BY SHELLEY GRETLEIN · NATIONAL INSTRUMENTS

Designing embedded-system applications with high-level tools

HIGH-LEVEL PROGRAMMING LANGUAGES HELP YOU MORE QUICKLY DEPLOY YOUR APPLICATION, BUT CAREFUL ATTENTION TO DETAILS CAN IMPROVE THE PERFORMANCE OF YOUR CODE AND LEAD TO AN EFFICIENT APPLICATION THAT YOU CAN STILL DEVELOP IN A TIMELY MANNER.

To meet time-to-market and productivity pressures, embedded-system developers increasingly consider and use high-level design-software tools that provide more abstraction, simpler representations of programming constructs, and automatic code generation. Embedded systems are generally restrictive, requiring a focus on execution speed, execution reliability, determinism, power consumption, and memory usage. To realize the promise of higher-level design software such as UML (Unified Modeling Language), National Instruments LabView, and frameworks such as Eclipse, designers must continue to pay attention to the challenges associated with embedded-system development. These higher-level design tools do not program themselves, and designers must continue to make trade-offs among factors including performance, memory usage, and power consumption. As a designer, you should evaluate the following development areas when using high-level design tools: memory usage and management, programming structures within embedded-system applications, processor-intensive calculations, and hybrid development—combining low- and high-level languages.

MEMORY MANAGEMENT

Applications that run solely on Windows rarely have to consider memory usage or management because Windows can access large amounts of virtual memory. However, an embedded-system application running on an RTOS (real-time operating system) does not use virtual memory because doing so hinders determinism. Also, when you restart an application in Windows, you "wipe clean" the application's memory, but in the RTOS, embedded-system applications start when you reset the real-time device and stop when you turn it off. Therefore, you should design your deterministic applications to be memory-conscious. For example, always preallocate space for your arrays equal to the largest array size that you will encounter (see **sidebar** "Data types and variables").

Proper memory allocation is a big part of efficiently programming embedded-system applications. In general, dynamic-memory allocation is an expensive operation that you should avoid. With dynamic-memory allocation, you can create data types and structures of any size and length to suit your program's needs. Dynamic-memory allocation differs from automatic or static-memory allocation and creates an object that remains allocated until the programmer or a garbage collector explicitly deallocates it.
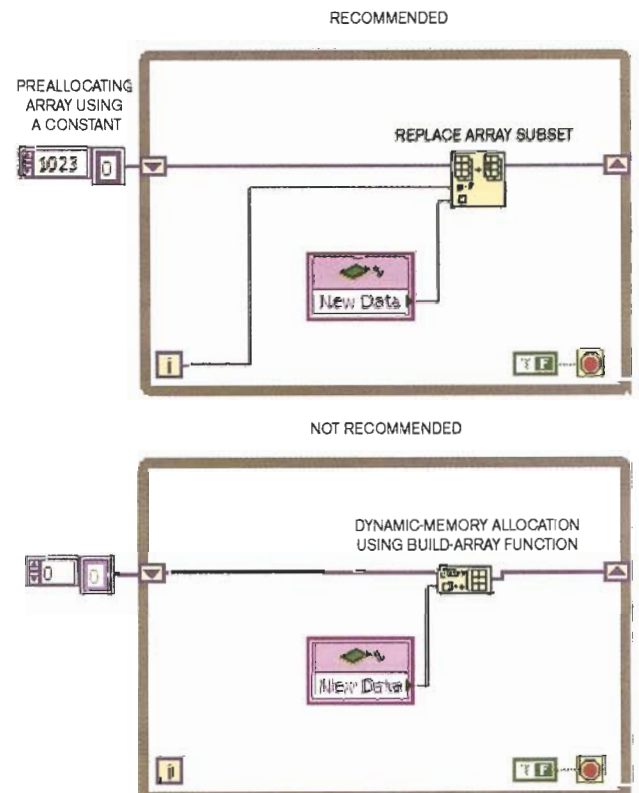

Figure 1 Dynamic-memory allocation is expensive in time-critical code. It is especially detrimental if dynamic allocation occurs inside a loop to store data in arrays. A common way to avoid dynamically allocating memory in a loop is to preallocate the memory for any arrays before the loop starts execution.

In general, dynamic-memory management is an important aspect of modern software-engineering techniques. However, real-time-system developers avoid using it because they fear that the worst-case execution time of the dynamic-memory-allocation routines is not bounded or has an excessively large bound. The following section highlights two examples—in C and in a higher-level graphical language, such as LabView.

In C, programmers most commonly use the function "malloc" to attempt to "grab" a continuous portion of memory and define it by: void *malloc(size_t number_of_bytes). Therefore, the function returns a pointer of type void * that is the start in memory of the reserved portion of sizeof() number_of_bytes. If the system cannot allocate memory, it returns a NULL pointer. Because it returns a void *, the C standard states that you can convert this pointer to any type. The size_t argument type is defined in stdlib.h and is an unsigned type.

Therefore, char *cp; cp=malloc(100) attempts to get 100 bytes and assigns the start address to cp. Also, it is common to use the sizeof() function to specify the number of bytes: int *ip; ip=(int *) malloc(100*sizeof(int)).

Some C compilers may require you to cast the type of conversion. The (int *) defines coercion to an integer pointer. Coercion to the correct pointer type is crucial to ensure that you correctly perform pointer arithmetic. It is beneficial to use sizeof() even if you know the size you want; it makes for device-independent—that is, portable—code.

In a graphical programming language such as LabView, dynamic-memory allocation can occur when using the build-array and concatenate-string functions. Alternatively, you can replace the build-array primitive with a replace-array-subset function to replace elements in a preallocated array. You should create the preallocated array outside the loop by using an array

Figure 2 UML case diagrams show the static structure of an object, the internal structure, and relationships.

constant or the initialize-array function. See the LabView code in **Figure 1**, which compares the implementations.

EMBEDDED-PROGRAMMING STRUCTURES

Engineers commonly use programming structures, such as for loops, while loops, and others, and case structures in desktop applications, but you should also review and optimize these structures for embedded-system applications. Each structure has its own use. At a simple level, if you know how many times you need to loop, then use a for loop. If you want to loop until the system meets a certain condition, then use a while loop. However, note that, whatever you can do with a for loop, you can do with a while loop; the number of loops can also be a condition. And, although you can use a while loop for anything a for loop can do, it's often recommended to use for loops instead of while loops because for loops have memory optimizations such that, if you know the number of iterations, you could preallocate the array. For embedded-system applications, there are a few more cases to avoid or consider when using programming structures.

Depending on the compiler, a constant inside a loop can cause each loop iteration to make a copy of that data, resulting in increased execution time and memory usage. You can avoid this situation either by moving the constant outside the loop or by using local variables to pass data into the loop. Therefore, avoid placing large constants inside loops. When you place a large constant inside a loop, the system allocates memory and initializes the array at the beginning of each loop iteration. This operation can be expensive in time-critical code. A better way to access the data is to place the array outside the loop and wire it through a loop tunnel, or you can use a global variable.

For simple decision-making, it is often faster to use the select function instead of a case structure. Because each case in a case structure can contain its own block diagram, there is significantly more overhead associated with this structure than with a select function. However, it is sometimes more optimal to use a case structure if one case executes a large amount of code and the other cases execute little code (**Figure 2**). You should decide whether to use a select function versus a case structure on a case-by-case basis.

Structs in C and clusters in LabView are useful for bundling heterogeneous data into manageable packages. However, information about the contents must also propagate with that data. Especially when passing data to subfunctions, individual

TABLE 1 SAMPLE SCALING, MULTIPLIER, AND BIT-SHIFT VALUES

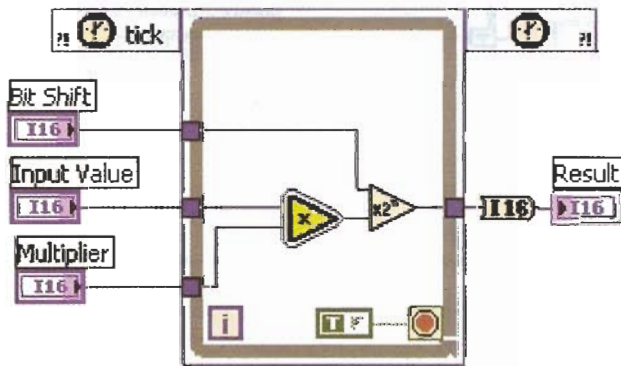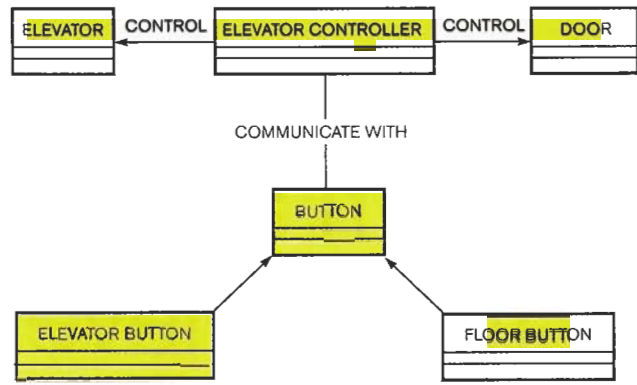| Desired scaling value	Multiplier	Bit shift	Actual scaling value	Absolute error
1.5	3	−1	1.5	0
0.1428	73	−9	0.1425	0.0003
10	3	−5	0.09375	0.00625
10	102	−10	0.09961	0.00039

Figure 3 Designers can use the Embedded LabView FPGA virtual instrument to scale by a noninteger value.
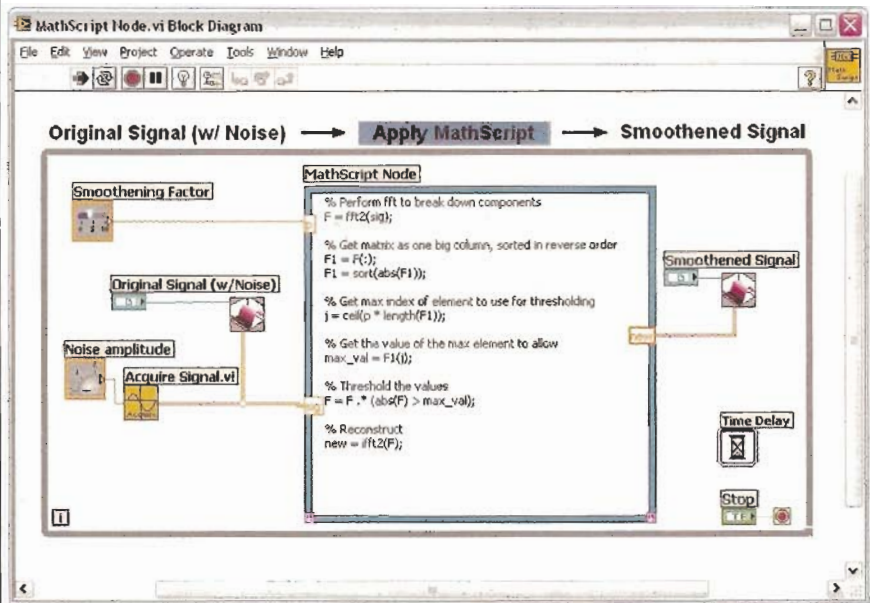
Figure 4 Graphical programming allows you to combine textual m-file-based mathematics plus C, assembly, or VHDL code into your designs.

elements rather than structs or clusters usually increase the speed of your applications. Use them to avoid passing unnecessary data-type information to the subfunction. The performance of passing a cluster or struct depends a lot on how you pass the information. For example, in C, you could pass the structure "in place"—that is, by pointer—and get good performance, possibly better than passing the individual elements, depending on the use case.

PROCESSOR-INTENSIVE MATH

You also have to consider the types of calculations you are performing to optimize your application. Some calculations, for example, are "processor-intensive" tasks. In general, a processor-intensive task is any task that is limited by how fast the processor can compute the data. Video encoding is an example of a processor-intensive application; I/O-bound tasks are more memory-intensive.

A more common way to optimize an application is through its calculations. A binary shift is one such optimization technique. In C programming, the operators for binary shifting are << and >>. Shifting to the left causes numbers to multiply by the power of two that you shifted them. Shifting to the right is identical, except it divides by the power of two.

For example, suppose you want to plot a pixel in VGA mode 13h by copying the color to the screen offset $x+y\times320$. Because 320 is the same as $64+256$, you can use screen$[((y<<8)+(y<<6))+x]=$ color instead of screen$[y\times320+x]=$ color. This approach is much faster than multiplication, a complex operation, because binary shifts are simple operations. Handling scaling in the LabView FPGA tool is similar: Just use a multiply function and then a scale-by-power-of-two function. First, multiply the input value by a known integer, generating a larger intermediate result. Shifting the intermediate result to the left (scale-by-power-of-two function with a negative n value) is effectively a division by a power of two. Combining the multiplication and division gives the effective scaling or multiplication of the input value by a noninteger value. Figure 3 shows an example of this scaling implementation. You can further optimize the code by replacing the bit-shift control with a constant.

As part of these calculations, it is important to make sure that the intermediate result of the multiplication fits into the data type you are using. With the saturation-multiply function, you can multiply two 16-bit integers, generate a 32-bit value, and know that the result fits into the 32-bit integer. If the final result needs to be a 16-bit integer, then the scale-by-power-of-two function must shift the intermediate product back into the 16-bit range with the coerce function (Figure 3). The integer multiplier

nd bit-shift value determine the noninteger-scaling value by which you are multiplying the input value. For example, to scale by 1.5, set the multiplier to 3 and the bit shift to −1. This step leads to 3/2, which equals 1.5. To scale by 1/7 (~0.1428), set the multiplier to 73 and the shift to −9. This step leads to 73/512, which equals approximately 0.1425.

Combining a multiplication and a scale-by-power-of-two function does not provide an exact result for every noninteger-scaling value, but it does offer a good approximation within the limited resolution of integers. The key is to find the right combination of multiplier and bit-shift value. For example, to divide by 10, you can use a multiplier of 3 and bit shift of −5 (divide by 32). Doing so

results in an effective scaling constant of 3/32, which equals 0.09375, an error of 0.00625 from your intended value. However, you could also use a multiplier of 102 and a shift of −10 (divide by 1024), which gives you an effective scaling constant of 102/1024, approximately equal to 0.09961, an error of 0.00039 from your intended value. In general, you achieve better results when you use larger multipliers and bit-shift values.

As you increase the multiplier, make sure that you do not exceed the range of the intermediate-result data type; otherwise, you will saturate this value and receive an incorrect result. To find the right multiplier and bit-shift values, it is often easiest to pick a suitably large bit-shift value that you base on the output-

DATA TYPES AND VARIABLES

Some environments automatically handle data-type conflicts by converting the smaller data type into the larger one. For example, if a type conflict exists between an integer and a floating-point number, your programming language may convert the integer into a floating-point number and then perform the operation. This conversion is expensive and, in many cases, unnecessary. In most cases, you can avoid casting and coercion by using the correct data type for each variable. However, if the data must be cast or coerced, it can be more efficient to convert the data before sending it to the operation or function.
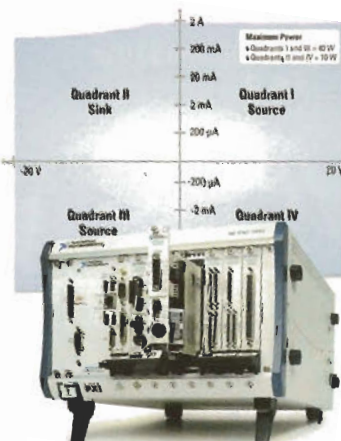
For this discussion, it is best to establish some definitions to make sure your programmers are speaking the same language. Wikipedia defines a global variable as "a variable that does not belong to any subroutine or class and can be accessed from anywhere in a program." Likewise, a local variable is "a variable that is given local scope. Such variables are accessible only from the function or block in which [they are] declared."

In general, you can modify global variables from anywhere

within the application; however, using global variables is a poor programming technique. A global variable has unlimited potential for creating mutual dependencies, and adding mutual dependencies increases complexity. Another challenge of using global variables is their associated complexity with code reuse because of the interdependencies.

Local variables, on the other hand, can have a scope that is declared, written to, and read, usually with no side effects—except if your variable data is bound to the user interface. For example, with National Instruments' LabView, every time you access a local variable, it executes extra code to synchronize the variable with the user interface or front panel. You can improve code performance, in many cases, by using a global variable instead of a local variable. The global variable has no extra front-panel synchronization code. Therefore, globals of 8 bytes or less are faster than local variables. If a global is larger than 8 bytes, you cannot access it atomically, and it requires a mutex, making it slower and making it a shared resource.

YOU OFTEN OBTAIN THE BEST RESULTS BY USING A HYBRID OF HIGH- AND LOW-LEVEL CODE.

value range and data types and then calculate the corresponding multiplier for your desired scaling value. Also, by using a constant value for the bit shift, you use fewer resources on the FPGA when compiled. **Table 1** shows sample scaling values and the corresponding multiplier and bit-shift values.

For processors with no floating-point units, converting to floating point to perform an operation and then converting back to an integer data type can be expensive. For example, using a quotient-and-remainder function is faster than using a normal divide function, and using a logical-shift function is faster than using a scale-by-power-of-two function.

Integer-math routines available on fixed-point platforms, such as an FPGA, make data processing a little more challenging. You can use integer-math functions to scale or multiply data by whole integer values. You can use bit shifting to multiply or divide a value by any power of two. When you combine these two operations, you create a simple method to scale or multiply a value by a noninteger scaling constant.

A common application of scaling an input value by a constant is in simulators of sensors, such as LVDTs (linear-variable-differential transformer) and synchro/resolvers. Each of these sensors has an excitation-voltage input that a sine-wave signal feeds. The sensor modulates the amplitude of the excitation signal based on the position of the sensor, and the system passes the resulting output signal to the measurement system. When you want to simulate such a sensor, you need to measure the excitation signal and generate an analog-output value that corresponds to the excitation voltage scaled by a value corresponding to the position of the simulated sensor. This operation requires you to quickly multiply the analog-input measurement by a noninteger value and generate the

result on an analog-output channel.

You often obtain the best results by using a hybrid of high- and low-level code. With many high-level, open programming languages, you can use C or assembly code directly within your application. It is best to use low-level algorithms within your code if you want to reuse algorithms or if there is a small numeric or array algorithm you can code more optimally. LabView uses a graphical-system-design approach that encompasses many models of computation. You can combine textual m-file-based mathematics with graphical programming; insert C, assembly, or VHDL code into your designs; and access models such as state and simulation diagrams (**Figure 4**). Therefore, you can choose the right approach for each unique challenge you are trying to solve. You must also emphasize the importance of performance profiling. Make sure that you spend time optimizing the 20% of the code that takes 80% of the time. This 20/80 pattern is extremely prevalent, yet programmers too often spend hours unnecessarily fine-tuning the 80% of their code that does not significantly affect performance.

By following good embedded-programming practices, you can better optimize your code to meet the constraints of your embedded-system application. Implementing one or two of these techniques may noticeably improve the performance of your application, but the best approach is to incorporate a combination of all these techniques.EDN

AUTHOR'S BIOGRAPHY

Shelley Gretlein is a real-time-and embedded-product strategist at National Instruments, where she works with real-time and embedded hardware and software products, including LabView Real-Time, LabView FPGA, LabView Embedded, and LabView control design and simulation, as well as hardware targets, including CompactRIO and PXI. She also evaluates future technologies for upcoming product integration, including RTOSs, FPGAs, and general software technologies. Gretlein holds bachelor's degrees in computer science and in management systems from the University of Missouri—Rolla. She enjoys wakeboarding, running, triathlons, and renovating.