C-sick?

or

How To Convert From SPL To C Without Making Waves

by

Stan Sieler

Stan Sieler has had seven years of experience on the HP3000, including over four years with Hewlett Packard in the operating system laboratory for the HP3000. He has been professionally involved with programming since 1972, and is currently a member of the Adager R & D team. He holds a BA degree in Computer Science from the University of California, San Diego.

This paper was made possible by Adager, S.A.

---

0. Introduction

For many years, SPL has been the language of choice for implementing efficient, high level programs on the HP3000.

Before I offend TOO many readers, let me mention that I am primarily referring to programming languages used for major programs that must be fast and maintainable. I realize that every HP3000 language has been used in such projects at various times, and for many people using something other than SPL is the preferred choice (for a variety of reasons, e.g.: not having any programmers who know SPL, or needing COBOL's packed decimal arithmetic).

During the early years of the HP3000, only two reliable languages existed: SPL and FORTRAN. After COBOL II was released, the number grew to three, but SPL remained preeminent. Of the three, SPL was clearly the most tightly coupled to the HP3000. That coupling, plus its ALGOL-like structure, made it the most powerful of the available languages. (That the power could be abused is true, but not relevant here.)

Eventually, other languages started appearing for the HP3000: RPG, BASIC, SPLII, APL, Pascal, FORTRAN 77, Business BASIC, and Fourth Generation Languages. But all of these had various drawbacks, and never managed to displace SPL.

However, the HP3000 is aging, compiler technology is improving, and computer architectures continue to change. As a result of this, many programmers on the HP3000 are looking for a language that will work well today, and will be available on tomorrow's computers as well. SPL clearly works very well today, but its death knell has sounded. HP has made it clear that a native mode SPL will not exist for Spectrum, the successor to the HP3000. What language, then, can we use instead of SPL to achieve the twofold goals of efficiency and portability?

Pascal immediately comes to mind. Unfortunately, Pascal was designed as a teaching language, and lacks a number of features that would qualify it as a replacement for SPL. Within HP, it is not Pascal that is being used to write their next operating system but MODCAL, a MODified version of pasCAL which overcomes most (but NOT all) of the weaknesses of Pascal. But...this language is not available to the HP3000 programmer today.

FORTRAN/77 also comes to mind, but this language has three strikes against it. First, many programmers will look at the name FORTRAN and dismiss it...this isn't fair, but it happens. Second, it is a new compiler on the HP3000 (released in late 1985) and has some reliability problems at present. Third, it isn't clear that FORTRAN/77 compilers will be available on most other machines. This third point, portability, is important for people who might want to move their programs to non-HP computers in the future.

What other languages are available on the HP3000? Until recently, the answer was: none. In 1986, two companys have announced C compilers for the HP3000.

What is C? Can it aspire to be the replacement for SPL?

To quote from the C bible ('*The C Programming Language*' by Kernighan and Ritchie), "C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators". While I have found the phrase about "economy" to really mean: "we hated to type much, so there is a lot of cryptic stuff in C", I do generally agree with the authors. C provides the programmer with a lot of power (and it is even easier to abuse than SPL!).

C compilers exist for most computers, ranging from the tiny 6502 microprocessor (the Apple II), to the mighty Amdahl 580. Now, C is on the HP3000.

The ease with which the C compiler, and the UNIX operating system, has been ported from the PDP-11 to many different computers says a lot for the portability of C.

The major strike against C on the HP3000 is the compiler reliability question. C is a new language on the HP3000, so we can expect some problems with the compilers initially. However, I think that the compilers will mature quickly. Unlike the FORTRAN/77 compiler, which was probably new code, I expect that both of the C compilers were ported from different machines, which would result in a compiler with less new code to debug. Additionally, I wouldn't be surprised if there is a larger number of programmers who want to use C rather than FORTRAN/77, so the C compilers will be "tested" more quickly.

For this paper I happened to use the C from CCSC. This should NOT be taken as an endorsement for CCSC's C over Tymlab's C, since I currently do not have enough information to make such a decision.

1. Introduction to C

This section is designed to introduce the reader to C, so I had better mention something about coding style in C. As a relative beginner in C, my C coding style still resembles my SPL coding style. I have noticed in the past that when I learn new languages, I often try to impose the style of an older language on the new one. This rarely works to anyone's satisfaction. It is important when using a language to try to code somewhat like the rest of the users of the language, so you can read their code, and vice versa. This shouldn't be taken to extremes...remember that the refrain "but everybody does it" does NOT necessarily make something right. One of the most important aspects of coding is to pick a style and stick with it.

Programs in C consist of one or more functions, including one special function called *main*. C refers to "function" in place of SPL's *procedure*, but the word "function" is actually never used in the language.

An example program that calculates the modulus-11 checksum for a bank account number is shown on the next page, with every line numbered as an aid to referring to them in the text. (The algorithm was extracted from the back of the V/3000 manual.)

Converting SPL to C

```
#include stdio.h.ccsc
#include stdefs.h.ccsc

short
   weights     [6] = { 2, 3, 4, 5, 6, 7 };

/*************************************************************/
short calccheck11 (ptr, len);
   char *ptr;
   short len;
  {
   short
      rslt,
      winx;

   printf ("MOD11 (%s) --> ", ptr);

   winx = 0;
   ptr += len - 1;        /*point at units digit*/

   rslt = 0;
   while (--len >= 0)
      {
      rslt += (*ptr - '0') * weights[winx];
      winx = (winx + 1) % 6;
      ptr--;
      }

   if ( (rslt = 11 - (rslt % 11)) == 11)
      rslt = 0;
   if (rslt == 10)
      return (-1);
   else
      return (rslt);
   }
/*************************************************************/
main ()
  {
   char  num [10];
   short rslt;

   while (TRUE) {
      printf ("Enter 8 digit number: ");
      scanf ("%s", num);
      if (num[0] == '/')
         terminate ();

      rslt = calccheck11 (num, 8);
      if (rslt < 0)
         printf ("Unable to determine mod11 for %s\n", num);
      else
         printf ("Rslt = %d\n", rslt);
      }
   }
```

The outer block of a C program is the function *main*. This one starts at line 40 and declares two local variables: an array of characters called *num* (index from 0 to 9), and a 16 bit integer called *rslt*. Line 41 contains a *{*, which is C's version of SPL's *begin* (remember...the authors of C hated to type!).

Line 45 is the start of a *while* loop, but notice two differences from SPL here: the boolean expression is parenthesized, and there is no *do* after the boolean expression, just a *{*.

Notice the uppercase *TRUE* on line 45. Most C compilers differentiate between uppercase and lowercase, unlike SPL. This is unfortunate, as it allows variables like *i* and *I*, which are NOT the same variable.

The traditional C usage is: most things in lowercase, identifiers that are macros (defines) are in uppercase, and when variable names are composed of more than one word, make the first character of each word uppercase (e.g.: *CardCounter*). *TRUE* in line 45 was uppercase because it was a define (found in the file stdefs.h). This is the area where I will depart from the existing C conventions. In SPL I use lowercase everywhere, with procedure names being verbs and variable names being nouns. I intend to apply the same rationale to C. Thus, after this paper is finished, I will edit stdefs.h to say:

```
#define true 1
```

instead of:

```
#define TRUE 1
```

(Note: *TRUE* is defined as 1 in stdefs.h...this could cause some problems when interfacing with other HP3000 languages, like SPL, where *true* = -1.)

C has NO I/O defined within it, just like SPL. And, just like SPL (or ALGOL), this was a serious mistake. Luckily, the users of C have developed a relatively standard set of functions to provide I/O capabilities. Most C compilers come with these functions. For example, line 46 will probably compile and run on every C compiler. The standard *printf* function is a lot like the Pascal *write* statement. It provides a variety of formatting capabilities. Line 55 shows two of them: the *%d* in the quoted string is a signal to *printf* that it should grab the parameter after the string (*rslt*), format it as an integer, stick it into the string "Rslt...", and print the result. Since *printf* doesn't do carriage control for you (just as *write* doesn't), the *\n* is a request to *printf* to do a CR/LF at the end of the line. Thus, any *printf* whose formatting string ends with a *\n* is similar to Pascal's *writeln*.

On line 48 note two differences between SPL's *if* and C's *if*. First, there is no *then*. Second, the boolean expression is parenthesized. The lack of a *then* doesn't bother me because I have always maintained that *then* is "syntactic sugar" (something that really didn't have to exist), which is why I always code my SPL *if* statements as:

```
if .... then
  ...
else
  ...
```

rather than as:

```
if ...
then        (or then ...)
  ...
else        (or else ...)
  ...
```

If you feel you REALLY miss the *then*, and just MUST have it, why, then, C has the answer for you! Just include the following line in your C programs:

```
#define then
```

This is equivalent to SPL's:

```
define then = #;
```

Line 48 shows one of the quirks of C, caused by people who hate to type. SPL, Pascal, and ALGOL use := as the assignment operator, and = as a comparison operator. C uses = as an assignment operator (like FORTRAN and BASIC), and == as the comparison operator. THIS CAN CAUSE A LOT OF MISTAKES FOR SPL PROGRAMMERS MOVING TO C!

Consider the following code fragment:

```
if (a = b)
  terminate ();
```

What this code does is: set *a* to the value of *b*; then, if *a* is non-0 (after the assignment), call *terminate*. The SPL programmer who glanced at the code would have said it meant: if the value of a equals the value of b, then terminate. Mistakes of this type will happen more frequently if the SPL programmer succumbs to the temptation to use the null define *then* (shown above), because then that statement in C would have looked like:

```
if (a = b) then
    terminate ();
```

Or, in other words, it would have looked so similar to SPL that no mental flags would have been raised saying "look at me closer, I am different"!

Comments in C are different, and similar to Pascal's (* and *). A comment is anything following /* and before the next */. Most C compilers do not allow nested comments, so the best practice is to not use them.

Line 8 shows the declaration of a function, *calccheck11*, which will return a 16 bit integer as its result. Notice that the word "function" is not used. Again, if you feel more comfortable, you could do the following:

```
#define function
```

and then say:

```
short function calcheck11 (ptr,len);
```

C often shows clues to the machine of its origin. It has two interesting operators, pre-increment and post-increment, that were one word instructions on the PDP-11. Line 27 shows an example of post-decrementing. Line 23 shows an example of pre-incrementing...it says: take the value stored in len, subtract one, store that number back into len, and use it in a comparison against 0. The basic pre/post operators are:

++foo	increment *foo*, use new value in expression.
foo++	increment *foo*, use prior value in expression.
--foo	decrement *foo*, use new value in expression.
foo--	decrement *foo*, use prior value in expression.

As we end our very brief introduction to C, let us look at line 20. The += shows that C originated before compilers were smart enough to optimize expressions like:

```
a[i+j] = a[i+j] + k;
```

Instead, C lets the programmer do the optimization:

```
a[i+j] += k;
```

which means the same thing (barring side effects while evaluating *i* and *j*). It's also easier to type.

2. SPL to C Translation

This section discusses techniques for translating SPL to C. It has two major components: the easy part and the hard part.

2.1 Easy Automatic Translation

Much of the conversion of an existing SPL program to C can be very easily automated by your favorite editor. In the following examples I use Robelle's QEDIT, but many other editors would work as well.

Note: the following lines are numbered to facilitate referring to various lines.

```
1. set shift down 2
2. pq down @
3. c "<<"/*" @
4. c ">>"*/" @
5. c "$include"#include"@
6. c "procedure"(S)""   @
7. c "integer"(S) "short" @
8. c "byte"(S)    "char" @
9. c "real"(S)    "float" @
10. c "logical"(S) "unsigned short" @
11. c "double"(S)  "**TEMP**"@
12. c "long"(S)    "double" @
```

```
13. c "**TEMP**"long" @
14. c ":="~"@
15. c "<="<~"@
16. c ">=">~"@
17. c "="==" @
18. c "~"=" @
19. c "<>"!="@
20. c "then"(S) "" @
21. c "do"(S) "" "while"(s)
22. c "begin"(S) "{"@
23. c "end"(S) "}" @
```

Line 1 tells QEDIT that I will be downshifting SPL style text. Option 2 says: don't change anything within double quotes (").

Line 2 downshifts all text within the source file EXCEPT quoted strings.

Lines 3 and 4 convert SPL comments to C comments. You will have to manually search for SPL comments using the new "throw away the rest of the line" comment character *!* (introduced in MPE V/E SPL). With most editors, this might change a few occurrences of "<<" and ">>" that you did not intend to change (for example, if you had "<<" in a quoted string, then you probably did not want it changed to "/*").

Line 5 changes references to SPL $include files to the C format. Note: be sure to change the include files too!

Line 7 changes SPL 16-bit integers to C 16-bit integers. With CCSC's C, *int* and *short* both mean 16-bit integer, but I chose *short* because I would guess that *int* might mean 32-bit integers on Spectrum...it is an aspect of C that is NOT well defined.

Lines 8 through 12 change the common SPL variable types to their C equivalents. Note the fancy editing in lines 11, 12 and 13. C uses the terms *double* and *long* EXACTLY backwards from SPL. Thus, the simplistic approach of saying: change *double* to *long*, and then change *long* to *double* would merely change all *double* and all *long* to *double* with no more *long*s in the file!

NOTE: changes like those in lines 10 and 12 can cause problems if the resulting line is too large for your text file.

Lines 14 through 19 effect the following changes:

```
SPL   :=  <>  =
 C    =   !=  ==
```

The extra trick here is to avoid changing things like <= into <== by accident. Again, these change commands could accidentally change items within quoted strings that you did not want changed.

Lines 20 and 21 get rid of the unnecessary *then*s and *do*s. Note that line 21 is qualified so that it only affects *do*s that are on the same line as *while*s, thus not touching *do/until loops*.

2.2 Easy Manual Translation

SPL *define*s can easily be converted to C *#define*s. For example:

```
define  twox   = (x + x) #, fourx  = (4 * x) #;
```

would be changed to:

```
#define twox   (x + x)
#define fourx  (4 * x)
```

Similarly, *equate*s in SPL are easy to handle as *#define*s in C. Another method of handling *equate*s might be chosen for some usages. For example, consider an input parser that returns the type of token found, via equates, in a variable called *iclass*:

```
equate unknownv = 0,
       tokenv   = 1,
       numberv  = 2,
       stringv  = 3;
integer iclass;          <<always is: unknownv...stringv>>
```

In C this could be changed to use *enum*, which is similar to Pascal's enumerated types:

```
enum scanner {unknownv, tokenv, numberv, stringv};
enum scanner iclass;
```

NOTE: not all C compilers support *enum*.

2.3 Harder Aspects Of Translation

Some of the conversion of an existing SPL program to C can be difficult. Consider some of the following constructs of SPL:

- address equation at variable declaration;
- subroutines within procedures;
- *move* statement;
- *scan* statement;
- *assemble* statement;
- register usage (*push* and *pop*);
- entry points;
- if expressions.

Some of these constructs will be very difficult to move to C, particularly *assemble*.

Although C does offer a form of address equation with the *union* statement, you may not need to use it since address equation is not used often in high-level SPL programs.

Subroutines pose a large problem. C does not allow functions within functions, which would have been a nice solution. If a short subroutine has call-by-name parameters (or call-by-value ones which are not altered), then it might be converted into a macro. Consider the following SPL example subroutine:

```
integer subroutine min (x, y);
      value  x, y;
      integer x, y;
  begin
  if x < y then
    min := x
  else
    min := y;
  end <<min sub>>;
```

It can be converted into the following macro:

```
#define min(x,y) ( (x) < (y) ? (x) : (y))
```

Then, calls to *min* will work as before.

Sometimes, a subroutine can be converted into a separate function. Ideal candidates for this are subroutines that use none of the local variables of the surrounding procedure.

If a subroutine only uses a small number of a procedure's local variables, then you might consider making it a separate function and pass those variables in as additional parameters.

The SPL *move* statement can be translated into a function call:

```
move p1 := p2, (10);
```

translates into:

```
move (p1, p2, 10);
```

This, of course, assumes that the original SPL 2move was a byte-oriented, not word-oriented. Note that you may have to write a *move* function yourself in C. Alternatively, the following macro accomplishes the same SPL move:

```
#define move(p1, p2, len) {short ktr;  for (ktr=0; ktr < len; ktr++) \
                                        p1[ktr] = p2[ktr]; }
```

(The backslash (\) indicates that the line is continued on the next input line.)

Or, as another alternative, some C compilers provide the routine *memcopy* for moving bytes:

```
#define move memcopy
```

The SPL *scan* statement can be handled in a similar manner.

There is almost no hope for the SPL *assemble* statement. But, luckily, there is almost no reason for it to be in your SPL programs anyway.

Similarly, any SPL code that explicitly accesses any hardware registers (e.g.: *Q, DB, X, S, DL*) will have to be examined by hand.

Some C compilers for the HP3000 do not support entry points. While most programmers do not use entry points, and therefore will not miss them, some do. For example, many HP programs in PUB,SYS have documented entry points (e.g.: EDITOR) that drastically affect their behavior. I have a policy of having a HELP entry point in every program I write, so that a new user only needs to say:

run tapedir, help

to get safe help information (such programs ALWAYS terminate after delivering the help information).

The C *if* statement differs in an important way from the SPL *if* (besides in appearance). In SPL, the *then* part will be executed if the boolean expression is true...but "true" is defined as "bottom bit is a 1". If C, the *then* part will be executed if the expression is non-0. This difference can drastically affect programs. Consider the following SPL code that checks to see if a variable is an odd integer:

if logical(k) then

If this were translated to C without thinking as:

if (k)

then it means: "if k is not 0". Instead, a proper C translation would be:

if (k & 1)

which does a "bitwise and" with *k* and 1, returning 0 if the value was even and 1 if the value was odd.

3. Conclusion

C is a very powerful language, and exists on many different computers. Because of this, I feel that it is an excellent candidate for replacing SPL.

I am not abandoning SPL right now, but I am going to start coding SPL with the thought in the back of my mind that someday, soon, I will be moving to C. This knowledge will provide an incentive to avoid those features of SPL that ARE hard to convert to C (or to ANY other language). Additionally, I plan to work further in C on both the HP3000 and on various microcomputers in an effort to get "up to speed" in it as fast as I can.

Remember...keep an eye on C...it may be the language in your future!