A Tcl-Powered Handheld Computer For Telecommunications Test Automation

Karl Lehenbauer
Procplace.com, Inc.

Abstract

This paper describes a recently completed project to create a Tcl-based handheld computer for telecommunications test automation and record keeping. The application is explained. The thinking behind the decision to use Tcl is recounted. What we did to fit Tcl into a small space (less than one MIPS and one megabyte of RAM) is chronicled. Finally, we share the lessons learned, and the software.

1. The Customer And The Application

The customer is Acterna, a large communications test and management equipment manufacturer. They had an existing product, the T-BERD 310, which is a portable network monitoring and analysis instrument that provides SONET, DWDM (Dense Wave Division Multiplexing), ATM, DS3, DS1/DS0, and optical media testing, all in one package. A picture of the T-BERD appears in Figure 1.

The user interface of the T-BERD consists of a two-line display and several dozen pushbuttons and indicators. Procedures to set up the unit, for example to prepare for the turn-up of a DS3 channelized SONET circuit, were complex and time-consuming. But the company's engineers had cleverly included a serial port and created a simple communication protocol for it, through which every indicator could be read and every button could be pressed by remote control. This meant that tens of thousands of units in the field were upgrade-ready for what they were now considering.

Figure 1: T-BERD 310 Communications Analyzer

2. Proof Of Concept

Acterna's developers had created a prototype handheld computer using a very limited device with a proprietary language. It was successful enough to convince them that they had a good idea, but the lack of expressiveness of the language and the miniscule memory capacity of the unit convinced them that they needed more power.

3. Design Issues

Acterna's team considered the possibility of using Palm Pilots or one of the other consumer handhelds for use with the TB-310. Advantages included low cost, wide availability and good development tools. These were rejected, however, principally due to the rigors of the rugged telecom environment. Likewise, laptops were considered too expensive, too fragile, and too cumbersome, as well as being attractive targets for theft. They decided to use a ruggedized handheld.

Minimizing dependence on a specific vendor's unit was a design priority. This provided protection in case the product went away or if a more desirable product became available. The design called for a scrollable menuing system that could break menu entries into multiple lines if needed, making the menu system fairly independent of the number of lines in, and character width of, the display.

Knowing that the scripts were going to be fairly complicated and recognizing the value of using a language that is somewhat standard and fairly widely known, they began looking at Tcl.

The handheld computers they were interested in ran an embeddable version of MS-DOS (this is pretty common). The one they selected is built by Two Technologies, Inc. (*http://www.twotech.com/*) and has 640K of addressable RAM and flash memory for disks. A picture of this computer appears in Figure 2.

The first question to answer was, can a useful enough version of Tcl fit in that space and leave enough room to run scripts of thousands of lines?

We looked to Tcl 6.7. An order of magnitude smaller than the subsequent Tcl version 8, and still several times smaller than Tcl 7, Tcl 6.7 had actually been used on DOS in the past. It was fairly easy to port. We had it up and running in a day or so.

Figure 2: Handheld Computer From Two Technologies

The 6.7 core compiled to 45 Kbytes, excluding the C library. The final version is around 180K, including the C library, a number of C extensions to interface to DOS, some isolated pieces of Extended Tcl (TclX), and a number of other C extensions. We called it *TinyTcl*.

4. DOS Extensions for TinyTcl

We created a number of new Tcl commands for interfacing with DOS, which works with MS-DOS, DOS windows under Microsoft Windows, and embedded DOS systems that have PC-like hardware. These extensions provide Tcl-level access to the hardware's *BIOS calls*, and the operating system's *DOS interface commands*, and *DOS video functions*.

4.1. BIOS calls

bios_memsize returns the amount of physical memory, while **bios_equiplist** a bit of insight into the hardware's capabilities.

com provides an interface to BIOS serial communications. **com** can configure them, check their status, and do I/O with them.

4.2. DOS Interface Commands

The DOS interface commands are split into a number of functional areas.

The *keyboard functions* comprise **kbhit** to check to see if a key has been pressed, and **getkey**, which reads the keyboard.

The *time, date, clock and timer functions* include **getdate**, **setdate**, **gettime** and **settime**, which get and set the OS's date and time. Also, **rawclock** returns the raw clock value in ticks since the program started, typically at 100 per second, and **wait** delays the specified number of milliseconds.

Among the *drive functions*, **getverify** returns whether or not writes are being verified, while **drive** queries or set the drive letter.† Also, **diskfree** reads the free kilobytes and total kilobytes available on the specified drive, while **getfat** reads a drive's characteristics in gory detail. Finally, **mkdir** creates directories, while **unlink** deletes files.

The *memory functions*, **memfree** and **stackfree**, return the amount of available heap and stack, respectively, while **heapcheck** checks the heap for corruption*.

Among the *DOS screen functions*, **cls** clears the screen and **gotoxy** addresses the cursor to a specified location.

Finally, the *miscellaneous functions* **execvp** terminates Tcl and runs a specified program in its place, and **sound** plays a square wave at the specified frequency.

4.3. DOS Video Functions

The DOS video functions are distinct from the DOS interface commands. They include **video normal**, **video dim**, and **video bright**, to set the display intensity.

Subcommands **video write**, **video clear** and **video goto** do their respective things, while **video color** and **video bgcolor** set the foreground and background color. Finally, **video direct** controls whether we write directly to video or go through the BIOS.

5. Handheld Control Extensions

A new command, **twotech** was written to allow Tcl programs to control the specific capabilities of the Two Technologies handheld:

†Are you getting a feeling of deja vu?

*Yes, there really is a DOS call to check the heap for corruption.

- **twotech home** - Force the virtual screen "home".
- **twotech status** - Return information including battery and charger status.
- **twotech suspend** - Suspend the processor.
- **twotech backlight** - Turn the backlight on and off, and set backlight display timeout.
- **twotech reboot** - Reboot.
- **twotech cpuclock** Set the CPU clock to 8 MHz or 16 MHz.
- **twotech cpuspeed** Set the CPU speed divider, from automatic, full speed, or fractional speeds down to 1/32nd of normal.

6. Greenleaf CommLib Extensions

An extension was also written to support the Greenleaf communications library, CommLib, from Greenleaf Software (*http://www.greenleafsoftware.com/*). This extension allowed serial ports to be accessed, serial port handshaking to be controlled, and files to be sent and received using *YMODEM* and *ZMODEM* protocols. In addition to the file transfer capabilities, CommLib can move data over serial ports faster and more reliably, and with more control, than the BIOS can.

7. Lessons Learned

7.1. Performance

Tcl's performance can be a major issue on a 16 megahertz, 16-bit machine -- we had to code a few things in C for speed, particularly parts of the menu scrolling. We also employed the typical Tcl 6 performance hacks, like using arrays in place of lists where it was a reasonable fit.

7.2. Running Without Memory Protection

When you blow an array index or a pointer in C on a non-memory protected machine, the machine tends to crash in random and mysterious ways. We expected the venerable and well-tested Tcl 6.7 release to perform flawlessly in this respect.

Tcl ran great at first, but when the scripts got a little more complex we started having mysterious crashes that would wedge the handheld and require a hard reboot. When running in a DOS window under Windows 98, the DOS window would crash and sometimes Windows 98 would hang. By "mysterious" I mean that the crash didn't seem to occur at the same place or happen in the same way. Something was definitely wrong. To make matters worse, we could never reproduce the problem on UNIX, where we had excellent debugging tools, and we had not planned (or budgeted) for needing an in-circuit emulator, because we were so confident that Tcl 6 wouldn't give us any trouble.

I looked and looked for the problem. I would add print statements, and the crash would start happening in a different place.

I could hardly believe there was a bug in Tcl 6 so egregious as to be causing this, a memory overwrite or something. I tried everything I could think of. I even started trying those desperate sorts of things that I was pretty sure wouldn't work. Finally after substantial suffering, I'm carefully reading every word about every switch that you can set on the compiler command line, hoping to find *something* that will crack this problem. Hmm. Here's this switch that says "compile code to detect stack overflow." Hmm. Why would it need to *compile code* to check for stack overflow? Hmm. Hmm. Hey, this is just a wimpy '186 clone... so that means... it has a less than 64 Kbyte stack for sure, and it doesn't grow automatically... **A HA!**

Sure enough, the Borland 5 compiler, when compiling for DOS, creates a default stack size of 4 Kbytes. (Can you believe it?) How to change that wasn't in the Borland documentation, but it was in the "Using C on DOS" FAQ on the Internet.*

When you exceed the stack size on a non-memory protected machine, the machine crashes in the same sort of random and mysterious ways it does when the program is buggy.

An Intel 8086 class computer has a maximum stack size of 64 Kbytes. Each level of the Tcl call stack used an average of 1100 bytes. This gives you a maximum Tcl nesting depth of about 50 with the stack set to the max size.

Once this was solved, Tcl performed flawlessly, as originally expected.

7.3. Running Without Much Memory

For more than a decade, I've pretty much programmed exclusively in a virtual memory environment where I didn't have to think too much about how big the program was. (Some, but not much.) When you run out of memory and you don't have VM, your application

*http://www.faqs.org/faqs/msdos-programmer-faq/

tends to die. Tcl was not designed to be robust in an out of memory situation, nor do I think it is really possible or a good idea to try to make it do so. Yet... here we were, writing scripts that were thousands of lines long, collecting data, reading and writing files, etc, and of course... we ran out of memory.

We wrote a little "snapshotter" that recorded the names of all the procs and global variables and arrays that were in existence before a test script was run. After the test script completed, we would unload it by removing all of the procs and globals that were added after the snapshot was taken. That helped, but it didn't completely solve the problem.

When you're running out of memory and you can't add any, you start looking for how to make things smaller, like looking at Tcl's C code and poking around how the "source" command works for instance. In 6.7, it read the whole file into memory. I bought us some space by making the source command be able to fall back and read the file in a line at a time if it couldn't allocate all the memory it needed, but I felt that this was grasping at straws in that, as soon as "source" can't allocate 30 Kbytes or so, you're probably going to run out of memory doing something else shortly thereafter.

If you're going to run short of resources, it usually happens near the end of the project, when you have the most at stake and have spent most of the development money already. That's when you start wondering how bad it's going to be and whether or not it can ever be made to work and how much trouble you're likely to get in if it doesn't.

"Hmm, maybe I should turn off memory debugging." "Don't. We need it in case there's an overwrite." Sigh. "We're running out of memory again." "OK, that's it, I'm at least gonna try building it with memory debugging off."

Turning off memory debugging bought us an extra 120 Kbytes of RAM. Saved!

8. The Problem Where It Never Worked Between 8 AM and 10 AM

We had this weird problem. It was working really great. Then one of the engineers took it to demo it for a customer, and the software died every time he tried it. He brought it back and we checked it out and it worked fine. Then another guy couldn't get it to work for a demo.

Yes, we got burned once again by Tcl's implicit octal conversion of numbers with leading 0's. It wasn't until we were demoing the machine that we discovered that the software would fail between 08:00 AM and 09:59 AM due to failed octal conversion of the hour part of the time of day. Our nocturnal development cycles rarely overlapped that time window.

9. Interesting, But Painful

The point comes where you just go "Why am I doing this?" When you're sweating to save 40 Kbytes, you can't help but mentally calculate that that's less than two cents worth of memory at today's prices.

10. Was it worth it?

It would have been easier, as the guy who created this Tcl development environment for the customer, if the handheld had had several megabytes of memory, a fast processor, and run Linux or FreeBSD. Then we could have just run the latest version of Tcl, with all its capabilities, and been done with it.

We looked for a handheld like that and didn't find one. Had there been one, it would have also needed to be price competitive with the one we went with.

In effect, I went back in time to a develop on a suckier environment that I'm used to. And by doing so, I had to face "old" problems like a slow processor, small memory, no memory protection, and not much of an operating system.

But the customer is happy. And their customers are happy. The product makes network technicians more productive, speeding service turn-ups and reducing downtime.

11. Conclusion

You can write sizable applications using Tcl 6 with 640K of RAM. Acterna's scripts run to thousands of lines.

In operation, it's really cool.

By agreement with Acterna we've released the base Tcl port plus the BIOS and DOS extensions as open source, calling it TinyTcl 6.8. It's available from *http://procplace.com*.