Copilot: Design and Development

Greg Hewgill
greg@hewgill.com

My goal was to emulate, with no documentation except the Motorola Dragonball reference manual, what was then known as the Pilot. The Pilot was a handheld device with a 68000-based CPU, an LCD touch screen, 512 KB of ROM, a serial port, and various hardware buttons. I was going to try to emulate it in software. To tell this story properly, I must start at the beginning.

Discovering the Pilot

My first encounter with a Pilot was when I saw one on a coworker's desk sometime in June 1996. He described it to me as "just like a Newton, only smaller." He showed me the basics of the Graffiti writing style and handed me the Pilot and the Graffiti reference card. I was hooked. A few days later I purchased a Pilot 5000 from a local retailer and, being a software developer, I started looking for information on how to write my own programs for this little gem.

I was encouraged by the amount of developer information on the Pilot web site, but I couldn't find the Windows tools. All I found was the Metrowerks CodeWarrior for Pilot package, which only ran on the Macintosh. I didn't have a Macintosh. So I bought the only thing that worked with Windows: The Conduit Software Development Kit for Windows. (I still haven't tried to actually write a conduit.)

Fortunately, there was at least one other developer out there who had a Pilot but no Macintosh. Darrin Massena (darrin@massena.com) had a Web site (www.massena.com/darrin/pilot/) and wrote a piece of software best described as a "discovery tool." PilotHack is its name; it's a monitor for computers where operating systems are optional. PilotHack shows you every single byte of the RAM and ROM in hex and ASCII. It still resides on my Pilot today.

Darrin managed to put together some small Pilot applications using Microsoft's Visual C++ 4.0 Cross-Development Edition for Macintosh. With a serious amount of tweaking, the compiler generated application images that the Pilot loaded. He also put together PILA, the PILot Assembler. PILA ran on Windows and created Pilot application images from 68000 assembly code.

Finally, Darrin wrote the article "Writing Pilot Applications Under Windows," which described what a state-of-the-art Windows SDK for Pilot would look like. This SDK included an editor, a compiler, an assembler, a resource editor, a linker, project management, a debugger, an emulator, samples, OS header files, documentation, on-line help, and a Windows-based IDE to integrate all the pieces together. Realizing that it was not possible for a single developer to do all this, he put out a call to other developers for support.

When I read Darrin's list, I couldn't help but be drawn toward the emulator project. After all, I had some experience with emulators (I had written an Apple][+ emulator in the past), I knew some 68000 assembly language, and the potential hack value if the project was successful was just too great to pass up.

The First Steps

The idea for Copilot was planted. Actually, it wasn't called Copilot until much later; its first name was Pilotsim. Before Pilotsim it didn't have a name, it was just a modified UAE.

The first order of business when writing an emulator is grab a copy of the code you're trying to execute. In the case of the Pilot, it was the 512 KB ROM that was the Pilot kernel, operating system, function library, and application suite. Darrin added a feature to PilotHack that let me download an arbitrary 16 KB chunk of data via HotSync. By running this 32 times over the entire ROM space, I constructed an image of the ROM on a PC. With the 512 KB PILOT.ROM file in hand, the next step was to fetch the first machine instruction and start executing.

To execute code for a CPU other than the one you're currently running on, you need a program called an emulator. In my case, the goal was to execute Motorola 68000 CPU instructions on an Intel x86 based computer. While it was possible to write my own 68000 emulator, surely most of the work had already been done by somebody else.

It didn't take much searching on the Internet to find UAE, the Un*x Amiga Emulator. The Amiga uses the same 68000 family of CPUs in the Pilot, so an emulator for the Amiga has a lot of code in common with an emulator for the Pilot. I extracted the core CPU emulator from UAE, removing all the support for the custom Amiga graphics chips and other hardware that wasn't present on the Pilot. Bernd Schmidt, the author of UAE, kindly let me use his 68000 core code in Copilot.

The MC68328 Dragonball CPU used in the Pilot has a very flexible memory architecture. There are many configuration registers for features such as 8-bit or 16-bit word size, memory bank addresses, and write protect flags. Nearly all these features are set once during the power-on cycle and never change again. Instead of emulating the exact behavior of these features, Copilot knows about the proper memory configuration of a Pilot and ignores the actual memory configuration parameters set by the ROM software.

Once past the initial startup code, the Pilot ROM starts to initialize the various hardware elements. During one of these initialization routines, the ROM code waited for some kind of timeout to expire. At this point, I didn't know exactly what it was trying to initialize. Carefully tracing the code showed that it was waiting in a loop for something external to happen. By looking up the register values in the Dragonball reference manual, I found that it was waiting for one of the hardware timers. Since my emulation simply ate the values programmed into the timer registers without properly acting on them, the code was waiting for something that would never happen.

Getting the timer to working was just one of many functions I studied in the Dragonball reference manual. This was a common pattern:

- Trace the code until it fails in some way (usually this involved hanging while waiting for something to happen in the hardware).
- Find out which Dragonball registers are being accessed and which part of the Dragonball they control.
- Study the Dragonball reference manual to find out what the ROM code is trying to do and what the emulated hardware does to properly respond to the ROM.
- Write code to emulate (as precisely as necessary) exactly what the real hardware does when accessed in the same way.

Notice my comment "as precisely as necessary." The Dragonball emulation in Copilot is far from a complete Dragonball emulation. In general, only those registers that are essential to the correct operation of the Pilot ROM code are implemented. In some cases only one or two control bits out of a single register are implemented. For all unimplemented Dragonball registers, there is a hard-coded breakpoint in the emulation code. I used this during development to find out when the ROM code accessed a register that I had not yet implemented. (Incidentally, this is why Copilot stops with an Exception 03H when running a PalmOS 2.0 ROM on a pre-2.0 version of Copilot. PalmOS 2.0 accesses a couple of new Dragonball registers to control the backlight, and these were not implemented in earlier versions of Copilot. Exception 03H is a hard-coded breakpoint exception on Intel CPUs.)

Up to this point, the only way to find out that the emulation code was doing anything was to trace it, instruction by instruction, in a debugger. There was no screen output at all. So the next task was grabbing the Pilot screen memory out of the RAM area and copying it to a window on the Windows display. Copilot became a multithreaded program at this point: the CPU emulation thread is compute-bound and does not have time to check for an updated screen area. Another thread was created to watch for updates to the screen area and copy them to the display when it noticed the changed contents.

The Input Problem

Finally, after many hours of tracing, coding, debugging, reading Dragonball documentation, more tracing, and certainly some luck, the "Welcome to Pilot" screen appeared on my Windows desktop. The first visible evidence of success took shape. There was no turning back now.

Of course, my excitement quickly dampened when I realized that I was still looking at the "Welcome to Pilot" screen. The next thing the Pilot does is enter the stylus calibration routine. As it turned out, there were a couple of problems. Due to a bug in the timer emulation, the ROM code executed extremely slow and there was another execution hang bug. With those problems out of the way, the stylus calibration screen appeared.

But the stylus calibration presented a big problem. The emulator had no idea how to accept input from the mouse. Sure, tapping on the Pilot screen with the stylus inserts a `PenDown` event into the event queue, but I wanted to know how it really happened. Since Copilot is a hardware emulator, it just knows hardware registers and not event queues. I wanted to know what really happens when you tap the stylus on the touch sensitive screen.

There is no special register on the Dragonball for stylus-tap input. I had no documentation on the Pilot hardware. The only reference I had was the ROM code, but 512 KB is a lot of code when you're looking at individual instructions. Instead of reading the ROM code, I opted for a more experimental approach.

Using PILA, I wrote a small assembly language program to run on my real Pilot. This program hooked the CPU interrupt vectors and displayed information on the screen regarding which interrupt fired and when. Since this code executed at interrupt time and minimized its interaction with the rest of the system, it didn't call the usual character output routines. Besides, I wanted to display information on more interrupt events than could fit on the screen in character mode. So these interrupt hook routines displayed status on the screen using specific dot patterns. To read them I literally counted pixels using a magnifying glass. Using this technique I found out which interrupt the pen fired.

The interrupt was only half the problem. Finding out how to supply the pen coordinates was an even tougher problem. After many more hours of tracing and experimentation, I discovered the pen interface uses the Serial Peripheral Interface Module (SPIM) of the Dragonball. The X and Y coordinates were read one at a time: first a command byte is output to the SPIM control register, then the pen coordinate is read from the SPIM data register.

The elapsed time between seeing the stylus calibration screen for the first time and being able to actually click on it was about a week and a half. This was probably the largest single hurdle in the development of Copilot. But what a reward. Once the mouse clicks were seen as pen taps by the ROM software, the rest of the emulation worked. I was paging through the Preferences screens, switching applications, reading the preloaded To Do items, and even writing Graffiti with the mouse. Since Graffiti is simply pen movements in a certain area of the screen, there was no additional effort needed in Copilot to support it.

This was a great moment in the history of Copilot. I knew it was all downhill from that point. There was still a lot of work to do, including a better debugger, serial port support, and hardware button support.

The Copilot Debugger

The debugger I used in Copilot up to that point was a very primitive version of the debugger in UAE. It had problems disassembling certain opcodes, no symbol support, no breakpoint support, and was unusable for all but the simplest debugging tasks. Darrin Massena offered to write a better debugger and he produced a very extensive debugger with many Pilot-specific features.

In addition to standard debugger features such as breakpoints, symbolic addressing (both input and output), and stack trace, the most important feature in Darrin's debugger is the `bp -na` command. The `bp` command sets a breakpoint at a specific address. This is great if you know in advance where your code is loaded, but does not work for a dynamically-loaded application. It's not easy to discover exactly where the application code resides. The `-na` options modify code in strategic places inside the ROM to trigger some debugger code when a new application starts. In this way, you simply execute your application and the debugger breaks at your application's first instruction.

Another feature of the new debugger is automatic symbol table loading. Nearly every PalmOS application has debugging symbols embedded in the application image. The symbols immediately follow the last `RTS` instruction of each subroutine and consist of a length byte (with the high bit set) followed by the name of the preceding subroutine (I understand this is the same convention used in the Macintosh for MacsBug). When a new application loads, the debugger automatically locates the symbols embedded in the application image and enters them into the debugger symbol table. This makes debugging applications immeasurably easier.

Completing the Emulation

While Darrin was working on the debugger, I was working on emulating the serial port. If you wanted to run an application other than one of the built-in applications, it was necessary to HotSync the application from the Pilot Desktop to the emulated Pilot device. Fortunately, the serial port support in the Dragonball CPU is a fairly straightforward single-byte I/O. This was easy to translate into Win32 communications function calls, and once this was working I successfully negotiated a HotSync by connecting the two communications ports on my computer together with a null modem cable. Copilot talked to one of the serial ports, the HotSync application talked to the other, and together they downloaded application code into Copilot.

Another hardware feature not yet emulated was the eight hardware buttons (four application buttons, up and down, on/off, and the HotSync button on the cradle). I discovered how these buttons work much the same way as the pen taps: by writing small programs that watched interrupt activity and displayed dots on the screen indicating what happened. It turns out that the hardware buttons are more or less directly mapped to the INTx pins of the CPU.

One of the features of the Dragonball CPU is hardware support for a sleep mode. This is a very low power mode that the Pilot uses when the power is turned off. Of course the power is never completely off, otherwise the Pilot would not respond to the application buttons or trigger alarms. Instead, the ROM software instructs the Dragonball to go to sleep, but also sets some special trigger bits that tell the Dragonball which interrupts it should interpret as a wake-up call. In the Pilot these trigger bits are set for all but the up and down buttons. Pressing any of these buttons while the CPU is in sleep mode immediately causes it to wake up and respond to the request. (In order to handle alarms, the Dragonball also periodically wakes up for a brief moment to check the list of pending alarms.)

Somewhere around this stage in the development, I publicly released Copilot on my web site as a Beta 1 version. I sent an announcement to a couple of Pilot mailing lists and news groups, and almost immediately I started receiving messages of thanks from fellow developers. Most were as skeptical as I was when I started the project (about creating an emulator for a largely undocumented piece of hardware), but soon Copilot became a favorite among developers and end users.

Loading Applications

Copilot was still not complete because loading applications was too cumbersome. HotSync was the only way to do it, and, while it worked, it really slowed down the compile, run, and test cycle. There had to be a better way.

I tried many approaches to solve this problem. My first idea was to internally simulate the action of the HotSync program, supplying the right data to the serial port to make the emulated Pilot think it was participating in a HotSync operation. I tried to analyze the HotSync data stream, but without the relevant documentation (I did not have the full PalmOS SDK at the time), this idea wasn't going to work.

While analyzing the HotSync data stream, I learned a little bit about what HotSync was doing. A PalmOS application image is structured as a set of resources. A resource can contain code, data, icons, and other types of program elements. During a HotSync, the application transfers to the Pilot device one resource at a time. When the HotSync operation completes, the application database contains all the resources necessary to run the application. So all I had to do was duplicate these operations without using HotSync.

I read about `DmCreateDatabase`, learned how to insert records in the new database using `DmNewRecord`, sorted out the differences between handles, pointers, and local ids, and figured I was ready. Unfortunately, I couldn't directly call `DmCreateDatabase` from Win32 code and my Intel CPU doesn't execute Motorola instruction codes. Somehow, I needed to make the correct database functions execute within the emulator.

The basic idea in my solution to the database problem is this: interrupt whatever the emulated CPU is doing, modify the next instruction to be the correct API call, modify the stack to make it appear as though the correct arguments to the API call are pushed, place any required data (such as a resource image) somewhere in memory accessible to PalmOS, and resume execution of the CPU. On return from the API call, capture the function result (either in the A0 or D0 register) and restore the instruction stream, the state of the stack, and the original contents of any used memory buffers. Finally, resume normal execution of the original instruction stream.

There were some important details that needed addressing in the above scheme. I was not sure that the PalmOS database routines were re-entrant, so I could not interrupt the emulated CPU while it was processing any other API call. To solve this problem, I added a feature to the CPU emulator that triggers a special breakpoint when it encounters a `TRAP $F` indicating an API call. When Copilot is asked to load a program, it first enables this breakpoint and then waits for the breakpoint to be hit. While the Pilot is turned on, PalmOS API calls happen very rapidly so it does not take long before one is hit. At this point the application load function stops the emulated CPU and performs all the subsequent database operations before resuming normal CPU execution.

Once everything is set up in memory, the CPU resumes and the API call executed. Copilot knows when the API call completes because of the breakpoint instruction (opcode `$4AFC`). This opcode is placed in the instruction stream immediately following the trap code in the `TRAP $F` instruction. When the API call completes, the emulated CPU hits the breakpoint instruction and notifies the application loader. At this point the function result (D0 or A0) is captured and the original instruction stream restored.

Once the above machinations rolled up into a neat little C++ class called `FakeCall`, making PalmOS API calls from Win32 code was as easy as this:

```
FakeCall fc;
fc.PushLong(dbid); // dbID
fc.PushWord(0); // cardNo
fc.Call(sysTrapDmDeleteDatabase);
err = fc.GetResultD0();
```

(This code snippet deletes the application database if it already exists.) I then wrote code to open the database, create new records, copy the data into the new records, and close the database. This code was rather complicated because it needed to read the PRC file from a disk file, split it up into individual resources, and insert each resource separately. I didn't have very good documentation on the format of a PRC file at the time, so this was very much a hit-and-miss bit of code. (Actually, it missed most of the time.)

For some reason, the process of adding each resource individually did not work very well. There could have been a bug in my code, misuse of one of the database API calls, or any one of a number of other problems. It was at this point, while I was reviewing the documentation for one of the database calls, where I came upon `DmCreateDatabaseFromImage` (I have no idea why I never noticed this call before). This function did exactly what I was trying to do manually in Win32 code. Simply point the function at a PRC file image, let it go, and it automatically creates the application database with all the proper resource records. This was exactly what I needed.

So, out came all the ugly code to deal with individual resources and in went `DmCreateDatabaseFromImage`. This worked really well, but there was one remaining problem. When calling `DmCreateDatabaseFromImage`, one of the parameters is a pointer to the application image. I was allocating this space using `MemPtrNew`, copying the image into this allocated space, and then freeing it once the application loaded. The problem was `MemPtrNew` allocates memory from the dynamic heap. As every PalmOS developer knows, the dynamic heap is very limited in size (at most 14 KB is available on my Pilot). Therefore, the largest application that loaded was about 12 KB (some extra space was needed to complete the operation). This was clearly not enough.

`DmCreateDatabaseFromImage` requires a pointer to the application image. I was allocating this pointer in the dynamic heap. I created a separate area of memory used exclusively for loading applications. The next iteration of the application loader did just that. I added support to the CPU module for a 64 KB scratch RAM area located at absolute address `0x10000`. I set up the six-byte heap block header to make it appear as legitimate allocated memory, loaded the application image into this space, and called `DmCreateDatabaseFromImage`. This worked wonderfully. Applications up to nearly 64 KB loaded with no problems.

The only limit left was the 64 KB limit, but this could not be increased without significant changes in the application loader. `DmCreateDatabaseFromImage` requires a valid heap pointer (it expects to see the heap block header preceding the pointer). In PalmOS 1.x and 2.x, the maximum size of a heap block is 64 KB, because there are only 16 bits available to store the size of the block. Working around this problem required a completely different approach.

Further Development

By late October 1996, Copilot was very stable, even though it was still labeled a Beta version. I started work on other projects and development on Copilot slowed down considerably. When the PalmPilot with PalmOS 2.0 released in the spring of 1997, there were some compatibility problems with Copilot, the most obvious being the 1 MB ROM size of the PalmPilot versus the 512 KB ROM size of the original Pilot. I increased the ROM size easily, but encountered a strange performance problem with the pen up action that I could not explain.

The problem was that pen up events were not delivered in a timely manner. This caused the operation of Copilot to appear sluggish and unresponsive. I talked with some of the developers at Palm Computing and developers that ported Copilot to other platforms. Nobody could come up with a reason why this problem occurred, and I never did find a solution. I made some changes that decreased the impact of the problem, but it never really went away. If anybody can discover what causes this problem, I would be happy to hear from you.

Due to the pen performance problem, I put off the next release of Copilot, hoping that I would find a solution. In June 1997 I finally released an updated version that supported the PalmOS 2.0 ROM. This version had the debugger disabled because the patches the Copilot debugger did to enable the `bp -na` feature no longer applied to the new ROM (users with non-English ROMs actually encountered this problem before).

Copilot was no longer near the top of my project list and had not been for quite some time. Another Windows developer, Heath Hunnicutt, offered to take over the development of Copilot, but after releasing a few versions (with several new features) Heath stopped working on Copilot, too.

Enter Palm Computing

In early 1998, I found out that Palm Computing had a Copilot developer on staff (Keith Rollin) and that they were planning a significant update to Copilot. This is excellent news. I no longer had the time to continue development on Copilot, but it is too valuable a tool to neglect. Of course, Palm Computing is in the perfect position to make Copilot a truly industrial-strength product. Expect to see some very cool things coming out of Palm in the Copilot area.

Conclusion

Copilot was a very exciting project for me. It seemed nearly impossible at first, but by tackling the problems one step at a time it eventually fell together. Seeing it actually run for the first time was a moment I won't soon forget. I am happy to see that Palm Computing recognized that Copilot is a valuable addition to their tool set, and I hope that it benefits future developers of PalmOS compatible software. ✔