schmidt in Germany (www.freiburg.linux.de/~uae/). Both the Commodore Amiga and the Palm devices are based on 680X0 microprocessors, so the CPU emulation source code for UAE was used as the basis for the Copilot CPU emulation code.

In mid-1996, Greg Hewgill (www.hewgill.com/) started working on Copilot, using UAE as a starting point. To the core 68000 CPU emulator, he added support for specific features of the 68328 Dragonball processor used in Palm devices as well as support for the rest of the hardware. In September 1996 he released the first beta of Copilot 1.0.

Over time, he added more features to Copilot and more support for subsequent Palm devices, resulting in Copilot 1.0 b9, released in mid-1997. *[For more details on Greg's work see "Copilot: Design and Development" on Page 34. Ed]*

Copilot Goes Travelling

Greg was interested in seeing his work benefit as many people as possible, so in October 1996 he posted the source code on his Web site. His invitation to others to port Copilot to other platforms and extend his work was accepted. So far, Copilot has been ported to Macintosh (two versions), Linux, BeOS, OS/2, and even Windows CE.

Palm Lends A Hand

In December 1997, I left Apple's Newton group to work for Palm Computing in their Development Tools group. As the second employee in this group, there was a lot for me to do. One of these tasks was making some modifications to Copilot for our upcoming Palm III product. Specifically, Copilot needed modification to support ROMs larger than 1 MB, grayscale displays, a larger dynamic RAM heap, and the new ways the Dragonball registers are accessed.

Working from Greg's 1.0b9 sources and the sources for the Macintosh version from Illume Software (members.aol.com/illumesoft/illume.html), these features were quickly added and made available to our developers (www.palmpilot.com/devzone/index.html).

Copilot Becomes the Palm OS Emulator

While working on those changes to Copilot, we realized that there was a lot of untapped potential in the emulator. Up until now, Palm's answer to Palm OS emulation on the desktop was the Simulator, a set of CodeWarrior libraries for the Macintosh built from the same sources sed for creating the actual Palm OS. Developers write their Palm OS applications, link with these libraries, create a stand-alone Macintosh application, and run and debug it just like a typical Mac application.

There are a number of advantages to this mode of development, including fast turnaround time, source-level debugging, fast execution, and the creation of desktop demos. There are some drawbacks as well, such as creating and debugging two versions of your application, having the Mac version of your application run in an environment that is not exactly the same as the environment in the Palm OS device, and being locked into using a Macintosh for your development. We thought that we should merge the advantages of the Palm Simulator with Copilot, resulting in a truly killer development tool for Palm OS developers, spanning multiple platforms.

We talked with Greg Hewgill (the original Windows Copilot author), Craig Schofield of Illume Software (author of the Macintosh version of Copilot), and Bernd Schmidt (author of UAE and the core 68000 CPU emulator) and got their permission to take over development of Copilot, using their work as a starting point. They were all in support of our taking over Copilot development, especially when we declared our intentions to keep the resulting product free and make the source code available to developers.

Along the way a transformation took place. Palm Computing is moving away from using the word "Pilot" in its product names. Thus, the PalmPilot became the Palm III, the PilotDebugger is becoming the PalmDebugger, and Copilot became the Palm OS Emulator.

Throughout the rest of this article, I use Copilot to refer to the emulator as Greg originally wrote it and Palm OS Emulator to refer to the product Palm Computing is working on.

New Features

With permission to take over its development and with an engineer dedicated to the task of development, we (myself and other Palm Computing engineers) started to map the changes we wanted to make to the Palm OS Emulator. The next few sections briefly discuss these changes.

Common Source Code Base

Every programmer has his or her own programming style: Bernd has his, Greg has his, and Craig has his. When Craig ported the Windows version to the Mac, he did a lot of fine work improving Greg's already excellent Windows version. Craig reformatted the sources, made function and variable names more descriptive, and added comments. When he was done, his Copilot version was functionally equivalent to the Windows version, but with a slightly different implementation.

The first task was to unify the source code base of the two versions. Because I was going to develop and support the Windows and Macintosh versions of the Palm OS Emulator, I wanted to make the changes once and have them appear in both versions simultaneously.

In the end, I ended up with a set of sources where 93 percent of the code is common to both platforms (not counting the PowerPlant sources going into the Mac version).

Latest UAE Sources

Since the time Greg took the UAE sources and used them as the basis for Copilot in mid-1996, Bernd updated his own version of UAE, culminating with version 0.69 released in May 1997. Since I was already playing havoc with the sources to unify the Windows and Mac versions, I also decided to update to the latest UAE sources. By doing this, I took advantage of the bug fixes and performance enhancements Bernd added to the 68000 CPU emulator.

Support for Large Applications

Since its inception, one of the long-standing problems with Copilot was its inability to load applications or databases from files larger than 64 KB. The reason for this limit was the way Copilot tricked the emulated Palm OS ROM into loading the application. Copilot would:

- Carve out a special section of memory (essentially adding 64 KB of RAM to the emulated memory space in which the ROM was running);
- Format it as a 64 KB memory chunk (like you get when you called `MemPtrNew`);
- Load the contents of the PRC or PDB file into that chunk; and
- Fake a call to the ROM's `DmCreateDatabaseFromImage` function.

The problem with this approach is its reliance on a chunk of memory compatible with what is returned by `MemPtrNew`. In Palm OS 1.0 and 2.0, these memory chunks are limited to 64 KB. Further, this approach has compatibility problems with Palm OS 3.0, where the memory chunk header changed from six to eight bytes. Using the original technique simply would not work when Copilot was executing a Palm OS 3.0 ROM.

It now takes a new approach. Instead of directly calling `DmCreateDatabaseFromImage`, the functionality of `DmCreateDatabaseFromImage` is implemented directly in the Palm OS Emulator. The PRC or PDB file opens, a new database is created based on information from the file's header information, the file's contents (a collection of Palm OS resources) are iterated, and each resource is added to the new database one by one. By using this technique, the Palm OS Emulator is not limited to 64 KB memory chunks, nor is it dependent on the format or size of memory chunk headers.

External Debugger

One of the advantages of the Palm Simulator on the Macintosh is that developers can use CodeWarrior's source-level debugger when developing their applications. By comparison, the Windows version of Copilot has a simple-machine-language, command-line-oriented debugger, but the Macintosh version doesn't have any sort of debugger at all.

In February 1998, we invited Eric Cloninger of Portable Computing Products, Mark Corry of Metrowerks, and Kenneth Albanowski of the Silver Hammer Group to visit us, enticing them with beta Palm III ROMs. The intent was to discuss adding support to get the Palm OS Emulator and external debuggers (such as Metrowerks debugger and UNIX's gdb) working together.

At the time of this writing (in March 1998), the basic support for external debuggers is working fairly well. A prerelease version of Metrowerks' debugger is performing source-level debugging of Palm OS applications running in the Palm OS Emulator. Palm Computing's internally hacked version of the PalmDebugger is debugging applications at the machine language level. Soon, we will give the Silver Hammer Group our source code so they can make similar changes to the UNIX versions of Copilot and their gcc/gdb tools. SoftMagic (the makers of Satellite Forms) and other purveyors of Palm OS development tools will also be given the source code.

Gremlins

Part of the Palm OS Emulator effort is bringing many of the tools that Macintosh developers have had all along to Windows. One set of these tools is Gremlins. A Gremlin is a series of pseudo-random events that are fed to a Palm OS application, causing it to respond to stimuli that might not be produced by human testing. If a bug in the application being tested is found, the series of events that cause the problem can be reliably reproduced, ensuring that the problem is fixed. Gremlins are supported in the PalmDebugger and in the Simulator, making them unavailable to developers using tools on other platforms.

The Palm OS Emulator is now enhanced with Gremlin support. By selecting the New Gremlin menu item, you can choose several options, including:

- The Gremlin to run (what seed to use for the random number generator);
- The number of pseudo-random events to feed to the application;
- The application to test; and
- The kind of event logging to perform.

With Gremlin support in the Windows version of the Palm OS Emulator, more developers can now participate in the Quality Partners certification program (www.qpqa.com/qp3com.html).

Speed Improvements

When examining the features of the Simulator that gave it an advantage over Copilot, the only one we can't really add to the Palm OS Emulator is performance. Programs created with the Simulator libraries are true, fully functional, native Macintosh applications running at the top speed of the Power PC microprocessor. Programs running in the Palm OS Emulator are executed by a virtual microprocessor with each machine language instruction emulated by a different subroutine that updates registers and accesses simulated memory locations. Even on top-of-the-line PCs and Macintoshes, emulation speeds are 2 million Dragonball opcodes per second. While this speed is on par with the execution speed of an actual Palm OS device, it is still orders of magnitude lower than a native desktop application.

I took two approaches to increasing the performance of the Palm OS Emulator over Copilot. The first approach is based on a major tenet of performance enhancement: if an operation is taking too long, make it faster. Using this approach, I found a number of places that could be sped up by a few percent. One of the more interesting ways the emulator could be sped up was by using a technique used in the Newton group: take a function that can logically be divided into commonly-executed and not-commonly-executed sections, and put the second section into its own function, thus simplifying the first section. The result is a smaller function that the compiler can easily optimize. By combining all of these small speed improvements, overall execution speed increased by about 20 percent.

The other approach to speeding up emulation is based on the other tenet of performance enhancement: if an operation is taking too long, don't call it. Application of this rule usually entails redesigning algorithms, adding caches, and being smart about calculations and recalculations. But in the case of the Palm OS Emulator, we can target a specific set of emulated operations, and, instead of emulating them, we program the Palm OS Emulator to execute them with native code.

The Palm OS trap dispatcher is an example of this technique. When a Palm OS application calls a system function such as `MemPtrNew` or `DmCreateDatabase`, the compiler embeds into the application a `TRAP $F` instruction followed by a dispatch number of the form `$Axxx`. When the application executes the `TRAP $F` instruction, the CPU generates a hardware exception, resulting in the CPU pushing the program counter and status register onto the stack and jumping through a vector in low-memory.

The Palm OS installs a pointer to a function called `TrapDispatcher` into this low-memory exception vector. Thus, when the `TRAP $F` executes, `TrapDispatcher` is called. `TrapDispatcher` looks at the program counter that was pushed onto the stack, uses it to fetch the dispatch number following the `TRAP` opcode, and uses the dispatch number to jump to the requested ROM function.

The process doesn't take very long, but it does take a finite amount of emulated time and it happens whenever a Palm OS application calls a ROM function (as well as when a ROM function calls another ROM function). That is to say, it happens a lot.

To speed things up, I gave the Palm OS Emulator special knowledge of this ROM function-calling technique. The emulator essentially bypasses the whole `TrapDispatcher` function and does the same thing by hand. Now, when the Palm OS Emulator encounters a `TRAP $F` instruction, instead of jumping to `TrapDispatcher` by loading the appropriate exception vector into the PC, it does the same operations as `TrapDispatcher` with native x86, PPC, or 68K code. It:

- Looks up the emulated program counter that was pushed onto the emulated stack;
- Uses the program counter to fetch the dispatch number following the `TRAP $F`;
- Finds the array of ROM function pointers residing in low-memory;
- Fetches the appropriate function pointer based on the dispatch number; and
- Stores that function pointer into the emulated program counter.

Having the Palm OS Emulator do all these steps instead of emulating the `TrapDispatcher` is like having fast trap dispatcher functionality built right into the microprocessor. With a native trap dispatcher, the overall execution speed increased by 12 percent.

Other ROM functionality was replicated in this fashion. For instance, `MemSet`, `MemMove`, and `RctPtInRectangle` are all frequently-called functions that can be implemented in the Palm OS Emulator directly, without worrying about being closely tied to a particular Palm OS implementation.

Logging

Because we want this tool to be useful to developers, we want to provide information that is helpful when developing applications. One of the most-often asked questions when a program crashes is about what just happened. Ninety percent of the time, the ultimate cause of a program crash is nowhere in the immediate vicinity of the crash itself.

With logging capabilities added to the Palm OS Emulator, a developer has a record of what happened immediately prior to the application's crash. The developer can look at the events posted to the application's queue and the events the application pulled off the queue and responded to. He or she can also get a list of the ROM functions called. Eventually, the developer can log the machine language opcodes executed prior to the crash. If I get inspired, I'm planning on even adding an Undo facility, whereby the entire machine state can be backed up and replayed so that the developer can examine in slow-motion the events and operations leading up to the crash.

Memory Stressing

The first step to getting an application working is to get it working in a friendly environment. That is, make sure the application does what it is supposed to do when fed expected input and when unexpected events don't get in the way.

The next step is to bullet proof the application by making sure it also works in an inhospitable environment. Examples of a way developers can do this today is by running their applications under debug versions of the Palm OS ROM and by using the `MemSetDebugMode` function to turn on selected memory-debugging options, such as heap scrambling and new block zapping.

The Palm OS Emulator includes support for these facilities, and adds a few of its own. For instance, the emulator examines the dynamic heap when an application quits, looking for unreleased memory blocks and memory leaks. If any are found, it reports them. Another feature the Palm OS Emulator adds is stack clearing: when an application's subroutine returns to its calling function, the stack below the current stack pointer is cleared, hopefully flushing out any stale pointers to old stack contents. Finally, the emulator also detects and reports stack overflows.

Conforming Applications

The Palm OS Emulator isn't intended just for developers. Users of Palm OS applications can use it to check for conforming applications. Palm Computing has an interest in seeing that applications running on its OS are as robust and bug-free as possible (hence tools like Gremlins and the certification and logo programs). Support was added to the Palm OS Emulator to ferret out applications that don't run entirely bug-free today and that may not run on future devices. The Palm OS Emulator checks for accesses to low-memory, Dragonball register memory, and screen memory. Programs accessing those locations are flagged as possible violators of the Palm OS programming guidelines. (Following Developer Support guidelines is a good idea. The Palm OS Debugger checks for violations of some of these guidelines and may check for more violations in the future. See "Guidelines and Practices" on page 32.)

Because of these conformance checks, a customer can download applications from specialized Web sites, run them through the Palm OS Emulator, and see how well they stack up when running in Full Gauntlet mode. If an application passes, the customer knows that an investment in that application is money well spent.

Miscellaneous Other Changes

Other features are planned for the Palm OS Emulator. Support for drag and drop will be added so users can drag ROM images or applications into the emulator window to load and run them. An AutoLoad folder containing items that are automatically loaded into the Palm OS's emulated memory at start time may be added. Synchronizing with the Palm Desktop, profiling applications to determine where time is being spent, and a save-game feature that saves the entire CPU state along with the contents of RAM (so that the whole thing can be restored later) are all possible additions.

Palm OS Emulator Internals

The complete source to the Windows and Macintosh versions of the Palm OS Emulator will be released, following the lead of the UAE and Copilot authors. For those ambitious folks who are interested in how it all works, the following sections provide an overview of the emulator's internals.

CPU Emulator

I mentioned earlier that 93 percent of the code going into the Palm OS Emulator was shared directly between the Windows and Macintosh versions. By far, the largest part of that 93 percent is the CPU emulator.

A CPU emulator is simple: it reads an opcode from memory and executes it. In fact, here's an excerpt from the main CPU loop in the Palm OS Emulator:

```
while (gCPUState == kCPUState_Running)
{
  …
  uae_u32 opcode = nextiword ();
  …
  (*cpufunctbl[opcode]) (opcode);
  …
}
```

This snippet does just what I described: it uses `nextiword` to fetch an opcode from the memory location indicated by the current PC, and then uses that opcode as an index into a table of 65,536 opcode handlers to fetch the address of and execute the correct opcode handler.

In another sense, a CPU emulator is very complex: there are lots of opcodes and many tiny details to account for when emulating those opcodes. Status register bits are updated, memory accesses are checked and performed, exceptions are handled, interrupts are triggered, supervisor mode rules are observed, conditions are determined, and branches are made; it's enough to drive anyone mad.

I'm sure Bernd Schmidt thought the same thing. The first version of UAE (version 0.1, available on the previously mentioned UAE Web site) was a painstakingly hand-written CPU emulator. As UAE development continued, Bernd hit upon the idea of a CPU emulator compiler. This compiler accepts input as a small, yet very descriptive, text file containing descriptions of all the 680X0 opcodes. It then uses those descriptions as a guide when it generates all the opcode handler functions. The result: 16 source code files containing thousands of opcode handlers and another file containing the massive `cpufunctbl` array that points to them all.

Here's an example of what I'm talking about:

```
1101 rrr0 zzss sSSS:00:XNZVC:——: ADD.z   s,Dr
```

This is the entry from the CPU description file for an ADD machine language instruction. The left hand side of the line contains the meanings for all the bits in the opcode: `0` and `1` are interpreted as-is, `r` stands for register numbers, `z` stands for register size, and `s` stands for source addressing mode. Following the opcode bit description is an indication of the CPU on which that opcode can be executed (`00` corresponds to the 68000, of which the Dragonball is a direct descendent). Following the CPU level is a description of the status register flags that are used and affected by the instruction. The opcode description is last, along with an encoding of the addressing modes enabled by the opcode.

Here's an example of a function created by the above opcode description:

```
void REGPARAM2 CPU_OP_NAME(_d000)(uae_u32 opcode)
   /* ADD */
{
 uae_u32 srcreg = (opcode & 7);
 uae_u32 dstreg = (opcode >> 9) & 7;
{{ uae_s8 src = m68k_dreg(regs, srcreg);
{ uae_s8 dst = m68k_dreg(regs, dstreg);
{{uae_u32 newv = ((uae_s8)(dst)) + ((uae_s8)(src));
{ int flgs = ((uae_s8)(src)) < 0;
 int flgo = ((uae_s8)(dst)) < 0;
 int flgn = ((uae_s8)(newv)) < 0;
 ZFLG = ((uae_s8)(newv)) == 0;
 VFLG = (flgs == flgo) && (flgn != flgo);
 CFLG = XFLG = ((uae_u8)(~dst)) < ((uae_u8)(src));
 NFLG = flgn != 0;
 m68k_dreg(regs, dstreg) =
  (m68k_dreg(regs, dstreg) & ~0xff) | ((newv) & 0xff);
}}}}}}}
```

This function first gets the source and destination registers. The `srcreg` is encoded in the lowest three bits of the opcode (corresponding to the `SSS` in the opcode description line shown above), while the `dstreg` is encoded in the three bits corresponding to the `rrr` on the opcode description. The register numbers then fetch the actual values out of the registers (an array of 16 integers maintained by the CPU emulator). The two values are added together, all of the appropriate status register bits are updated, and the new value is stored back into the destination register.

Memory Access

In addition to the performance optimization tenets mentioned earlier in this article, there's the size versus speed rule. It is often the case that you can make your functions and algorithms smaller, but usually at the expense of slowing them down. The converse is also true: you can often speed up your application by letting it eat up gobs of memory (until you reach the point where you're using so much memory that virtual memory thrashing slows your program to a crawl).

The `cpufunctbl` previously shown in the main CPU loop is an example. By using a 256 KB table loaded with 64 KB function pointers, we can dispatch opcodes to their corresponding handlers very quickly.

Emulated memory access is handled similarly. The entire addressable 4 GB memory range is divided into 65,536 segments (called a bank in the UAE sources) that are each 64 KB long. Each of these banks is managed by a set of functions. When a memory access operation needs to be done, the Palm OS Emulator uses the upper 16 bits of the address (the bank number) as an index into a table with 64 KB entries. Each of these entries is a four-byte pointer to the set of functions managing that bank. The Palm OS Emulator then retrieves the function appropriate for the operation it needs to carry out and calls it.

A simple example clarifies this description. Assume that we treat the first two banks (the first 128 KB of the 4 GB address range) as plain RAM. First, we create a set of functions to treat a chunk of memory as readable and writable RAM:

```
uae_u32 RAMBank_GetLong (uaecptr iAddress)
{
 return do_get_mem_long((
  (char*) gRAM_Memory) + iAddress);
}

uae_u32 RAMBank_GetWord (uaecptr iAddress)
{
 return do_get_mem_word(
  ((char*) gRAM_Memory) + iAddress);
}

uae_u32 RAMBank_GetByte (uaecptr iAddress)
{
 return do_get_mem_byte(
  ((char*) gRAM_Memory) + iAddress);
}

void RAMBank_SetLong (uaecptr iAddress, uae_u32 iLongVal-
ue)
{
 do_put_mem_long(
  ((char*) gRAM_Memory) + iAddress, iLongValue);
}

void RAMBank_SetWord (
  uaecptr iAddress, uae_u32 iWordValue)
{
 do_put_mem_word(
  ((char*) gRAM_Memory) + iAddress, iWordValue);
}

void RAMBank_SetByte (
  uaecptr iAddress, uae_u32 iByteValue)
{
 do_put_mem_byte(
  ((char*) gRAM_Memory) + iAddress, iByteValue);
}

int RAMBank_ValidAddress (
  uaecptr iAddress, uae_u32 iSize)
{
 int result = (
  iAddress + iSize) <= (uae_u32) gRAMBank_Size;

 assert(result);

 return result;
}

uae_u8* RAMBank_GetRealAddress (uaecptr iAddress)
{
 return (uae_u8*) &(((char*) gRAM_Memory)[iAddress]);
}
```

In the above functions:

- `uae_u32` is an unsigned 32-bit value (similarly for `uae_u16` and `uae_u8`);
- `uaecptr` is the UAE type for memory pointers (akin to `void*`);
- `gRAM_Memory` is a block of memory we allocate with `malloc` when the application starts; and
- `do_get_mem_long` and it's friends are bottleneck memory accessors.

After a consistent set of functions is defined, they are grouped together into a data structure called an `AddressBank`:

```
AddressBank  gRAM_Bank =
{
 RAMBank_GetLong,
 RAMBank_GetWord,
 RAMBank_GetByte,
 RAMBank_SetLong,
 RAMBank_SetWord,
 RAMBank_SetByte,
 RAMBank_GetRealAddress,
 RAMBank_ValidAddress
};
```

A pointer to this address bank is then installed into an array of 65,536 `AddressBank` pointers:

```
gMemory_Banks[0] = &gRAM_Bank;
gMemory_Banks[1] = &gRAM_Bank;
```

After the entire array is initialized this way, memory accesses are performed using a set of macros that do all the bank selecting, function fetching, and function calling.

Dragonball Support

Many of the internals examined so far are based on the emulation work done by Bernd Schmidt. This emulation supports a generic 680X0 processor. Greg Hewgill added support for the 68328 Dragonball registers.

The Dragonball registers are memory-mapped CPU registers that control hardware functions. Examples of their uses include LCD control, serial port I/O, real-time clock access, and alarms. The registers are found in a 4 KB range of memory starting at `0xFFFFF000`. Access to them in the Palm OS Emulator is managed by a special set of `AddressBank` functions.

Even after working on the Palm OS Emulator for several months, I'm amazed at the level of Dragonball support Greg added to Copilot. By adding the right set of functionality behind the UART registers, he added support for serial I/O, bridging the emulated ROM serial routines to the Win32 serial access API. By twiddling the right bits and triggering the right interrupts, he got Copilot to think that the mouse was a pen drawing on the LCD screen. By strategically incrementing counters at exactly the right rate, he got the emulated ROM to go to sleep right on cue. And he did all of this emulation work without the benefit of access to the Palm OS source code, something I find extremely handy when I forget to twiddle a particular bit in just the right way and send the ROM spinning off into the middle of nowhere.

Screen Updates

Naturally, all of this hard work is useless if we can't see the results. The emulated Palm OS writes data into the emulated LCD buffer, but none of that means anything to the host running the emulator. We need a way of transferring the contents of the LCD buffer to a host window.

The method used in the Palm OS Emulator is blunt, but it works. Ten to 20 times a second, the Palm OS Emulator takes a break from emulating instructions and devotes its attention to the LCD buffer. Using an ingenious function called `memcmpy`, it not only compares the current contents of the LCD buffer with a previously saved version of the LCD buffer, but it also updates that saved copy with the new contents. The result of `memcmpy` is a Boolean value indicating whether or not the contents of the LCD buffer changed since the last time we performed the comparison. If it did, we take the additional step of converting the contents of the LCD buffer into a host-specific bitmap and copying the bitmap to the emulator window.

Upon casual examination, this approach seems a bit overkill. I thought a better approach might be modifying the `AddressBank` functions to detect accesses to the range of memory for the LCD buffer, setting some sort of flag when a change was made to the buffer.

This approach is not as efficient as I first thought. Those `AddressBank` functions are called a lot. The check to see if the memory being written to is part of the LCD buffer is more expensive that I initially expected. The result is that the brute force method is rather efficient by comparison.

Another approach I considered was to somehow hook into the drawing functions and set my LCD-dirty bit when they are called. I stay away from programming specific OS information into the emulator as much as possible; hard coding a set of known drawing functions violates that guideline. If the set of functions ever changes, I must release a new emulator.

One variation on this approach that should work is to take advantage of an undocumented system function called `ScrDrawNotify`. If a certain undocumented low-memory global is true, then all the drawing functions call `ScrDrawNotify` as the last thing they do, passing in the bounds of the affected drawing area. I figured that I could set this low-memory global, intercept calls to `ScrDrawNotify`, and set my LCD-dirty bit whenever it was called.

That was the theory. But, as the wise man said, "In theory, theory and practice are the same. In practice, they're not." While the approach based on `ScrDrawNotify` seems most efficient, I opted not to use it for several reasons. It requires that I hard code special knowledge about

Guidelines and Practices

Maurice Sharp, Manager of Palm Computing Developer Support, suggests the following tips for Palm software developers.

Here are some guidelines and practices for making your programs more robust and compatible with next generation Palm Computing products.

One general-purpose way to make your code more robust is to write defensive code by adding lots of calls to `ErrNonFatalDisplayIf`, so that your debug builds can check assumptions. Many bugs are caught in this way, and these extra calls don't weigh down your shipping application. You can keep more important checks in the release builds by using `ErrFatalDisplayIf`. Here are some additional, specific items to check:

- Reading and writing to NULL or low memory: Remember to check your calls to `MemSet`, `MemMove`, and so on, to make sure the pointers are non-`NULL`. (If you can do better validation than that, so much the better.) Also validate that pointers your code obtains from some structure or API call are not `NULL`. Consider, in a debug build, overriding `MemMove` (and similar functions) with a `#define` to validate the parameters.
- Allocating zero-length objects: It's not valid to allocate a zero-byte buffer, or resize a buffer to zero bytes. Palm OS 2.0 and earlier releases allowed it, but future revisions of the OS might not permit zero-length objects.
- Making assumptions about the screen: The location of the screen buffer, its size, and the number of pixels per bit aren't set in stone; they might well change. Don't hack around the windowing and drawing functions. If you are going to hack the hardware that will circumvent the APIs, save the state and return the system to that saved state when you quit.
- Directly accessing globals or hardware: Global variables and their locations can change. Use the documented API functions or disable your application if it's run on anything but a tested version of the OS. Future devices might run on a different processor than the current one.
- Don't overfill the stack: Allocating large numbers of local variables, or extremely large ones, can result in hard-to-debug heap corruption. The stack is only about 2 KB; be stingy with stack-based variables.
- Built-in apps can change: The format and size of the preferences (and data) of the built-in applications is subject to change. Write your code defensively, and consider disabling your applications if they are run on an untested version of the OS. ✔

a low-memory global that is set to `true`. Not only is this low-memory global undocumented, but its location changed between Palm OS 2.0 and Palm OS 3.0. Marking the screen dirty only when using the supported graphics API modifies the screen and causes compatibility problems with applications that draw directly to the screen. While Palm Computing frowns upon such practices, we don't want to be blind to the fact that such applications exist.

In the end, I chose to stay with the `memcmpy`-based approach because, frankly, it works and is good enough. Profiling shows that almost immeasurable time is spent updating the screen, so it makes little sense to make it faster.

Patching System Functions

In order to support keyboard input and sound output, Greg devised a way of intercepting calls to system functions. By intercepting calls to `EvtGetEvent`, Greg took the keys typed by the Copilot user, turned them into an event record, like the one filled out by `EvtGetEvent`, and returned the result to the caller of `EvtGetEvent`. All the caller of `EvtGetEvent` knows is that it got back a data structure containing a key event. It doesn't matter what is the source of the event.

Similarly, by patching `SndDoCmd` he intercepted calls by clients of the Sound Manager and performed platform-specific operations based on those calls. By looking at the parameters passed to `SndDoCmd`, he told Windows to beep in a manner similar to a Palm OS device.

The method for intercepting ROM function calls is very simple and based on the technique for speeding up trap dispatching. When an application calls a ROM function, it executes a `TRAP $F` instruction. Copilot intercepts that `TRAP $F` instruction, fetches the dispatch number that follows it in memory, and then checks to see if the function corresponding to that dispatch number needs special handling.

The Palm OS Emulator takes this notion of system function patching several steps further. It replaces some ROM functions with identical functions implemented directly in the emulator. This use of system function patching completely replaces the intercepted function. The ROM is not involved at all.

The emulator also patches system functions to feed Gremlin events into the system. For instance, the System Manager calls `SysEvGroupWait` when there are no events to report to the application. Normally, `SysEvGroupWait` blocks the current application thread, letting other threads execute, or putting the Palm OS device into doze mode until there is an event to report. If a Gremlin is running, the Palm OS Emulator intercepts `SysEvGroupWait` and posts a new event. It then lets `SysEvGroupWait` execute as normal. Since we just posted an event to the event queue, `SysEvGroupWait` returns immediately, letting that event report to the application.

All uses of system function patching described so far are instances of head patching – inserted processing that takes place before the actual ROM function is called (if the ROM function is executed at all). But there are cases where we need to do tail patching. In tail patching, we interrupt the flow of control after the desired system function executes.

Tail patching is trickier because there is no signal that a system function is returning. When a system function is called, the event is heralded with the execution of a `TRAP $F` instruction, which is easily identified and intercepted. When a system function returns to its caller, it merely executes an RTS instruction, which is not special by any means.

To regain control after a system function executes an RTS requires a little setup. The technique first intercepts the `TRAP $F` instruction that results in that system function being called. Then, instead of emulating the current PC being pushed onto the stack (which would result in control being returned to the caller of the system function when the `RTS` was executed), the Palm OS Emulator saves the PC in a table and pushes a pointer to a `TRAP $E` instruction onto the stack. That system function is now tail patched. When the system function executes the `RTS`, the PC points to the `TRAP $E` instruction. We already know that intercepting `TRAP` instructions is easy for the Palm OS Emulator to do, so regaining control is simple and fast. When the `TRAP $E` is executed, the Palm OS Emulator changes the PC back to the value that it saved earlier in a table and then performs any necessary tail patching operation. In this manner, the Palm OS Emulator tail patches `EvtGetEvent` to record retrieved events when event logging is turned on.

Calling System Functions

Intercepting system functions is only half of a very interesting relationship between the Palm OS Emulator and the Palm OS. Effectively, we are transferring control from the emulated Palm OS ROM to native code in the Palm OS Emulator. In order to complete this relationship, we need a way for the Palm OS Emulator to call functions in the ROM.

The method used in the Palm OS Emulator is a variation on a technique Greg devised for Copilot. The idea is to stop emulation of the ROM at some convenient and generally safe location (Greg stops execution at `TRAP $F` instructions), save the CPU register state, push some parameters onto the emulated stack, and then emulate a different `TRAP $F` instruction for the desired ROM function. Control is regained when the system function returns in the same way as for tail patches: we store a return address to a `TRAP` instruction (in this case, a `TRAP $D` instruction) on the stack before calling the system function. When the system function returns, the `TRAP $D` function is executed, the Palm OS Emulator picks up on that fact, and cleans up by restoring the CPU register state to what it was before we made the call. In this manner, we are able to invasively make system function calls without disturbing the normal ROM flow.

The Palm OS Emulator uses this technique a lot. For example, as previously discussed, applications are installed from PRC files by creating a database and adding the resources from the PRC file to that database. These operations are performed by calling into the emulated ROM and executing functions such as `DmCreateDatabase` and `DmNewResource`.

The Road Goes Ever On

In this article, I discussed how the Palm OS Emulator got its start from the pioneering work done by Greg Hewgill, Bernd Schmidt, and Craig Schofield, and how the Palm OS Emulator is implemented. You also received a glimpse of where the Palm OS Emulator is going.

Development of the Palm OS Emulator is currently in progress. I expect that incremental releases will occur throughout the year as features are added and as support for new Palm OS devices is provided. If you would like to participate in the evolution of the Palm OS Emulator, you can subscribe to the interest group mailing list at ls.palm.com and follow the appropriate links. ✔

Resources

- Palm Computing developer Web site: www.palmpilot.com/devzone.index.html
- Palm Computing mailing list Web site: ls.palm.com/
- Developer support or information: mailto: devsupp@palm.com
- Greg's Copilot Web site: www.hewgill.com/
- Illume Software's Copilot Web site: members.aol.com/illumesoft/illume.html
- UAE Web site: www.freiburg.linux.de/~uae/

We thought that we should merge the advantages of the Palm Simulator with Copilot, resulting in a truly killer development tool for Palm OS developers, spanning multiple platforms.