Jack Crenshaw
Tampa, Fla.

Jack Crenshaw is an independent consultant and a contributing editor to *Embedded Systems Programming*. He wrote his first computer program in 1954, and has been involved with software design, development, and methodologies ever since. He did much early work in the space program and has developed many analysis and real-time programs. He is the author of the “Programmer's Toolbox” column in *Embedded Systems Programming*. Crenshaw holds a PhD in Physics from Auburn University.
OVERVIEW

In the February, 1992 issue of Embedded Systems Programming, a new column, Programmer's Toolbox, was introduced. The name was not chosen at random: The intent of the column is to provide useful tools and techniques that readers can apply to their own problems in embedded systems development. The idea is to have a library of reusable modules, written in several languages, that others can use to avoid re-invention of the wheel and improve their productivity and software quality. In this paper, I'll describe the thoughts behind the column, the software tools that have already been presented, the ones yet to come, and the ultimate goal of the effort. A successful effort will require support from the user community. Part of the message of this paper is a plea for help and feedback.

BACKGROUND

What is software? Some say it's an art, some a craft, some an engineering discipline, and some a science. There are even those who want to turn the act of software development into an automated, rote process like baking bread or smelting iron ... the Japanese concept of the Software Factory. I don't think that's ever going to happen. Compare the history of software to that of, say, gunmaking. Before Eli Whitney introduced mass production to that industry, and helped begin the industrial revolution in America, every gun was custom built, hand-crafted by a master. Today, software is in that state: It's a craft, and it requires craftsmanship. [Note for those into political correctness: No gender bias is intended in the use of the word ... there is no such word in the dictionary as "craftersonship." In the interest of brevity and clarity, I hope you'll allow such words and accept that no offense is intended.]

But craftsmanship is not enough. Even Stradivarius couldn't have built great violins with a ball-peen hammer and a bent screw-driver. Good tools are also needed. All other things being equal, the more tools in your toolbox, the better (as long as you know what they're there for!)

In their seminal book, "Software Tools," Brian Kernighan and P.J. Plauger suggest that one of the reasons that some programmers write bad programs is because no one's ever showed them examples of good ones. They said, "We don't think that it is possible to learn to program well by reading platitudes about good programming." Instead, their approach was to show by example, presenting useful software tools developed using good programming practices. By doing so, they killed two birds with one stone: They added tools to our toolboxes, while at the same time teaching good programming skills.

Because Kernighan and Plauger (K & P) used Unix, the tools they presented had very much a Unix flavor and were primarily text-based tools ... the kind of thing you develop using
getchar and putchar. Though somewhat dated now (the tools included a forerunner of ed, for example), the K & P tools took on a life of their own. They were translated to many languages and platforms, and a users’ group was formed to help distribute, standardize, and improve them. What’s more, the style introduced by K & P was duplicated in other programmers, thereby meeting their goal of disseminating good software practices. These tools may well have started the now-fashionable trend towards reusable software modules. Certainly the Software Tools have been reused as often as virtually any other modules in the world.

I won’t claim to be able to duplicate K & P’s success, but the tools in Programmer’s Toolbox are offered in very much the same spirit. As was the case with Software Tools, I have a dual purpose: Not only to provide modules that can be reused in a variety of languages and platforms, but also to disseminate my own ideas as to what constitutes good programming practice. I don’t expect everyone to always agree with these ideas, but they can at least serve as a starting place for discussion.

The main difference between the tools is the intended audience and applications. Whereas K & P were providing production programs for Unix-like, text-based applications, we are interested here in tools that can be used for the development of embedded systems. What’s more, the nature of a column necessarily restricts the size of the tools to snippets of code ... subroutines or even code fragments, that can be incorporated into embedded systems software. If the tools presented by K & P were bandsaws and drill presses, you might think of my tools as a good set of socket wrenches or jeweler’s screwdrivers. The tools will be smaller, but the goal is the same. The tools in our toolbox are not standalone programs, but rather small, oft-used and oft-abused subroutines that can be dropped into your embedded programs. In other words, a library of reusable modules. Ultimately, what you should get out of this is a well-equipped toolbox of useful routines. If things go according to plan, you will also hone your programming skills in the process.

PROGRAMMING STYLE

Now, good programming practice is an ephemeral thing. No two programmers agree 100% as to what it is. What you really get is my own view of what constitutes a good tool and a good use of it. The things I’ll present here come from lots of years of experience, but I don’t expect you to always agree, nor do I expect to always be right. Consider this a plea for feedback, comments, and improved methods. You won’t find me obstinate without good reason. I’m never too proud to steal someone else’s code or ideas when they’re better than mine.

I tend to write software in very small pieces ... sometimes one-line subroutines. I also tend to use straightforward and easily readable code, rather than resorting to clever programming tricks that I can’t read a week later. I believe with Dijkstra that speed comes from correct algorithms, not from tricky code. Maybe these are habits I picked up from my days of maintaining someone else’s spaghetti FORTRAN code. I learned very early that it’s a lot easier to go back and replace or optimize a subroutine that’s proven to be a bottleneck, rather
than to retrofit structure to a monolithic, all-in-line program, or try to figure out what a particularly obscure bit of code is supposed to be doing.

Unlike most assembly-language programmers, this tendency is even more pronounced in my assembly code, where subroutines cost only the price of a simple CALL and RET statement. In general, I find that I get enough speedup by using assembly in the first place, that I don’t have to go to extremes to squeeze out that last ounce of performance. All of this works to our advantage in the format of a library of tools, since the toolbox that’s evolving consists of small and easily readable subroutines that can be readily plucked out and dropped into your own software.

You’ll find, however, that I sometimes violate the simplicity rule, particularly in the vector and matrix operations, where the subroutines tend to be used quite often, and the potential for optimization is high. In a couple of columns, I presented some algorithms that were hand-optimized to squeeze maximum performance using strength-reduction on the indices. This was done mainly for the benefit of the assembly-language programmers, who have enough problems getting the indexing right, and typically don’t have time to figure out the optimal way to step through the indices.

THE TOOLS

So far, the items that have been covered in Programmer’s Toolbox have included the following tools and techniques:

- Assignment of global constants
- Simple and ubiquitous functions like abs and sign
- Angle and other conversion routines
- Trig functions with exception handling
- Hyperbolic functions, power function, etc.
- Vector and matrix operations, including conformant arrays
- Cartesian/Polar conversions, Euler angles, etc.
- Optimization of routines via strength reduction

In addition to the contents of the column, there have been companion articles almost every month. These are important parts of the library, because they include algorithms for the efficient implementation of such ubiquitous functions as square roots, sine/cosine functions, etc. In a higher-order language, these functions are provided, of course. But if you’re writing for an embedded processor with little or no library support, the algorithms can be invaluable. Though I’m not a stickler for high performance in most cases, these routines were optimized for that purpose since they tend to be used very often, and in time-critical code. You’ll find that the algorithms, particularly for the most often-used routines, are about as efficient as you’ll find anywhere, and many of them have never been published before. No point in creating a library if the tools are mediocre.

You’ve probably noticed that the toolbox has a decidedly math-oriented flavor. This is not accidental. I find that most programmers (in fact, most Americans) don’t feel very
comfortable with math, or have good backgrounds in that area. But face it: A weakness in math can be hazardous to your career. Most embedded systems require the implementation of things like data conversion and compensation, digital filters, feedback control systems, etc. If you’re going to work with embedded systems, you’ll be at a big disadvantage if you can’t handle the math. In other words, math is the area where embedded systems programmers need the most help. The tools and techniques offered in the Toolbox are intended to make life easier in that department. You’ll get tested tools and techniques that you can incorporate into your own software. Who knows? You might even learn a little math!

There’s no way I can present the entire contents of the columns here, so I’ll just cover the high spots, and try to give you a flavor for the kinds of things we’ve done. For more details, I’ll have to refer you to the columns themselves.

**DEFINING CONSTANTS**

One interesting problem involves the use of fundamental constants. I can’t tell you how many programs I’ve seen with five or six different values for \( \pi \) or the acceleration of gravity. This can lead to extremely subtle and hard-to-find errors when variables that are supposed to be equal turn out otherwise. The problem gets trickier when some constants depend on others, as in the case of the angle conversion \( \frac{180}{\pi} \). The result is a set of constants that are not internally consistent. Over the years, I’ve wrestled with the problem of how best to initialize a self-consistent set of constants and make them globally available. I found the solution in the Unit mechanism of Turbo Pascal (which works in much the same way as that of the Ada package). The general idea is shown in Listing 1.

There is no way to accomplish the same result in C, because C does not support the runtime initialization of packages. I tossed out the challenge to readers to come up with an equivalent mechanism in C, but got few takers. One reader’s solution is presented in the November column. It requires an explicit initialization routine to be called, but otherwise is quite tolerable.

In C++, on the other hand, the problem tends to go away completely, because this language supports constants defined by expressions. So, ironically, does assembly language.

**DATA CONVERSIONS**

Example of both the use of the defined constants and my one-line subroutine style are given by the two angle-transformation routines shown in Listing 2. I blush to admit that it took me twenty years to figure out how handy such routines are: before that, like most programmers, I tended to write the conversions in line. I suppose my myopia sprang from a misplaced concern for efficiency. But I missed the main point, which is that such conversions are normally done only for I/O anyhow, and efficiency is not an issue since the program will be I/O bound.
FUNDAMENTAL MATH FUNCTIONS

If you’re programming in a high-order language, such fundamental routines as sine, cosine, square root, etc., are provided for you by the compiler’s math library. But what if you’re writing in assembly language? Then you’re on your own. To fill the gaps, I’ve provided two sets of functions, some in articles and some in the column. The more difficult and tricky functions such as square root, sine/cosine, arctangent, etc., were given in articles. In the column, I’ve provided the “functions that time forgot” … things like the real mod function, abs, sign, arcsine, arccosine, integer and real exponentiation, etc. Some languages, notably FORTRAN, support most or all of these in their libraries. Many do not, so I’ve provided them.

The word “tools” does not always imply “code.” Sometimes it’s not enough to have subroutines; you also need to know what to use them for. In the column that gave the trig functions, I also provided some useful rules for using them to solve geometry problems involving triangles.

In the case of the fundamental functions, I’ve violated my “Keep It Simple” rule and tried to provide the most efficient implementations I could find. The reason should be obvious: These functions are used extremely often, and inefficiencies or inaccuracies in them can hurt an otherwise well-written program. Many of the algorithms were developed specifically for Embedded Systems Programming, and published there for the first time. They are about as good as you are likely to find. For the benefit of the embedded systems audience, I’ve provided integer implementations that are very quick.

BELT AND SUSPENDERS

Even library subroutines can be annoying: Sometimes they provide almost, but not quite, what you need. In particular, library routines are notorious for not having very good exception handling. Pass a zero to the square root routine, and you’ll get zero back. Pass it an “almost zero” that happens to be negative, and you’ll get a fatal error. Fatal errors are not smiled upon in embedded systems such as nuclear power plants, so we writers of embedded systems programs have to go to special lengths to deal with situations that aren’t supposed to happen, but do, because of roundoff errors. The functions provided in the toolbox all have tests to deal with boundary conditions that can be troublesome. The square root routine, for example, simply returns a zero for any negative input. This in itself can also lead to error, since it can mask a real problem, but in a checked-out, production system, it’s better than a system crash.

VECTORS AND MATRICES

Among the most powerful of math tools are those that perform vector and matrix algebra. No math-oriented toolbox would be complete without them. Over the past several months, I’ve been conducting an ongoing “tutorial” on this important subject. So far, we have routines to perform most of the standard operations: vector add, subtract, dot and cross products, etc. Also the matrix routines add, subtract, and multiply (of both scalars, vectors,
and other matrices).

In this area, we have one problem: Neither of the languages C or Pascal supports the idea of conformant arrays, that is, arrays whose dimensions are defined when the procedure is called. This is one of the main reasons why FORTRAN is still in use for scientific programming. To get around this limitation, the toolbox includes two different sets of routines. One operates only on vectors of three dimensions, and 3x3 matrices. The rationale is that since these constitute about 90% of the cases we deal with in practical programs, there’s no point in burdening the routines with the extra overhead required to handle the general case. The second set uses dynamic memory allocation to support vectors and matrices of any size. This approach does have its drawbacks: The code is considerably less transparent, because the languages also don’t support the “variable dimension” features of FORTRAN. To circumvent this problem, we allocate matrices as linear arrays, and perform our own index management.

STRENGTH REDUCTION

This situation has an upside and a downside. The upside is that, by dealing with the indices explicitly, we can effect some considerable performance savings. In vector routines and many cases of matrix routines as well, it’s possible to organize the code so that at least one index is stepped sequentially through the array. This is the idea behind optimization via strength reduction. It’s done automatically by optimizing compilers, but you’re on your own in assembly language or with non-optimizing compilers. What’s more, the ability of even an optimizing compiler to do its thing depends upon the way the routines are written, the loops are nested, etc. We spent a couple of months going through strength reduction in all its gory glory, primarily for the benefit of the assembly language programmers. Unless you’re a true genius, you’re not likely to hit on the right combination by accident, and few assembly programmers have the time or inclination to sit down and work out the optimizations by hand. So I did it for them. The resulting subroutines are horribly difficult to read (as is the output from an optimizing compiler), but they do get the job done, and done well.

OBJECTIVELY SPEAKING

In the beginning, I very carefully avoided the use of object-oriented techniques, since the whole point of the toolbox was to provide common algorithms that could be used in any language, including assembly language. Some day we’ll all have C++ compilers designed explicitly to produce tight code for embedded systems, but that day is just plain not here quite yet.

In the case of the vector and matrix code, though, the temptation to resort to objects was overwhelming, and I succumbed to it. We ended up with two versions of these routines: one written in classical fashion, and one written with the vectors and matrices as objects. The second version has considerable advantages if you have access to an OOP compiler.

As a matter of fact, before we’re finished, we’ll have no less than three versions of these tools, and therein lies a tale.
LANGUAGE ISSUES

For years, my favorite language has been Turbo Pascal. I won't go into the reasons here, because we border on religious issues. But, especially for the purposes of the column, the fast compilation speed of Turbo lets me write my articles faster. When I began the column, it was my intention to make the toolbox tools as language-independent as possible. Yes, I used Pascal in my examples (and got chided for not using more C), but I also emphasized that they were to be used more as pseudocode templates than as production software. I still intend for the tools to be usable by assembly programmers, which is why I devoted so much time to strength reduction to optimize array processing.

However, the universal nature of the algorithms and tools began to fuzz over quite a bit when I introduced the idea of objects, and things have continued to diverge ever since. Modern Turbo Pascal, of course, supports objects very nicely, thank you, but these days, when someone says, "objects," almost everyone within earshot thinks, "C++.”

I had known that it was just a matter of time until I had to deal with C++ face on, because, unlike C, it has so many features that are useful in scientific and embedded system programming. Of all the languages, including Ada, C++ seems to me to offer the best support for truly reusable software. Already, we're seeing major software packages that extend the language capabilities to perform scientific and statistical mathematics. In fact, before I ever began using C++, I made this prediction, which I'll still stand behind:

C++ will soon replace FORTRAN as the language of choice for scientific programming.

One thing I hadn't counted on when I began delving into C++ was the way my solutions would diverge from those I had used with Pascal, even Pascal using objects. The primary reason is operator overloading.

OBJECTS AND OPERATORS

In my old FORTRAN days, we had very conventional subroutines that operated on arrays passed as arguments through a calling list. FORTRAN, of course, cooperated with this by supporting pass-by-reference and variable dimensions. Over the years a de facto standard arose, which said that input arguments should be first, output arguments next, and any indices or flags last. So any FORTRAN programmer who saw a subroutine call like:

```
CALL ADDV(A, B, C, 6)
```

could tell you that this routine would add six-dimensional vectors A and B, and store the result in C. No prototype needed!

Suppose we had to implement a vector equation such as:

```
x = y - 2z + (y * z)(y * z)
```

The FORTRAN code might look something like:
CALL VCOPY(Y, X)
CALL VCOPY(Z, TEMP)
CALL VSSCALE(2.0, TEMP, TEMP)
CALL VSUB(X, TEMP, X)
CALL VCROSS(Y, Z, TEMP)
CALL VSSCALE(DOT(Y, Z), TEMP, TEMP)
CALL VADD(TEMP, X, X)

In a way, what we did here was to define an abstract machine for vectors, with each subroutine call representing an operation in the abstract machine. This being the case, we’re in effect writing assembly language for the abstract machine. Going to ordinary C or Pascal doesn’t change this; we’re still going to be doing abstract assembly language. In fact, in non-object Pascal, the code would be identical to the FORTRAN fragment above. We need only delete the CALLs and add semicolons!

Using Turbo Pascal with objects simplifies things only a bit, by avoiding the need to define the target variable ... it’s defined by the object-oriented syntax:

```pascal
x.Assign(y);
Temp.Assign(z);
Temp.Scale(2.0);
x.Subtract(Temp);
Temp.Cross(y, z);
Temp.Scale(Dot(y, z));
x.Add(Temp);
```

C++, on the other hand, opens up a whole new universe: high-order language programming for vectors and matrices. That’s because of two key features of C++, neither of which really have much to do with objects as such:

- Overloading of both functions and operators
- Functions that can return array or object values

Using properly coded C++ routines and making use of overloading, the same equation can be implemented almost exactly as it’s written:

```cpp
x = y - 2.0 * z + (y * z) * Cross(y, z);
```

What a difference! At last, we have a language that can be extended so we can use it as a high-order language for any objects we can think up, for which mathematical operations make sense. That’s precisely why I feel my prediction made earlier is a safe one. But that brings me to a dilemma, and a plea:

**I NEED HELP!**

Creating and maintaining a library of subroutines in one language is difficult enough, which
is why the vendors of such libraries as Matlab charge so much for it. But combine that with the desire to support multiple languages, and the realization that the best ways to solve problems can diverge drastically, as we’ve seen above, depending upon the capabilities of the language, and you can see that it’s a monumental task to support all the libraries in parallel. Some help would be nice. It’s my profound hope that this effort will develop enough of a critical mass so that other people will be interested in getting involved. I’d love to see a user’s group of the sort that grew up around Software Tools, with support to help maintain and disseminate the toolset. At Embedded Systems Programming, we’ve been discussing ways to make the toolset more readily available. Options run from simply offering reprints of my articles and columns, to providing libraries online in a variety of languages. We haven’t arrived at any conclusions but, needless to say, all such services require manpower, and any help would be appreciated.

For that matter, I can’t and don’t claim to be an expert on every single aspect of mathematics useful for embedded systems programming. There are plenty of readers out there who know tricks that I’ve never dreamed of. If you’re in that category, now’s your chance to be immortal, by supplying neat algorithms of your own. I promise that all due credit will be given to any contributors.

WHERE DO WE GO FROM HERE?

In future installments of Programmer’s Toolbox, I plan to press on with more vector and matrix operations. There are two important matrix operations that are yet to be given: the matrix inverse and diagonalization of symmetric and non-symmetric matrices. Companion articles are either completed or planned, to show how to use these routines for such things as least-squares fits and linear algebra. Lately we’ve been getting into calculus, and the next series of articles will discuss techniques for numerical integration, digital filtering, digital control systems, and much more. All of these have associate tools to be given. Other areas planned for the future include:

- Dynamic simulations, including real-time
- Function minimization and optimization
- Solution for zeros of functions
- Polynomial algebra, including root-cracking
- Further explorations into C++ language extensions

Again, any suggestions for further topics are most welcome. Stay tuned.
LISTING 1 - PASCAL UNIT STRUCTURE

Unit Constants;

Interface
   { Define constants here   }
   { Define any computed constants as typed constants  }
   { Give them dummy values to be filled in  }

Const Two = 2.0;
Const Radians_Per_Degree: real = 0.0;

Implementation
   { This section empty   }
Begin
   { Initialize constants here   }

   Radians_Per_Degree := Pi/180.0;
End.

LISTING 2 - CONVERSION ROUTINES

   { Convert angle from radians to degrees  }
Function Degrees(A: Real): Real;
Begin
   Degrees := Degrees_Per_Radian * A;
End;

   { Convert angle from degrees to radians  }
Function Radians(A: Real): Real;
Begin
   Radians := Radians_Per_Degree * A;
End;