Optimizing LabVIEW Embedded Applications

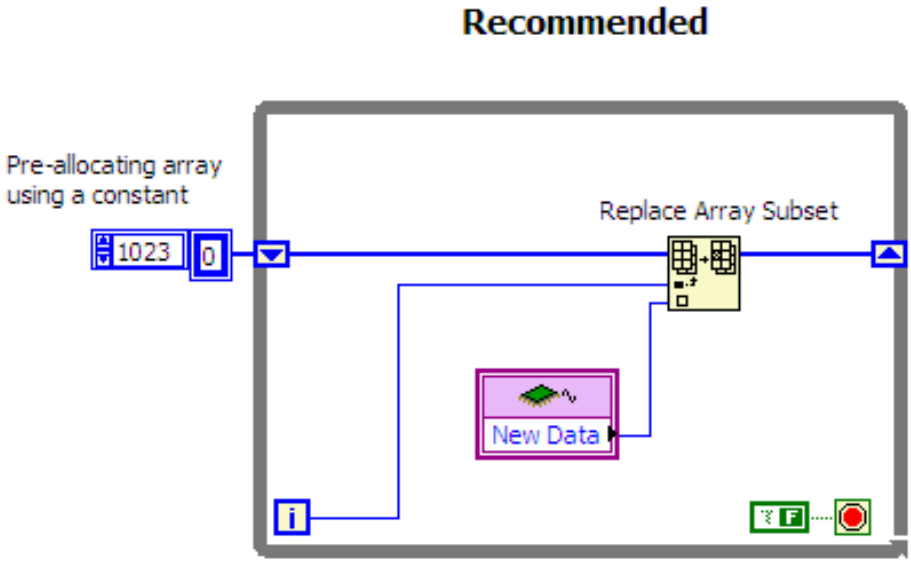
Overview

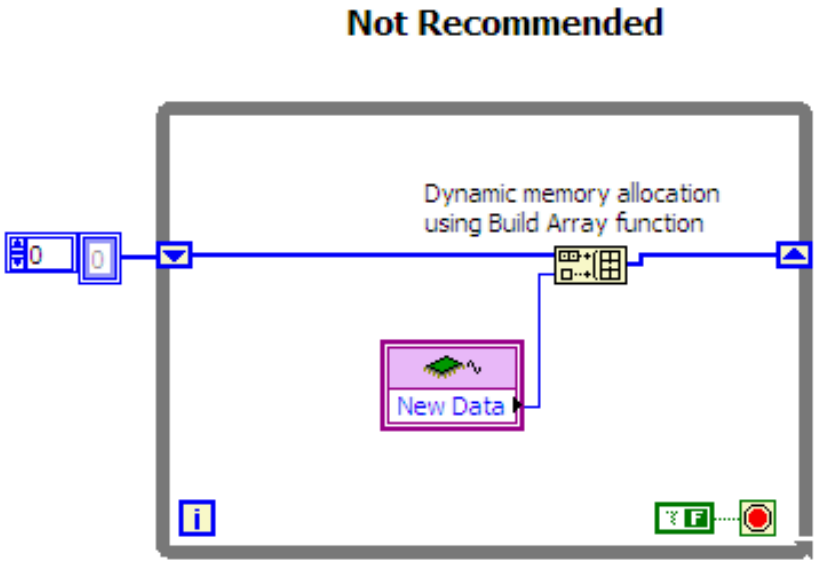
When developing an embedded application, system constraints such as memory limitations and time-critical code requirements can play a crucial role in your programming approach. You can't afford the software implementation to have any bottlenecks. This document outlines good programming practices that can be used to help optimize your embedded application with LabVIEW 7.1 Embedded Edition. The techniques described are applicable for any embedded target.

Memory Allocation

Avoid dynamic memory allocation. Dynamic memory allocation is very expensive in time-critical code. In LabVIEW Embedded, dynamic memory allocation can occur when using the Build Array and Concatenate String functions. Alternatively, the Build Array primitive can be replaced with a Replace Array Subset function in order to replace elements in a preallocated array. The preallocated array should be created outside of the loop by using an array constant or with the Initialize Array function. LabVIEW code is shown below to contrast the different implementations.

Optimizing LabVIEW Embedded Applications



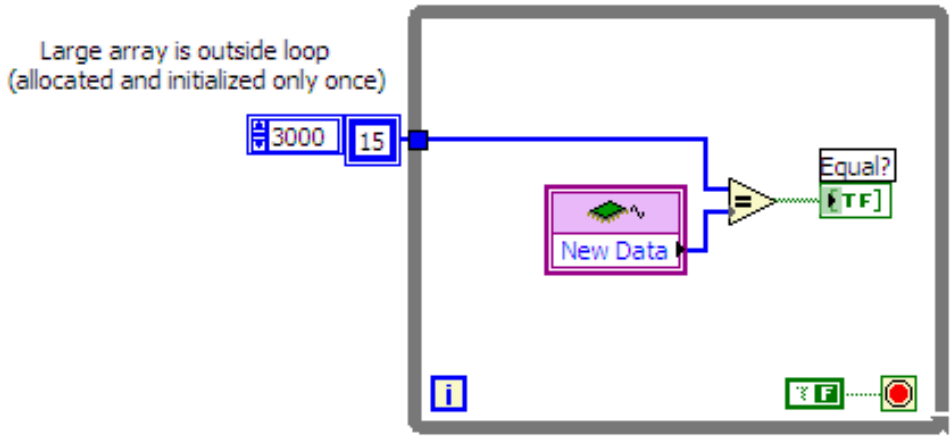


Data Placement

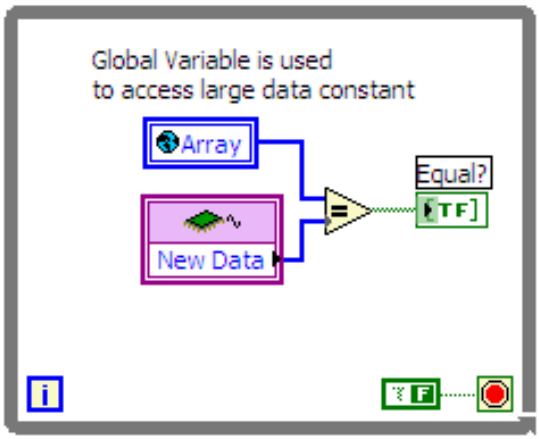
Avoid placing large constants inside loops. When a large constant is placed inside a loop, memory is allocated and the array is initialized at the beginning of each iteration of the loop. This can be an expensive operation in time-critical code. A better way to access the data place the array outside of the loop and wire it to through a loop tunnel, or to use a global variable. Examples of the two recommended methods are shown below.

Optimizing LabVIEW Embedded Applications

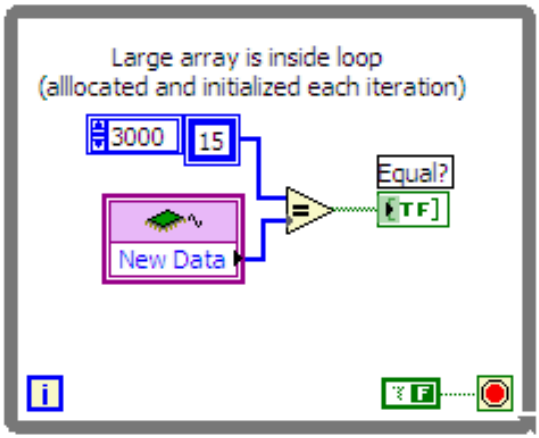
Recommended - Method 1



Recommended - Method 2



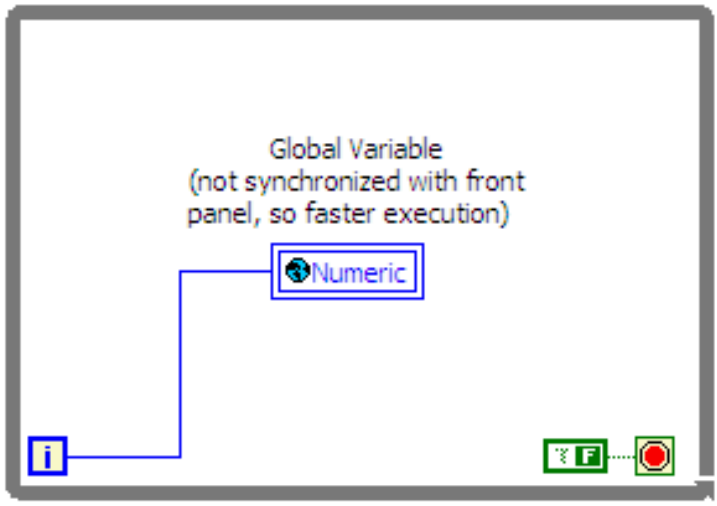
Not Recommended



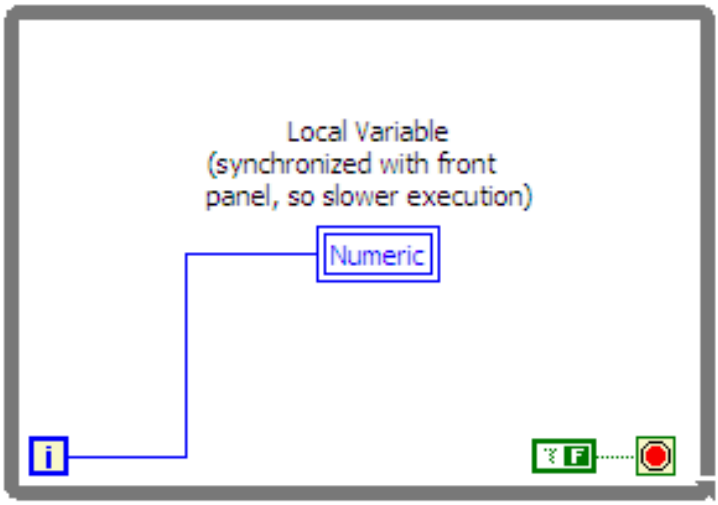
Optimizing LabVIEW Embedded Applications

Use Global Variables instead of Local Variables. Every time a local variable is accessed extra code is executed to synchronize it with the front panel. Code performance can be improved, in many cases, by using a global variable instead of a local. The global has no extra front panel synchronization code and so executes slightly faster than a local.

Recommended

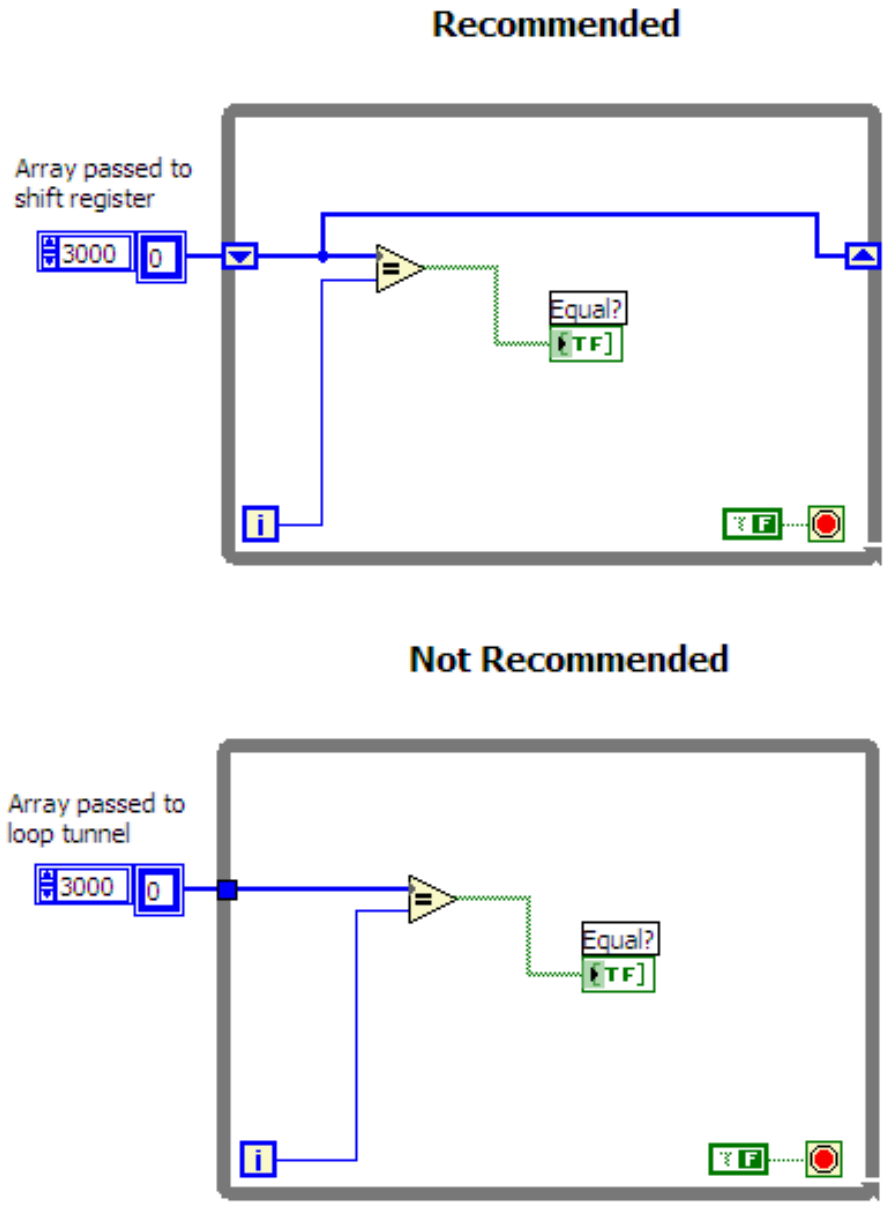


Not Recommended



Use shift registers instead of loop tunnels for large arrays. When passing a large array through a loop tunnel, the original value must be copied into the array location at the beginning of each iteration, which can be expensive. The shift register does not perform this copy operation, but make sure to wire in the left shift register to the right if you don't

want the data values to change.

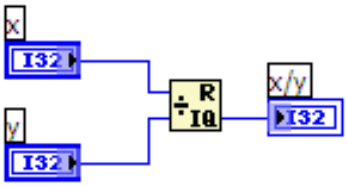


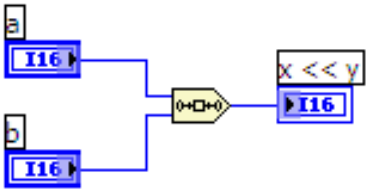
Numeric Conversion

Use integer operations instead of floating point operations. If your processor does not have a floating point unit, converting to floating point to perform an operation and then converting back to an integer data type can be very expensive. In the examples below, using a Quotient & Remainder function is faster than a normal Divide function, and using a Logical Shift function is faster than a Scale by a Power of 2 function.

Recommended

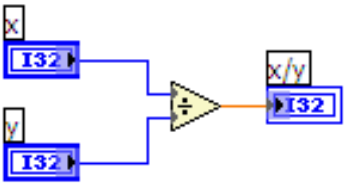
Integer Operations

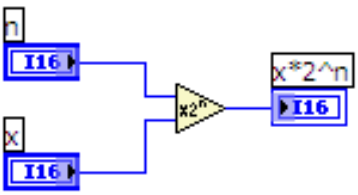




Not Recommended

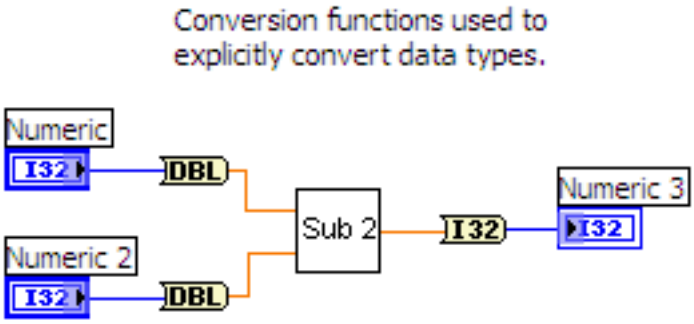
Floating Point Operations



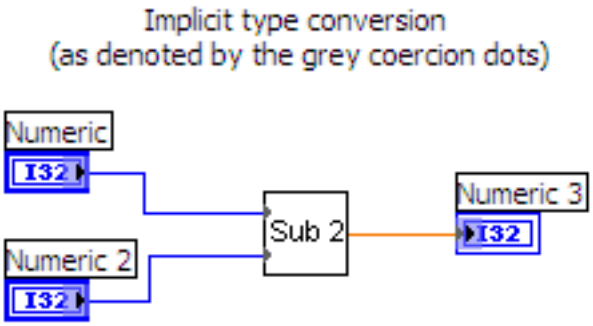


Avoid automatic numeric conversions. Another technique for improving code performance is to remove all implicit type conversions (coercion dots). Use the Conversion functions to explicitly convert data types as this avoids a copy operation and a data type determination.

Recommended

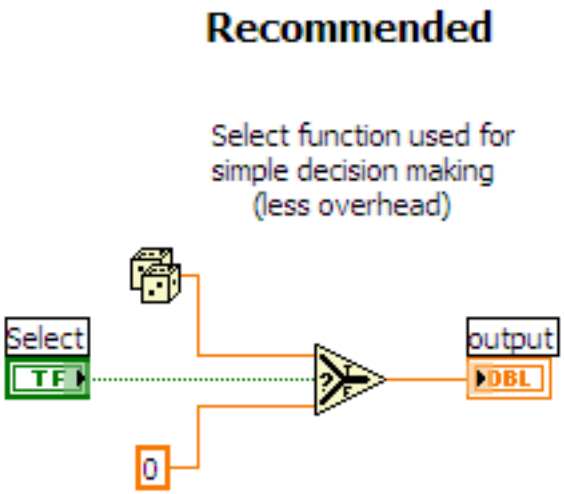


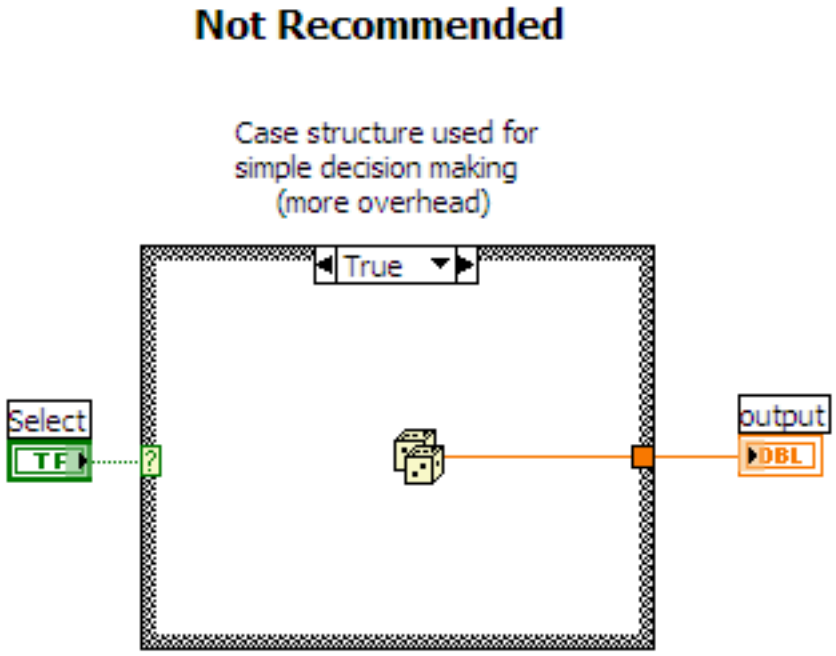
Not Recommended



Functions and Data Types to Avoid

Avoid Case Structures for simple decision making. For simple decision making in LabVIEW, it is often faster to use the Select function rather than a Case structure. Since each case in a Case structure can contain it's own block diagram there is significantly more overhead associated with this structure when compared 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 very little code. The decision to use a Select function versus a Case structure should be made on a case by case basis.

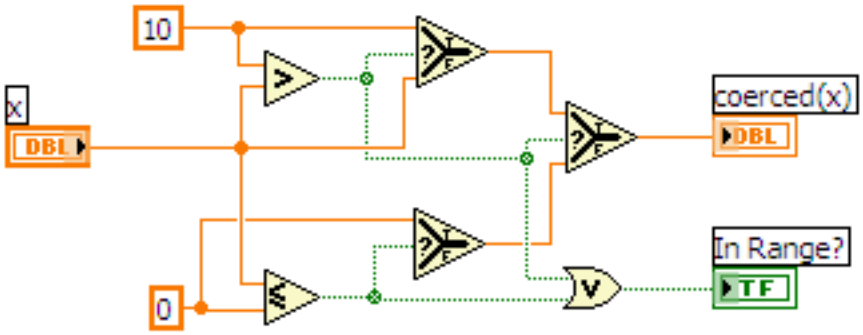




Avoid In Range and Coerce in Time-Critical Code. The In Range and Coerce function has significant overhead associated with it due to the special user configurable features and extra data type determination operations. This function should be re-implemented with comparison and Select functions if it is used in time-critical code.

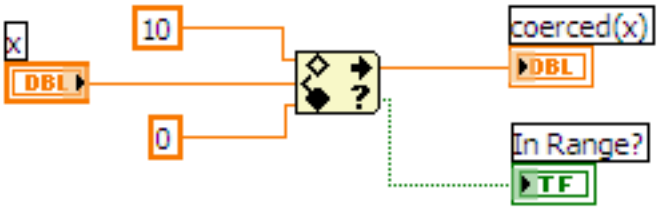
Recommended

Comparison and Select functions used
to implement In Range and Coerce function
(less overhead)



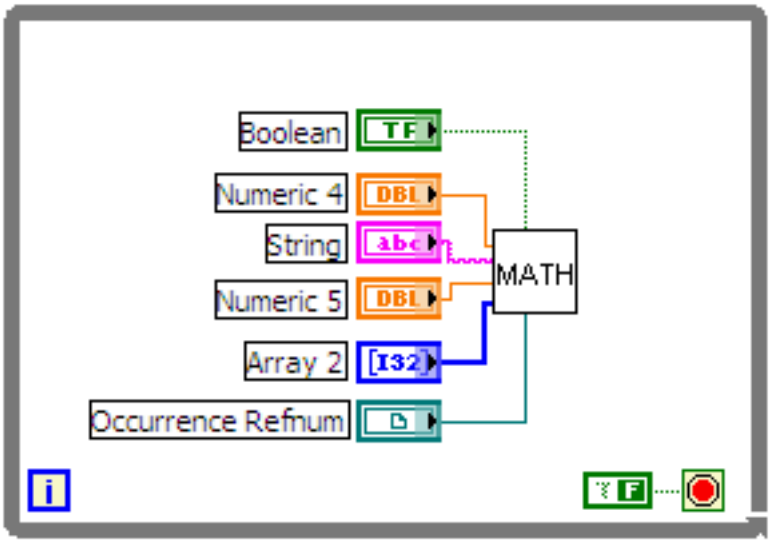
Not Recommended

In Range and Coerce function
(more overhead)

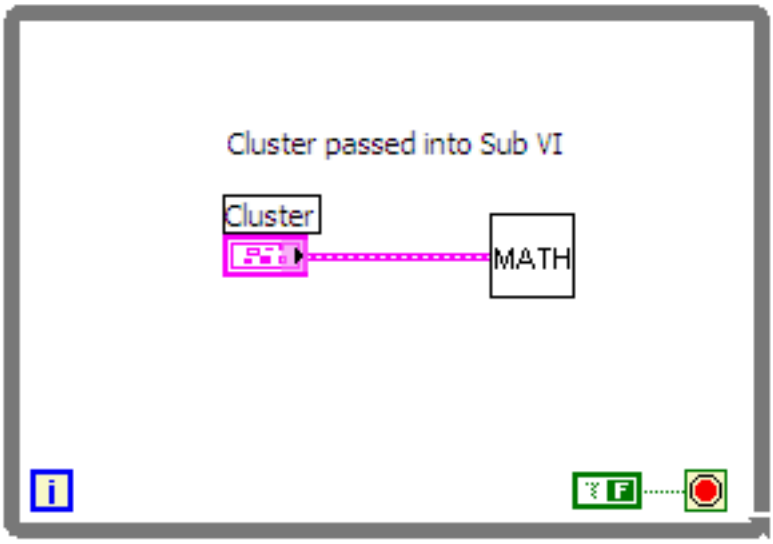


Avoid Clusters in Time-Critical Code. Clusters passed into subVIs will result in unnecessary data type information being passed as well. In order to speed up your code, try not to use clusters in time-critical areas of your block diagram.

Recommended



Not Recommended



Using the Inline C Node

Often the best results can be obtained by using a hybrid of LabVIEW and C code. The Inline C Node and Call Library Node allow the use of C code directly within your LabVIEW block diagram. See the Embedded Development Module documentation (linked below) for more information on the use of the Inline C and Call Library Nodes.

Optimizing LabVIEW Embedded Applications

The best cases for using C-based algorithms within your LabVIEW code are:

1) If you already have existing C algorithms that you'd like to reuse.

2) If there is a small numeric or array algorithm that can be coded more optimally in C.

Optimal Build Settings

The following build settings will usually give your code the fastest execution time:

Build Setting	Purpose
Generate Serial Only = TRUE	Increases performance in code with parallel operations.
Generate Debug Info = FALSE	Removes calls to debug code.
Generate Guard Code = FALSE	Removes extra protective code from math and array routines.
Generate Integer Only = TRUE	Improves performance on targets without a floating point unit.
Use Stack Variables = TRUE	Uses stack space rather than statically allocated memory locations.
Generate C Function Calls = TRUE	Generates more efficient code when calling subVIs, although all inputs to all subVIs must be wired.

Conclusion

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

Refer to the links below for more information on LabVIEW 7.1 Embedded Edition and the LabVIEW Embedded Development Module.

Related Links:

Product Manuals - LabVIEW Embedded Development Module Release Notes

Product Manuals - LabVIEW Embedded Development Module Porting Guide