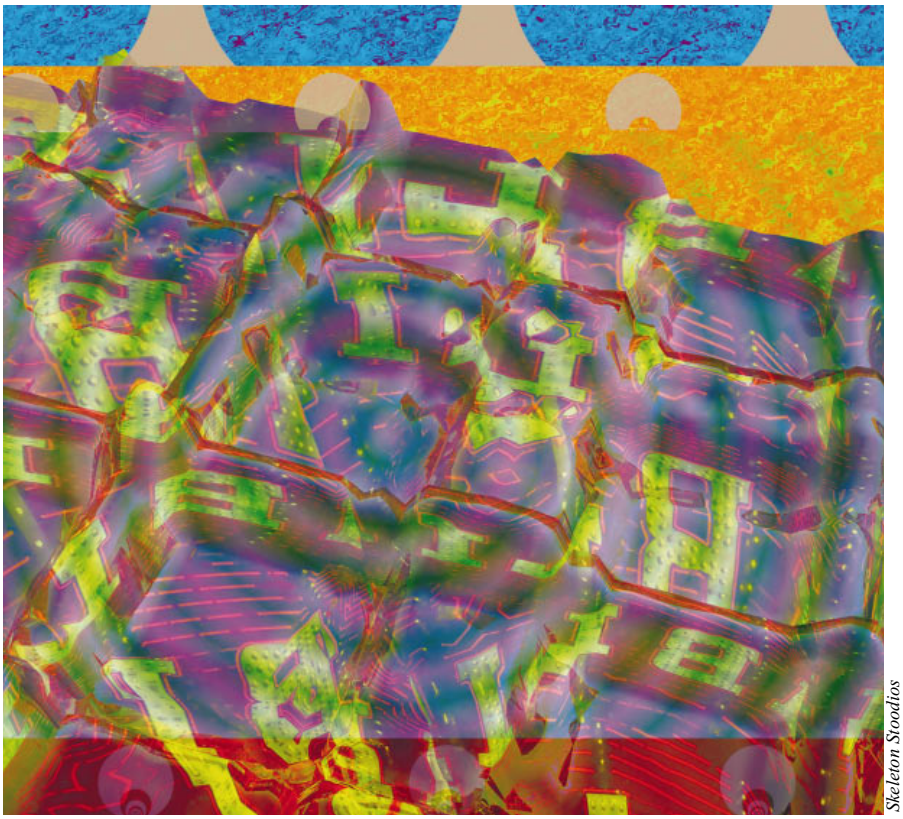
by JEAN J. LABROSSE

How to Develop Programming Conventions

The subject of coding conventions can be touchy. Anyone trying to impose a set of conventions is likely to offend somebody. But because everyone working on a project needs to row in the same direction, this article offers a couple of oars.

Today's competitive world forces us to introduce products at an increasingly faster rate due to one simple fact of business life: having a product out first may mean acquiring a major share of the market. One way to help make this possible is to assure that the mechanics of writing code become second nature. All project members should have a clear understanding on where each file resides on the company file server, what each file should be named, what style to use, and how to name variables and functions.

The topic of coding conventions is controversial because we all have our own ways of doing things. One way is not necessarily better than another. However, it is important that all team members adopt a single set of rules and that these rules are followed religiously and consistently by all participants. The worst thing that you can do is to let each programmer do his or her own thing. Such an undisciplined activity will certainly lead to chaos. When you consider that close to half of the development effort of a software based system come after its release, why not make the sometimes unpleasant task of supporting code less painful?



Skeleton Stoodios

One of the questions that you will have to answer is: who sets the rules? The answer is simple if you use the Golden Rule: (s)he who has the gold, makes the rules. In other words, it should be the software engineering manager or the project leader. Because I'm in charge of my company's software department, I wrote a set of conventions that everyone in software development follows, including any contract engineers and consultants. In this article, I will share some of these conventions with you with the hope that you will find some of them useful in your own organization. I urge you to document your own conventions as we did because it makes life easier for everyone, especially when it comes to supporting someone else's code.

DIRECTORY STRUCTURES

One of the first rules I established is that nobody can store company related files on local hard drives. Instead, all data must reside on the company file server and must follow a pre-defined structure. The file server gets backed up every day, thereby minimizing the chance of losing valuable data. We use MS-DOS and Windows and company related files are stored on a Novell-based file server. The main drive on the server is called N:. The general directory structure is shown in Figure 1.

Each product (ProdName) has its own directory under PRODUCTS\. If a product requires more than one microprocessor, then each has its own direc-

The worst thing you can do is let each programmer do his or her own thing—such undisciplined activity will lead to chaos.

tory under ProdName\. Each product that contains a microprocessor has a SOFTWARE\ directory. The SOURCE\ directory contains all the source files that are specific to the product. Our code is highly modular and we strive to reuse as much code as possible from product to product, so the SOURCE\ directory generally only contains about 20% of the software that makes the product unique. The remaining 80% of the code is found in the N:\SOFTWARE directory. The DOC\ directory contains documentation files specific to the software aspects of the product (specifications, state diagrams, flow diagrams, software descriptions, and so on). The TEST\ directory contains product build files (batch files, make files, and so on) for creating a test version of the product. A test version will build the product using the source files located in the SOURCE\ directory of the product, any reusable code (such as building blocks), and any test-specific source code you may want to include to verify the proper operation of the application. The latter files generally reside in the TEST\ directory because they don't belong in the final product. The version control (VC) directory contains version control software generated archive files of your source code, documentation, and executables. The PROD\ directory contains build files for retrieving any released versions of your product. The other directories under ProdName\ are provided to show you where other disciplines within your organization can store their product related files.

The N:\SOFTWARE\ directory is where we store all reusable, non-product specific files. We call these building blocks and each contains its own documentation and version control directories.

A product build consists of creating a product directory tree on a workstation's local drive and copying all the required source files to build the product. The source files are then compiled, assembled, and linked. Using a MAKE utility avoids copying and compiling files that haven't changed. Compiling directly on the workstation's hard disk is much faster than through the network. Because all the necessary source and object files are conveniently located in a single location, it becomes easy to run an emulator with source level debug capabilities.

SOURCE FILES

It's well known that coding is the only aspect of programming that must be done to have a product. All the documentation in the world is useless if it doesn't reflect what the code does. C has been called a write-only language because once you have written the code, it's difficult to read and understand what it does. But I believe that you can make any language write-only because it depends more on your attitude than the language you use. I always say that if you hate writing (or typing), then you're in the wrong business. Many programmers write for themselves and are not concerned with the life of a product. Another sign that you're in the wrong business is if you believe that your job is done once the code works. A major portion of a product's life consists of maintenance. In fact, in most cases, maintenance accounts for the majority of a product's development cost. You should write source code to facilitate maintenance.

One of the easiest ways I found to accomplish this goal is to adopt a clean and consistent coding style. Which style you decide to use doesn't matter, as long as you and others in your organization follow a common guide. To

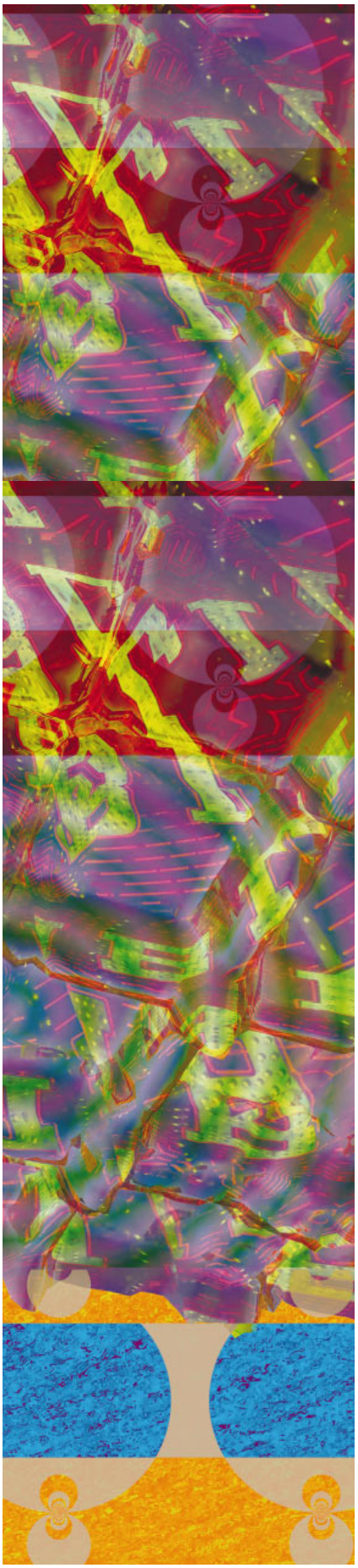
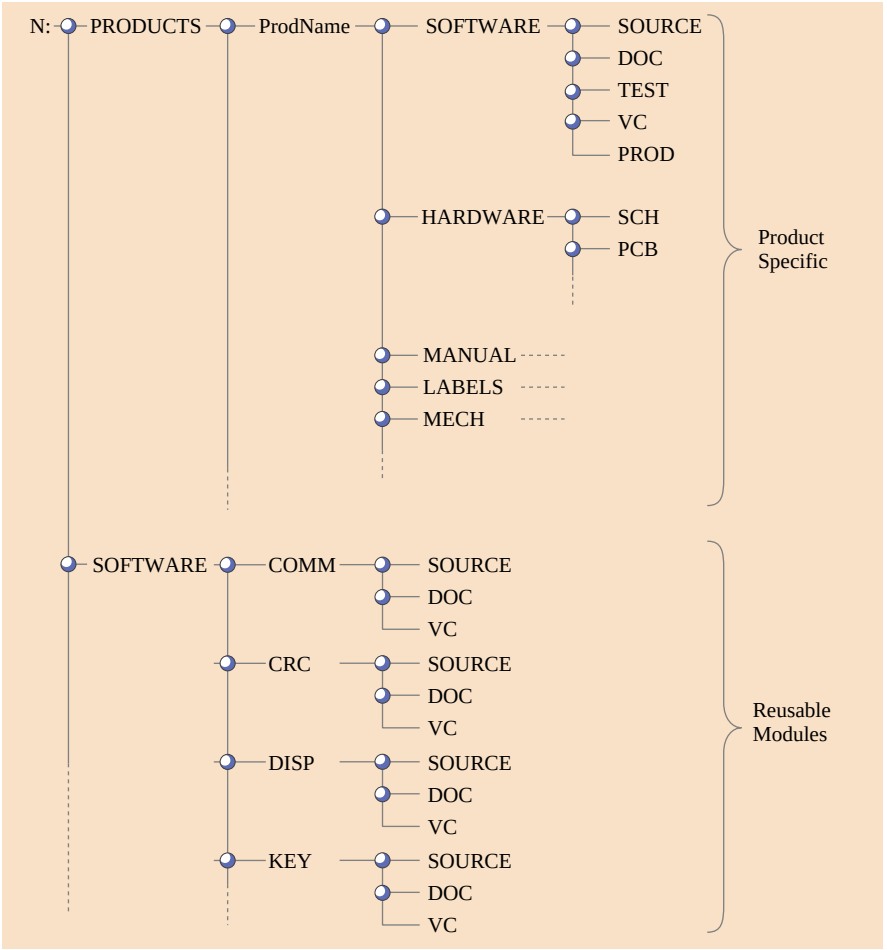


FIGURE 1
Sample directories.



that end, someone should document the style used in your organization and have everyone follow it religiously. A couple of years ago, I read an article in the *Hewlett-Packard Journal* which stated that the product development manager decided to have the team adopt a common coding style—a potentially dangerous management decision.[1] The team members were initially reluctant about having to conform to the style guide, but at the completion of the project everyone was impressed by the productivity gains. Team members were able to help each other because they didn't have to adjust themselves to another member's coding style to find bugs.

Our source files are grouped by modules, and each module has its own directory. Taking C as an example, a directory may contain as few as two files (a .C and a .H file) or as many files as needed to perform the functions of the module. In some cases, look-up tables are placed in the .C file along with the module's code, while in other cases, large tables are located in their own files. All files within a module share a common file name prefix. For example, a display module may consist of three files: DISP.C contains code and variables, DISP.H contains function prototypes and DISP_TC.C contains tables (the _TC means C source Tables). We have multiple types of display modules (LCD, LED, and so on) and each is located in its own directory. For example, the LCD module is found in N:\SOFTWARE\LCD while the LED module is found in N:\SOFTWARE\LED. The name of the files is the same in both cases. In fact, we try to provide the same functionality (function names and variable names) whether we have an LCD display or an LED display. The application code only knows that it has a display and the product requirements dictates which type.

We don't limit the width of a line of source code to 80 characters. Instead, the width is actually limited to how many characters can be printed on an 8.5" by 11" page using compressed mode (17 characters per inch). Using compressed mode, you can accommodate up to 132 characters and have enough room on the left of the page for holes for insertion in a three ring binder. Allowing 132 characters per line prevents having to interleave source code with comments. We also like to control when page breaks will occur, by inserting the special comment /*$PAGE*/ whenever a page break is desired. The file is printed using a utility that I wrote called HPLISTC. When HPLISTC encounters this comment, the utility sends a form feed character to the printer.

A source file should contain the following sections, which should always be in the same order. Header files should never contain code except for macros.

- File Heading
- Revision History
- `#defines` and macros
- `#includes`
- Variables
- Function prototypes
- Functions

The file heading is a comment block that contains our company name, address, copyright notice, file name, author's name, and a description of the module. The revision history is also a comment block that describes what changes were made to the module over time. This section is automatically filled in by our version control software when the module is released.

We place `#defines` and macros in

three different places. First, if `#defines` and macros are only applicable to the module they are placed in the module's .C file (for example, equating the different states of a state machine). There is no point in extending the scope of something that is used locally. If used by multiple .C files in a given module, `#defines` and macros can be placed in the module's .H file to make them globally visible to the module. Finally, if the `#defines` are meant to be product specific, they are placed in a file called APP.H in the product's directory. For example, the size of a module's buffer can be specified in APP.H instead of the module itself. You can also `#define` a constant to enable/disable compilation of a feature in a module to reduce your code/data size if you don't need all the functionality of a module. In other cases, conditional compilation is used to enable a version of an algorithm. For example, our CRC (Cyclic Redundancy Check) module contains two versions. One version is slow but requires very little ROM, while the other is very fast but requires the use of a 512-byte table and consumes more ROM. Setting the `#define CRC_FAST_EN` to one selects the faster version. The application itself doesn't know the difference. The `#defines` and macros are always written using upper case characters with an underscore character separating words. This style agrees with the conventions established by Kernighan and Ritchie.[3] The `#includes` section always contains a single statement as follows:

```
#include "INCLUDES.H"
```

We decided to use a single "master" include file called `INCLUDES.H` whose contents are defined in the product's directory. We like to use a single master header file because it prevents us from having to remember which header file goes with which source file, especially when new modules are added. The only inconvenience that we found is that it takes a little longer to compile each file.

VARIABLES

Most of today's C compilers conform to the ANSI X3J11 standard that allows up to 32 characters for identifiers. Descriptive variables can be formulated using this 32-character feature and the use of acronyms, abbreviations, and mnemonics (see Acronyms, Abbreviations and Mnemonics). Variable names should reflect what the variable is used for. We like to use a hierarchical method when creating a variable. For instance, the array `KeyBufIn[]` indicates that the variable is part of the keyboard module (`Key`), it is a buffer (`Buf`)—specifically, the input buffer (`In`). Upper case characters are used to separate words in a variable but this rule only applies to global variables.

In C, you can have two types of global variables: global to a file (also called local globals) and global to the rest of the world (such as, the product). Unlike my friend Jack Ganssle, I don't think that globals are necessarily bad and thus need to be avoided.[2] If used properly, globals can actually be cleaner than locals. All global variables (variables seen by other modules) are placed in the .H file of the module and not in the .C file. In C, however, the .H file generally contains an extern statement, so the question is, "How do you `extern` a variable and allocate storage for it at the same time?" The answer is that you use conditional compilation through the C preprocessor as shown in the following statements that are placed at the beginning of the .H file:

```
#ifdef xxx_GLOBALS
#define xxx_EXT
#else
#define xxx_EXT extern
#endif
```

where `xxx` is the name of the module. Each variable that needs to be declared global will be prefixed with `xxx_EXT` in the .H file. The module's .C file will contain the following declarations:

```
#define  xxx_GLOBALS
#includes "INCLUDES.H"
```

When the compiler processes the .C file, it forces `xxx_EXT` (found in the corresponding .H file) to "nothing" (because `xxx_GLOBALS` is defined) and thus each global variable will be allocated storage space. When the compiler processes the other .C files, `xxx_GLOBALS` will not be defined for that module and thus, `xxx_EXT` will be set to `extern` allowing you to reference the global variables. To illustrate this concept, let's suppose you need to create the following global variables:

```
UBYTE CtrlState;
FP32  CtrlLevel;
UWORD CtrlCtr;
```

CTRL.H would look like this:

```
#ifdef  CTRL_GLOBALS
#define CTRL_EXT
#else
#define CTRL_EXT extern
#endif

CTRL_EXT UBYTE CtrlState;
CTRL_EXT FP32  CtrlLevel;
CTRL_EXT UWORD CtrlCtr;
```

CTRL.C would look like this:

```
#define    CTRL_GLOBALS
#includes "INCLUDES.H"
```

The nice thing about this technique is that you don't have to declare global variables in the .C file and duplicate the statements with the addition of the extern attribute in the .H file. You not only save a lot of time but you also reduce the chances of introducing an error in the process. Once you use this technique, you'll never want to declare globals any other way.

Variable names should be declared on separate lines rather than combining them on a single line. Separate lines make it easy to provide a descriptive comment for each variable. You should also explicitly declare the data type of every variable instead of rely-

ing on the default, `int`.

By convention, all variables are prefixed with the module's name. This convention makes it quite easy to know where variables are declared when you're dealing with large applications. Furthermore, a file scope global should have the underscore character after the module name. For example, a local global variable in the file CTRL.C would be prefixed by "Ctrl_." Because CTRL.C will probably manipulate Ctrl variables, you'll know whether the variables are declared at the top of CTRL.C ("Ctrl_" prefix) or in CTRL.H ("Ctrl" prefix).

Formal arguments to a function and local variables within a function are declared in lower case. The lower case rule makes it obvious that such variables are local to a function because by convention, global variables will contain a mixture of upper and lower case characters. To make local variables or function arguments readable, you can use the underscore character. Within functions, certain variable names can be reserved to always have the same meaning. Some examples are given below, but others can be used as long as consistency is maintained.

i, j and k	for loop counters.
p1, p2 ... pn	for pointers.
c, c1 ... cn	for characters.
s, s1 ... sn	for strings.
ix, iy and iz	for intermediate integer variables
fx, fy and fz	for intermediate floating-point variables

Structures are `typedef` because this distinction allows a single name to represent the structure. The structure type is declared using all upper case characters with underscore characters used to separate words.

```
typedef struct {  /* LINE structure */
  UWORD StartX; /* X, Y start coordinate */
  UWORD StartY;
  UWORD EndX;  /* X, Y end coordinate */
  UWORD EndY;
  UWORD Color;  /* Line color          */
} LINE;
```

We find it useful to include the name of the structure in the suffix of a pointer to let the reader know what structure the referenced element belongs to, for example:

```
pline->Color;
```

To summarize, global variables should use the file/module name (or a portion of it) as a prefix and should make use of upper/lower case characters. File scope globals should have an underscore character following the module's prefix. Function arguments and local variables should use only lower case characters. `#define` constants and macros are always written in upper case with underscore characters separating words for sake of legibility.

ACRONYMS, ABBREVIATIONS & MNEMONICS

When creating names for variables and functions, it is often useful to use acronyms (such as OS, ISR, TCB), abbreviations (such as buf and doc) and mnemonics (such as clr and cmp). Their use allows an identifier to be descriptive while requiring fewer characters. Unfortunately, if the terms are not used consistently, they may add confusion. To ensure consistency, we have opted to create a list of these terms that we use in all our projects. Once it is assigned, the same acronym, abbreviation, or mnemonic is used throughout. As we need more terms, we simply add them to the list. Once everyone has agreed that `buf` means buffer, all project members should use that instead of having some individuals use `buffer` and others use `bfr`. To further this concept, you should always use buf even if your identifier can accommodate the full name. In other words, stick to `buf` even if you can fully write the word buffer.

In some instances one list for all products may not make sense. For instance, if your company is an engineering firm working on projects for different clients and the products that you develop are totally unrelated, then a different list for each project would be more appropriate; the vocabulary for the farming industry is not the same as the vocabulary for the defense industry. We use the rule that if all products are similar, they use the same dictionary.

DATA TYPES

While we are on the subject of variables, you may have noticed that we don't use the standard C types in variable declarations. In fact, unless you have to use the C standard library, you should avoid using C's data types because they are not portable. An `int` can either be 8, 16, 32, or even 64 bits. Similarly, a float is either a 32-bit, a 64-bit, or 80-bit value depending on the target processor and compiler. To resolve the portability issue, we create a header file which defines the following data types:

```
typedef unsigned char UBYTE;
typedef signed   char BYTE;
typedef unsigned int  UWORD;
typedef signed   int  WORD;
typedef unsigned long ULONG;
typedef signed   long LONG;
typedef float    FP32;
typedef double   FP64;
```

By definition, a `BYTE` is always 8-bits, a `WORD` is 16-bits, a `LONG` is 32-bits, and so on. Your application code as well as the reusable modules can now assume the appropriate range for each variable in a portable fashion. If you then decide to port your code to a different target you will only need to look up the definition of various data type sizes in your compiler literature and change the above type definitions.

FUNCTIONS

Function naming follows the same convention as global variables. Every function is prefixed with the module name; we use acronyms, abbreviations and mnemonics; the first

letter of each word is upper case; and local functions (file scope) have an underscore after the module name. We found that indenting four spaces works out well, but you should use whatever you are confortable with. Whatever you do, use spaces instead of tabs to indent your code. Tabs are interpreted differently on different editors and printers. Avoiding tab characters does not mean that you can't use the tab key on your keyboard. A good editor will give you the option to replace tabs with spaces (in this case, four spaces).

Functions that are only used within the file should be declared static to hide them from other functions in different files. Each local variable name should be declared on its own line, an action that allows the programmer to comment each one as needed. Actual code statements should start after adding two blank lines after local variable declarations. This space makes the delineation between variables and executable statements clear.

Our style guide also specifies how every C construct should be written. A space follows the keywords **if**, **for**, **while**, and **do**. The keyword else has the privilege of having one before and one after it if curly braces are used. We write **if** (condition) on its own line and the statement(s) to execute on the next following line(s) as follows:

```
if (y > 2) {
    z = 10;
    x = 100;
    p++;
} else {
    z = 5;
}
```

We always try to fully enclose statements within the **if** (condition) with curly braces even though the condition executes a single statement. The placement of curly braces follows the Kernighan and Ritchie style, but obviously you should adopt whatever style your organization is comfortable with.[3]

Treat **switch** statements as you would any other conditional statement. Note that the case statements are lined up with the case label. The important point here is that switch statements must be easy to follow. Cases should also be separated from one another by a blank line.

```
switch (key) {
  case KEY_BS:
       if (cnt > 0) {
       p--;
       cnt--;
     }
     break;

  case KEY_CR:
      *p = NUL;
      break;

  case KEY_LINE_FEED:
```

```
        p++;
        break;

  default:
     *p++ = key;
     cnt++;
     break;
}
```

By convention, we use **for** loops when we can know ahead of time the number of iterations the loop will perform. On the other hand, we use **while** and **do-while** loops when the number of iterations can only be known at runtime:

```
  for (i = 0; i < MAX_ITER; i++) {
  *p2++ = *p1++;
  xx[i] = 0;
}

  while (*p1) {
  *p2++ = *p1++;
  cnt++;
}

  do {
  cnt--;
  *p2++ = *p1++;
} while (cnt > 0);
```

All statements and expressions should be made to fit on a single source line. Even though this format is correct in C, when the variable names get more complicated, the code might be hard to read, making errors difficult to find. You should thus avoid the following constructs.

```
x = y = z = 1;
```

The following operators are written with no space around them:

```
->    Structure pointer operator   p->m
.     Structure member operator    s.m
[]    Array subscripting            a[i]
```

Parentheses after function names have no space(s) before them. A space should be introduced after each comma to separate each actual argument in a function. Expressions within parentheses are written with no space after the opening parenthesis and no space before the closing parenthesis. Commas and semicolons should have one space after them:

```
strncat(t, s, n);
for (i = 0; i < n; i++)
```

The unary operators are written with no space between them and their operands as shown below.

```
!p  ~b  ++i  --j  (long)m  *p  &x
sizeof(k)
```

The binary operators are preceded and followed by one or more spaces, as is the ternary operator:

```
c1 = c2   x + y   i += 2   n > 0 ? n : -n;
```

The keywords **if**, **while**, **for**, **switch**, and **return** are followed by one space.

For assignments, numbers are lined up in columns as if you were to add them, allowing you to quickly spot errors. The equal signs are also lined up:

```
x        = 100.567;
temp     =  12.700;
var5     =   0.768;
variable =  12;
storage  =  &array[0];
```

COMMENTS

Comments should be meaningful and help you and others understand how the code works. Don't just state what a reasonable programmer would conclude by simply looking at the code. Each function should be preceded with a comment block to describe what the function does, what arguments are passed, what the function returns and any other notes about the function. This comment block is also a great place to describe the meaning of any "magic" numbers. For example, if you decide to fold the constant *PI* divided by four into 0.785398, then you should say so in the comment.

We avoid interleaving code and comments because we find it difficult to mentally separate them from each other. Comments should go to the right of the actual C code. When large comments are necessary, they are written in the function description header or in a comment block before the actual code. Comments are lined up as shown in Figure 2 (see page 70).

The comment terminator (`*/`) does not need to be lined up, but for neatness we prefer to do so. You don't need one comment per line because a comment could conceivably apply to several lines.

EMPHASIZE QUALITY

Recently, I visited a small company that wanted to sell us an expensive piece of manufacturing equipment. The unit was impressive, the showroom was impeccable, and the production floor was acceptable, but the software engineering department seemed alarmingly chaotic. I almost recommended that we didn't buy the equipment for fear of having software problems. We ended up purchasing the equipment anyway, based on the word of satisfied customers, unit features, and performance.

The software department at my company is one of the first places that is visited whenever we give a tour of our facilities to customers. Upper management emphasizes the quality of our software activities to potential clients, which has sometimes led to additional business because of the peace of mind we were able to provide the client companies. Obviously, you shouldn't write software so that you can give plant tours, but if your company is concerned about the quality of its products, it should also focus attention on what goes inside.

You may not agree with some of the conventions we've adopted but what's important is that you recognize that you will increase productivity by having your organization work from a common set of rules. Programmers will certainly resist this kind of change but the long term benefits will be worth the struggle. I have concluded over the years that I'd much rather fight to have things done correctly the first time than spend double the effort when a programmer moves on to greener pastures and leaves the rest of us holding the proverbial bag.

Jean Labrosse is software engineering manager at Dynalco Controls in Fort Lauderdale, FL and he has been designing embedded systems for many years. Labrosse is the author of two books: µC/OS, The Real-Time Kernel *and* Embedded Systems Building Blocks *(R&D Books, Lawrence, KS).*

FIGURE 2
Function to update the time of day.

```
/* ************************************************************************************
                                        UPDATE THE TIME-OF-DAY
  * Description : This function is called to update the time (i.e. hours, minutes and seconds) or a
  *               software managed clock.
  * Arguments   : None.
  * Returns     : TRUE     if we have completed one day.
  *               FALSE    otherwise
  * Notes       : This function updates the global variables: ClkSec, ClkMin and ClkHr.
  ************************************************************************************************ */
  static BOOLEAN ClkUpdateTime(void)
  {
      BOOLEAN newday;

      newday = FALSE;                          /* Assume that we haven´t completed one whole day yet
  */
      if (ClkSec >= 59) {                      /* See if we have completed one minute yet
  */
          ClkSec = 0;                          /* Yes, clear seconds
  */
          if (ClkMin >= 59) {                  /*    See if we have completed one hour yet
  */
              ClkMin = 0;                      /*    Yes, clear minutes
  */
              if (ClkHr >= 23) {               /*       See if we have completed one day yet
  */
                  ClkHr = 0;                   /*       Yes, clear hours ...
  */
                  newday    = TRUE;            /*       ... change flag to indicate we have a new day
  */
              } else {
                  ClkHr++;                     /*       No,  increment hours
  */
              }
          } else {
              ClkMin++;                        /*    No,  increment minutes
  */
          }
      } else {
          ClkSec++;                            /* No,  increment seconds
  */
      }
      return (newday);
  }
```

Jean can be reached via e-mail at 72644.3724@compuserve.com.

REFERENCES

1. Long, David W. and Duff, Christopher P., *A Survey of Processes Used in the Development of Firmware for a Multiprocessor Embedded System,* Hewlett-Packard Journal, October 1993.

2. Ganssle, Jack G., "A Modest Software Standard," *Embedded Systems Programming,* March 1996, pp. 105-107.

3. Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1988.

OTHER SOURCES

Labrosse, Jean J., *Embedded Systems Building Blocks*, R&D Books, Lawrence, KS, 1995.

Labrosse, Jean J., *µC/OS, The Real-Time Kernel*, R&D Books, Lawrence, KS, 1992.

Maguire, Steve, *Writing Solid Code*, Microsoft Press, Redmond, WA 1993.

McConnell, Steve, *Code Complete*, Microsoft Press, Redmond, WA, 1993.