stack. Further iterations can be obtained by NSTP. FF2 can be used for finding further solutions for the same equation from the same or different starting values.

Input				Command	Output	
Level 4	Level 3	Level 2	Level 1	->	Level 2	Level 1
0	1	1	'2*Y+X+1'	FF1	11.6808481406	11.6808481721
				NSTP		11.6808481731
	0	1	.4	FF2	2.94469662483	2.9446966249

Jeremy Hawdon (#600)

REF : A Structured Introduction to Numerical Mathematics by P.J. Hartley and A.Wynn-Evans

HP48S or HP48SX ?

David Hodges (#347) and Peter Embrey (#628)

In his recent article in DATAFILE [1], Simon Bradshaw criticised HP's decision to compromise on "adaptability" rather than "functionality" in the HP48S. He also invited comments.

Despite our reputation as "Forty-Eight Bashers" and, some would say, over zealous critics of its keyboard and menu layout [2], we belong to an exclusive group within HPCC, being proud owners of both the HP48SX and the HP48S. We have come to appreciate the virtues of the S; it is lighter in weight and much cheaper than the SX, and sharper in responding to keystrokes, yet it provides identical computing power. For virtually the same price as the HP28S, the 48S offers more potential and two I/O modes.

We believe that the idea of producing an "HP48X" from existing SX hardware, with card slots but fewer functions, is unrealistic. ROM development on a machine such as the HP48 is exceedingly expensive, and once complete it pays to standardise. A loss of functionality, far from creating a cheaper calculator, would require an expensive re-write of the firmware. It may be possible to simplify the calculator by suppressing certain menus, which would no doubt please the syseval lovers among us, but this would only decrease capability without reducing costs; hardly a recipe for success.

We like the HP48S. For once, we think HP has made the right decision.

REFERENCES

[1] Bradshaw, S; "Functionality vc Adaptability, or Swiss Army Calculators and Socket Set Handhelds"; DATAFILE V10N7P6 (1991).

[2] Embrey, P and Hodges, D; "A Better Keyboard for the HP48"; DATAFILE V10N3P3 (1991).

THE COLLECT-94 DEVELOPMENT SYSTEM

Craig A. Finseth (#745)

(This is article number two of a -- Datafile editor willing -- series of four.)

The Collect-94 development system was created between March and October in 1986. It was written in the C language on an IBM PC-compatible computer. The bulk of the programming was done by one person.

If you were to compare the above dates and the HP-94's introduction date, you would notice that work on the development system had started two months before a unit was available. We were able to get this "head start" for two reasons:

First, even though a unit wasn't available to us, we knew:

- that the CPU was an 8086-compatible,
- that there would be at least 64 KBytes of memory,
- that there was some form of serial I/O, and
- the screen was 4 x 20.

Second, we knew who we were trying to sell to and what would be important to them.

We were thus able to design the overall system and start prototyping applications using readily-available tools.

THE GOALS

We were evangelists: we wanted to make it possible for anyone to create a high-quality, easy-to-use data collection system. We looked at numerous existing systems: Smalltalk, the Macintosh system, other handhelds (including HP of course), many versions of the Basic language, and general user interface design criteria. We wanted to create something that:

1) was immediately familiar to people,
2) allowed them to create flashy programs with little work (this also allowed *us* to create flashy demo programs with little work...),
3) fit within their existing data processing environment,
4) was easy to implement and support.

A TYPICAL CUSTOMER

We judged that a typical customer would be a corporation's data processing department. As part of an overall switch to computer-tracked inventory or transactions, they would select a bar code system, code their applications to process the data on a mini- or mainframe computer, and expect the data collector to fit within the system.

They would also handle programming of the data collector: the people in the field would not be expected to do anything other than data entry.

The customer would have either IBM PCs (or compatible) or HP-150 computers that could be used for development.

We also expected two other types of customers: VARs and OEMs. In this market, we expected them to have needs similar to the corporation buying for its own use.

THE OVERALL SYSTEM

The development system consisted of these pieces: compiler, interpreter, testbed, decompiler, question-and-answer program, and assorted utilities.

We selected a compiler-based approach because we felt that the customer would rather program on an IBM PC-type computer than the handheld itself. We could thus save the memory in the handheld that would otherwise be required to handle the program-development side of things.

We also felt that the data processing customer would be used to a compiler-based environment and the ability to add comments, whitespace, use long variable names, and other such features all without an execution-time penalty.

This decision turned out to be a good one for several reasons. These reasons will be mentioned throughout the rest of this article.

THE COMPILER

This program did the obvious: it accepted a program source file and output an object file. The object file had been thoroughly cross-checked for syntactic validity so the interpreter did not have to (one of the aforementioned benefits). The object format was custom and had three main parts:

- a header
- a declarations area
- the program itself

The program was stored in a compact, tokenized form. All jumps (including implied jumps due to if-statements, etc.) had their target locations resolved.

The interpreter required that its expressions be in RPN. The compiler could accept both RPN and algebraic expression formats (if RPN, the expression was enclosed in @@ ... @). As it turned out, the algebraic parser was almost the last piece of code written: all of the demos were coded directly in RPN. The first algebraic parser worked almost perfectly: it could handle just about any expression that you could throw at it. However, it failed to parse a test expression that looked something like this:

```
(((((((a+1)))+2))*4)))
```

i.e., excessively-nested parentheses. Eventually, the whole parser was thrown out and another one written. This one was based on the Lisp language where it first turned the expression into a CONS-tree, and then did the compilation. Worked perfectly.

The algebraic parser did show one sign of its RPM history. A fragment such as:

```
max(3, -4)
```


would be treated as "max(3-4)" or "max(-1)" which is, of course, a syntax error as the max function requires two operands. You had to program around this by doing either:

```
max(-4, 3)
```

or

```
max(3, (-4))
```

As with other things, this would have been fixed.

THE INTERPRETER

This program interpreted the user's program. It pretty-much took over the whole machine: by the time we were done, the only thing that we use the ROM-based code for was the file system and, if we had continued, that would have been replaced as well.

Early on, we noticed that the display hardware was capable of smooth vertical scrolling. We wanted to take advantage of that feature and proceeded to try it out. We found that the display would start scrolling, then jump back. After fighting with it for a while, we gave up.

A few months later we had to replace the timer-interrupt code so that we could use the wand port (we were writing a software UART). This change also meant that we had to replace the keyboard handler and several other pieces of software, because the existing interrupt handler could not be partially replaced (it was in ROM). Buried deep in the timer interrupt was a gratuitous reset of the display scrolling register! As with many other features, if development had continued we would have added the smooth scrolling feature.

THE TESTBED

Only a small fraction of the interpreter was written in assembler. The rest was written in C. Of that code, only a small fraction was specific to the HP-94. By replacing only that fraction of code, we were able to run the same interpreter on the IBM PC or compatible or HP-150 development system. Virtually all features operated the same. The programmer could thus rapidly test new versions of their program. In addition, the testbed took advantage of the larger screen and keyboard available on the desktop machine. The programmer could use the testbed system to examine variables, display trace data, and save datafile snapshots.

If we had not been compiler-based, our testbed system would not have been so close to the actual data collection software. This closeness allowed us to maintain a high code quality as most problems would be in the common code and thus once fixed would stay fixed. In addition, the small amount of per-platform code made it easy for us to envision porting the environment to other platforms.

THE DECOMPILER

Because of our compiler-based approach, we guaranteed that source code would always work across product releases. We were thus free to change (presumably improve) the object format as much as we liked.

To make this approach work, we had to plan for the case where someone had a working program but had lost the source code. We thus wrote a decompiler, which did the obvious.

This was a valuable development tool as well. With it, all language changes had to be made in two places (the compiler and decompiler). We thus had both a means of testing their implementations and confidence that they were implemented properly. (Object format changes were made in three places: the compiler, the interpreter/testbed, and the decompiler.)

THE QUESTION-AND-ANSWER PROGRAM

One of the HP-94's major markets was data collection, typically either counting static inventory or recording transactions. Most of these applications are pretty much the same, with just the names of what is being counted and the acceptable values changing from one to the next.

We took advantage of this commonality and wrote a question-and-answer program. This program asked for the names, value ranges, and data storage formats and wrote the program to do the collection. It was our goal that 2/3 of all programs could be completely written using just this method and, in hindsight, it appears that we could have met this goal.

Of the other 1/3 of all applications, many could benefit from being started with the question-and-answer program and would only require minor tailoring.

ASSORTED UTILITIES

We also included a variety of special-purpose utility programs. These were:

- Programs that implemented the Kermit and Xmodem protocols to transfer data to and from the handheld.

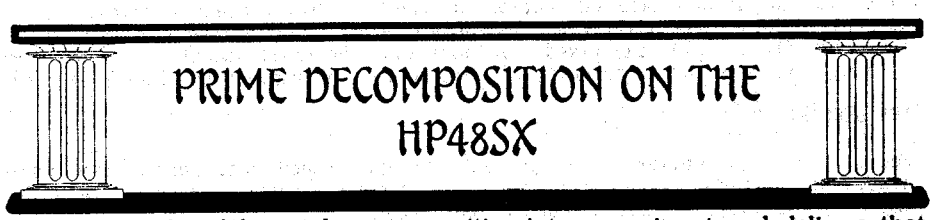
- A program that build the special data format required by the HP-94 for files downloaded to its file system.

- A program that could construct file system images for creating ROMs.

- Assorted demo programs and documentation files.

NEXT

The next article will cover the unique features of the language and how applications were put together. The final article will cover the corporate history and its interactions with HP.
※



The program listed here takes any positive integer as input and delivers that number as a product of primes. The original version was my first exercise in 48SX programming and it was pleasing to end up with a reasonably efficient and elegant program.

Variables used are as follows.

n	is the input number or its current residue.
pdc	is the prime decomposition.
len	is the length of the last display line.
div	is the current divisor.
pwr	is local to subroutine REC and is the exponent of the lastfactor.
nl,n2	are stores, local to program INC.
i	is a FOR counter in program PRDC.

To use the program proceed as follows.

1. Hit the VAR menu key for PRDC.
2. See the prompt "Enter number for primedecomposition". ENTER a positive integer.
3. Read the prime decomposition from the display. The display is complete when it ends with a full stop.
4. To re-run the program, hit the menu key again.

Examples.

1. Key in 223092870, hit ENTER.
   See 1*
   then 1*2*
   then 1*2*3* etcetera
   then 1*2*3*5*7*11*13*17*19*23.
2. Key in 3195731, hit ENTER.
   See I*
   then 1*7^4*
   then 1*7^4*11^3.

Performance. This program is built for speed to reduce the waiting time for very large numbers. To this end no external subroutines are called except when a factor is found, even though this involves some repetition of code. I found that routine calling of subroutines slowed the program considerably. I obtain the following processing times for three large primes;

368,507 : 13.6 secs 3,948,173 : 38 secs 31,465,099 : 111 secs

These times are about two thirds of the times taken by an early version in which succesive odd numbers were used as divisors. The presented version uses divisors which are zero (mod 2,3,5,7). If anyone can suggest other ways to streamline the program I should be delighted to hear from them. My address is not in the Memberpack because I re-enrolled only recently, so it is given below.

Peter Gatenby, [#030]

20, Hathaway Drive, WARWICK, CV34 5RD. (0926)497516.

P.S. 9,999,999,967 takes 292 minutes to display as a prime.

Notes on the listing.

PRDC. PRime DeComposer, the main program.
Sets the number display format to Standard mode.
The input number is stripped of quotes. A copy is stored in 'n' and it is used to start the display, with " -=".
Variables 'pdc', 'len' and 'div' are initiated. Subroutine REC is called with 1 on the stack to start the output product.
If n=O(mod2) then TEST is called to find the exponent of 2 and add it to the output product.
Odd numbers 3 to 11 are tested similarly with the residue of n in a 2-step FOR loop.
The list INCS (see below) is piled on the stack by OBJ and its SIZE is DROPped.