Multiple Programming Approaches in NI LabVIEW

NI LabVIEW is a graphical dataflow programming environment. When using dataflow in LabVIEW, you define an execution flow in code by creating diagrams that show how data moves between functions (known as virtual instruments, or VIs). However, with LabVIEW, you can combine multiple programming approaches besides graphical data flow (G) in a single application. Use this flexibility to select your tool of choice for creating algorithms and solving an infinite variety of engineering problems.

Defining Programming Approaches

The phrase ‘programming approaches’ encompasses different languages for programming, models of computation, levels of abstraction, methods for interacting with existing code, and ways for representing algorithms. Over the years, National Instruments has added interfaces and methods for communication in LabVIEW to extend the number of approaches that are available.

You can write and import multiple approaches into the same block diagram as the familiar G dataflow language. LabVIEW compiles all of these approaches for the appropriate hardware target, which can span desktop computers, real-time OSs, field-programmable gate arrays (FPGAs), mobile devices, and embedded processors such as ARM.[1]

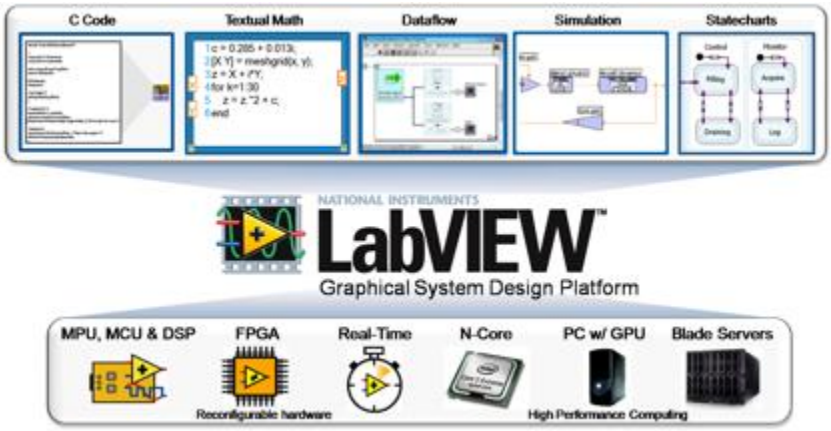

Figure 1. The LabVIEW graphical system design platform offers many options.

Learn more about supported targets and platforms.

Sending and receiving information between programming approaches is easy in LabVIEW. Data flow is the glue that links different languages and computational models together. Information and data values can easily be passed from the customized user interface (known as the front panel), network interfaces, analysis libraries, databases, and I/O to a different language or interface using G.

Programming in G

Data flow, the fundamental LabVIEW programming method, was the original, and only, programming approach when NI introduced LabVIEW 1.0 in 1986. Unlike sequential-style programming, the flow of data in a dataflow program dictates when, and in what order, operations are executed. In sequential languages such as C and C++, the order of the commands in the source code (as opposed to the availability of data) determines the order in which execution will occur.

G follows a dataflow model for running functions and primitives, or VIs. A block diagram function or node executes when all its inputs are available. When a node completes execution, it supplies data to its output terminals and passes the output data to the next node in the dataflow path.

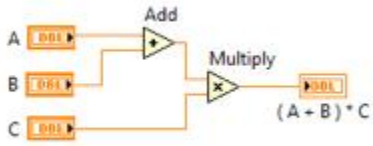

Figure 2. In this model, A and B are added, and the result is multiplied by C and displayed.

The graphical code in Figure 2 shows how a mathematical equation can be represented in G. This diagram consists of two nodes (an add node and a multiply node), and has three numerical inputs (A, B, and C). First, A and B are added. The multiplication node does not execute until both inputs are provided, so it depends on the addition node to complete and provide the result of A + B, at which point it computes the result – (A+B)*C.

Although it is possible to explicitly define variables in G, one of the most obvious differences between G code and other languages is that the functional equivalent of a traditional variable is a wire. Instead of passing variables between functions, wires

define the functions to which a value is passed. Other familiar programming concepts such as While Loops, For Loops, conditional code, callback functions, and digital logic are all part of the G dataflow programming language

Learn more about graphical programming.

Using Configuration-Based Programming

In 2003, National Instruments released NI LabVIEW 7 Express, which featured Express VIs – a new technology designed to further simplify common programming tasks and algorithm creation. Unlike traditional VIs, Express VIs abstracted tasks by offering a configuration-based approach to programming.

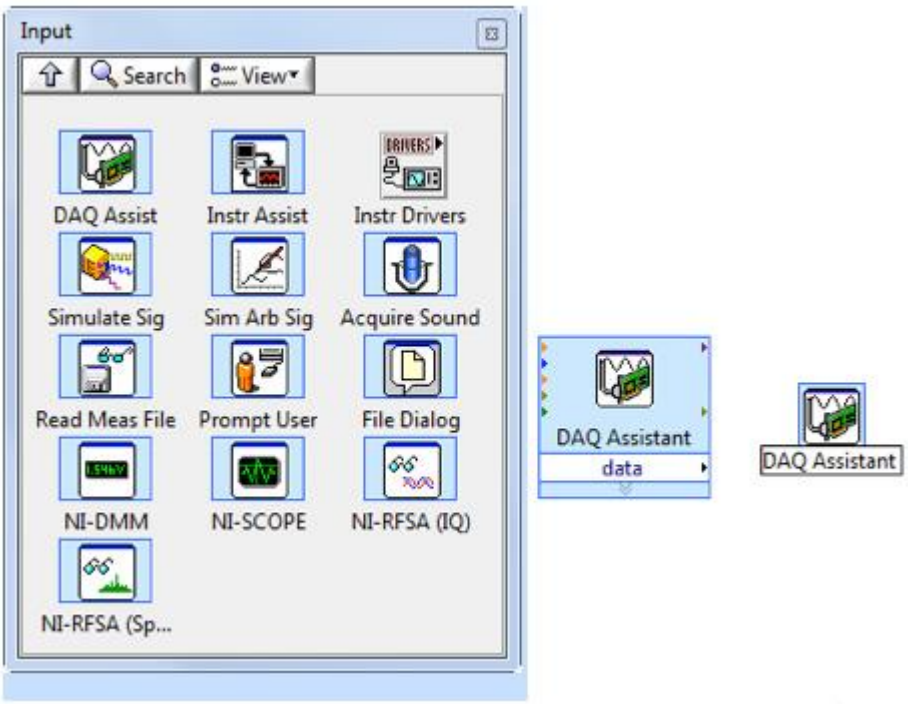

Figure 3. These are Express VIs as they appear on the palettes, an Express VI when first placed on the block diagram, and an Express VI when represented as an icon.

LabVIEW distinguishes Express VIs with large blue icons. When you place an Express VI on the block diagram, a dialog appears so you can configure how the function executes. After completing the configuration, the LabVIEW development environment writes the necessary code (represented by the Express VI) for you. You can view and modify this code, and you can change the Express VI configuration by simply double-clicking the Express VI icon.

Consider the task of reading real-world signals into software for analysis. LabVIEW is designed to make integration with hardware for I/O simple and easy thanks to native drivers and support for thousands of instruments. However, even a task that would otherwise take a handful of VIs to execute can be simplified to a single Express VI. The DAQ Assistance Express VI prompts you to select the channels you want to send and receive I/O to and from, and configure parameters such as sample rate, terminal configuration, scales, triggering, and synchronization. You also can preview the data within the interface before saving the configuration.

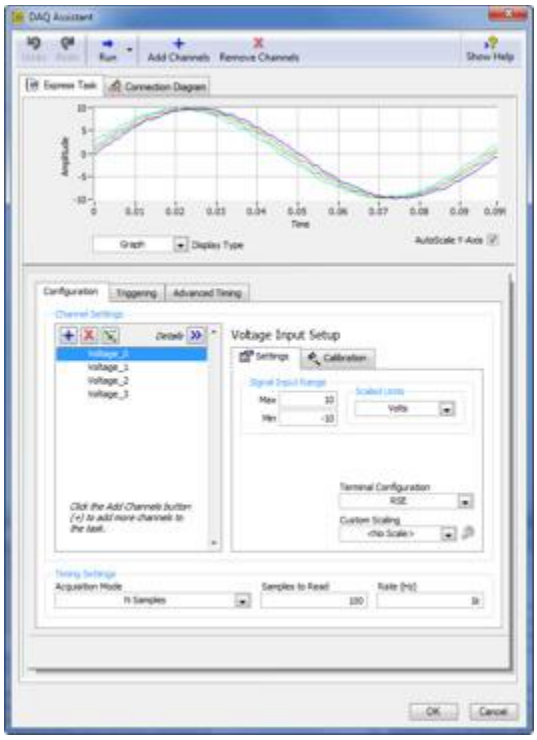

Figure 4. The DAQ Assistant Express VI makes configuring timing and channel parameters extremely simple.

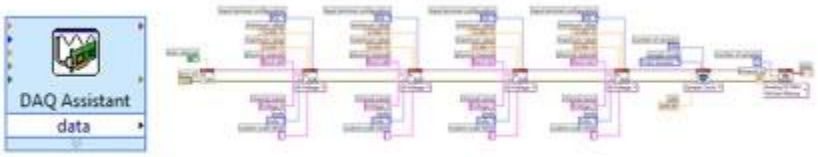

Figure 5. The DAQ Assistant Express VI is functionally equivalent to this G code.

Express VIs do not offer the same low-level control as VIs, which is why you may prefer to write the code entirely using VIs. New users interested in learning low-level

constructs can easily convert an Express VI to the underlying G code by right-clicking the Express VI and selecting Open Front Panel. Normal VIs can do everything an Express VI can do. The LabVIEW Professional Development System also includes a utility for creating custom Express VIs.

Working with .m File Format

With the LabVIEW MathScript RT Module, you can import, edit, and execute .m files, a common programming approach for numeric analysis, signal processing, and advanced mathematics. By combining them in a VI with G, you can build these scripts into stand-alone executables that run on a desktop computer, as well as deterministic real-time hardware.

There are several ways to work with MathScript in LabVIEW. For an interactive interface in which you can load, save, develop, and execute .m file scripts, you can use the MathScript window, as shown in Figure 6.

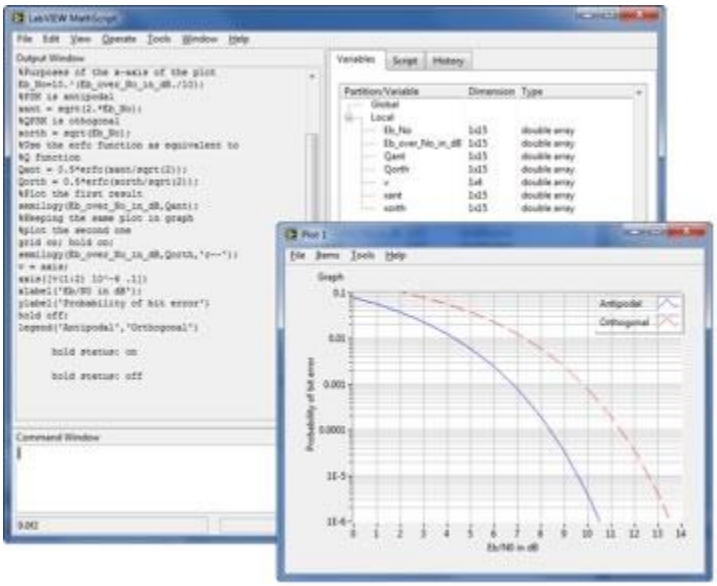

Figure 6. Interactively develop your textual algorithms with the MathScript window.

To deploy your .m file scripts as part of a LabVIEW application and combine graphical and textual programming, you can use the MathScript Node. With the MathScript Node, shown in Figure 7 as the blue structure on the block diagram, you can run .m file scripts from LabVIEW graphical programs. You can use the MathScript Node to insert textual algorithms into a VI and then use the LabVIEW graphical programming environment to instrument the scripts by adding knobs, slides, buttons, graphics, and other user

controls and indicators. The MathScript Node can utilize user-defined functions that are defined elsewhere in the application, or are loaded from disk.

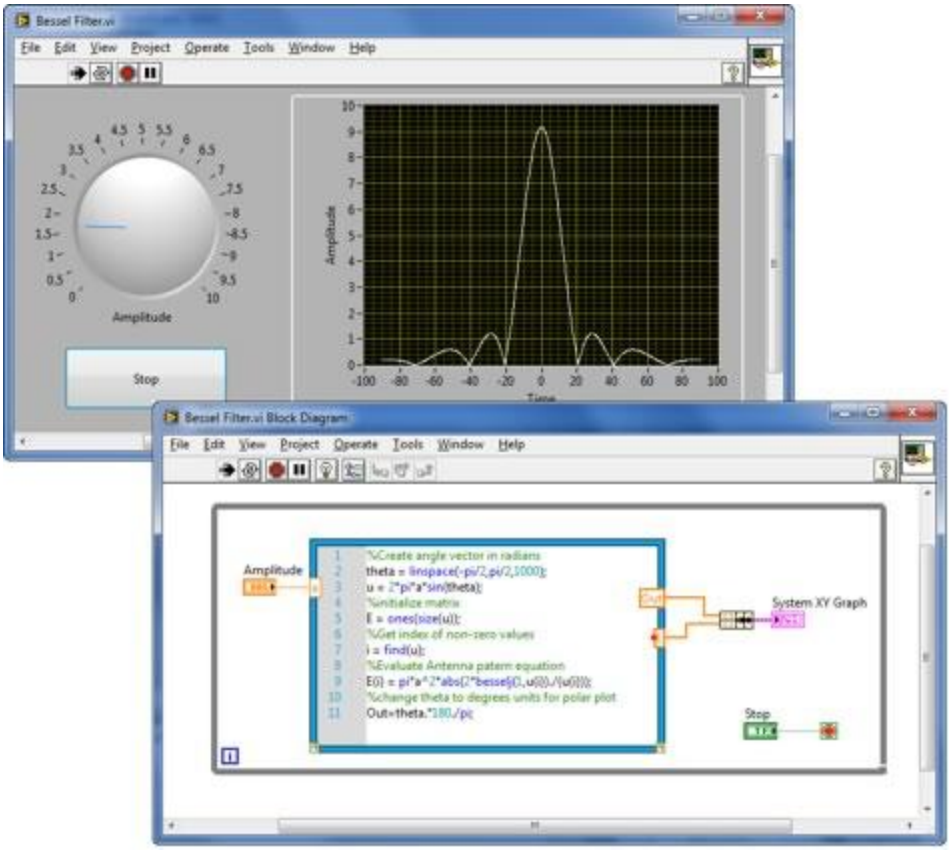

Figure 7. The MathScript Node makes it easy to interface G programming with .m scripts.

The LabVIEW MathScript RT Module adds native .m file script support to LabVIEW without requiring additional third-party software. Because you can include and run .m scripts using the MathScript Node, you can use a popular programming approach while still taking advantage of tight LabVIEW integration with I/O, an interactive user interface, and the other approaches described here.

Using Object Orientation

Object orientation is a popular programming approach across a wide variety of programming languages. It allows a variety of similar, yet different items, to be represented as a class of objects in software. The class definition includes characteristics of each object and actions that the class is capable of, often described as properties and methods. Classes can have children that inherit these properties and actions, but you can add more characteristics or override existing ones.

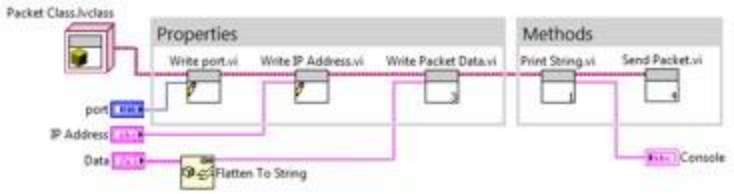

Figure 8. Object orientation uses a class, such as this one, and its associated properties and method VIs.

LabVIEW developers who feel more comfortable with an object-oriented programming approach can define a class in LabVIEW 8.2 or later. The primary advantages of using a class in LabVIEW are:

- **Encapsulation**: Encapsulation is the consolidation of data and methods into a class where you can access data only through class-member VIs. With encapsulation, you can create modular blocks of code that you can easily update or change without affecting other sections of code within the application.
- **Inheritance**: Inheritance allows you to use an existing class as the starting point for a new class. If you create a new LabVIEW class and set it to inherit data and member VIs from another class, the new class can use the public and protected member VIs of the class from which it inherits. It also can add its own data and member VIs to increase its functionality
- **Dynamic Dispatch**: You can also define methods by multiple VIs with the same name throughout the class hierarchy. These are called dynamic dispatch methods because exactly which one of the set of VIs LabVIEW calls is not known until run time.

These object-orientation characteristics enforce practices that make code more readable and scalable and limit access to information to VIs that have been explicitly given permission. This development method is another way to represent algorithms and operations using G, thereby giving you the flexibility to choose designs and methodologies that you feel more comfortable with, and that are best-suited for the application.

Modeling and Simulation Diagrams

Physical system simulation and modeling in software is a popular approach if you are designing a complex system that can be described by a differential equation. With it, you can analyze models to learn about dynamic system characteristics and create a controller that achieves desired behavior.

The Control & Simulation Loop in Figure 9 allows for the deterministic execution of a differential equation according to the ordinary differential solver – a variety of which are

available in LabVIEW. This programming approach uses a data flow that looks very similar to G, but is better described as signal flow. As shown in Figure 9, you can combine a model-based programming approach with other methods, including G and the MathScript Node.

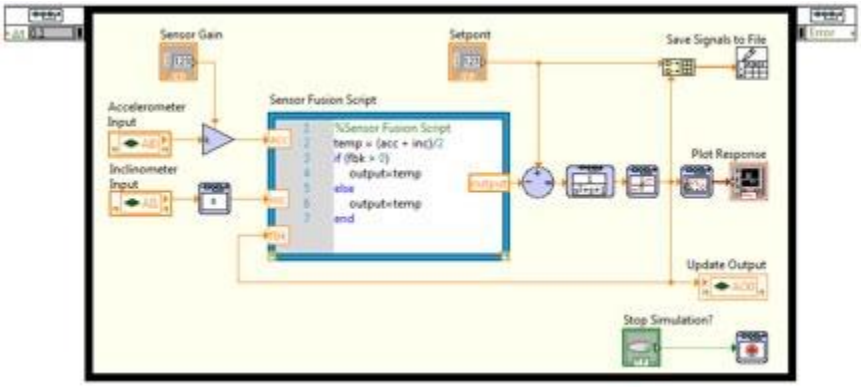

Figure 9. This simulation diagram shows signal flow, hardware I/O, and a MathScript Node.

The Control & Simulation Loop supports functions that you use to deploy discrete linear time-invariant (LTI) system models to National Instruments real-time hardware. You can use these functions to define discrete controller models in transfer function, zero-pole-gain, or state-space form. With time- and frequency-analysis tools such as time-step response or Bode plot, you can interactively analyze open- and closed-loop behavior. You can also use built-in tools to convert your models developed in The MathWorks, Inc. Simulink® software to work with LabVIEW. hese dynamic systems can be deployed to real-time hardware targets without any intermediate steps thanks to the LabVIEW Real-Time Module, which is well-suited for rapid-control prototyping and hardware-in-the-loop applications.

Documenting with Statecharts

The NI LabVIEW Statechart Module offers statecharts, which are high-level design documents that you can use to diagram system functionality. When combined with LabVIEW graphical data flow to define the behavior of each state, statechart diagrams can function as executable specifications. Statecharts expand classic state diagrams to add concurrency and hierarchy, so you can describe systems that contain parallel tasks. Also, statecharts add a formal way to respond to events, making them ideal for describing reactive systems. This is especially useful for designing embedded devices, control systems, and complex user interfaces.

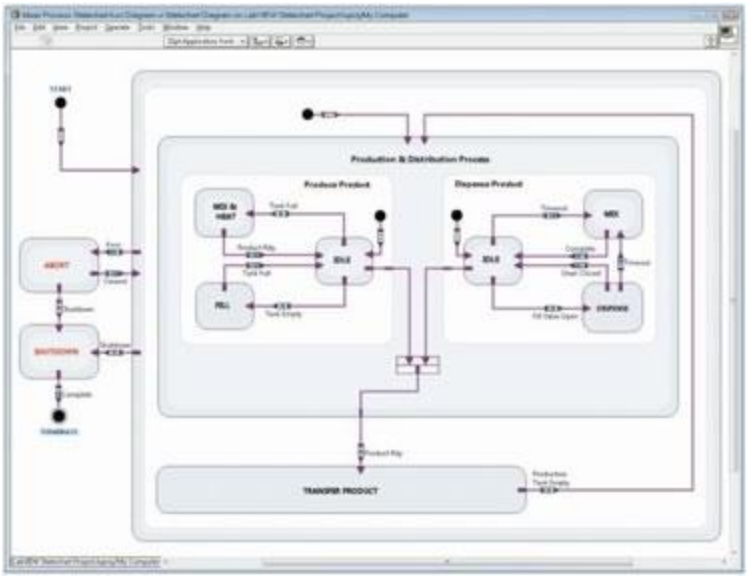

Figure 10. The LabVIEW Statechart Module uses statecharts to diagram system functionality.

You may often create high-level block diagrams to illustrate the working relationship between different systems and subsystems in an application. For example, an application may have separate processes, each responsible for tasks such as data acquisition, data output, network communication, data logging, and updating a user interface. These diagrams are typically used to determine which information to provide and share among different systems, and the logic that dictates the order in which they are executed. You can easily translate these diagrams into a LabVIEW statechart diagram, making it simple to develop real-world systems in software.

The statechart model of computation offers a sophisticated way to tackle complex application development. Statecharts are especially useful for programming event-response applications such as intricate user interfaces and advanced state machines used to implement dynamic system controllers, machine control logic, and digital communication protocols.

Learn more about the NI LabVIEW Statechart Module.

Writing VHDL for FPGA Targets

You can use the LabVIEW FPGA Module to write code that runs on an FPGA using G. However, as with the previous programming approaches, you might want to reuse existing code or have the flexibility to choose the means of implementation. Most FPGAs are programmed using VHDL, a text-based dataflow description language.

Instead of rewriting existing intellectual property (IP) from VHDL in G, you can import the VHDL into your VI using the Component-Level IP (CLIP) Node.

The CLIP Node provides a framework for importing external FPGA intellectual property (IP) into the LabVIEW FPGA Module. A CLIP XML file is typically required to map the existing IP interface to values that you can use on a VI block diagram, but LabVIEW includes a CLIP Import Wizard that creates this interface automatically. It lists IP inputs and outputs in the LabVIEW project that you can drag onto the block diagram for use on your FPGA, as shown in Figure 11.

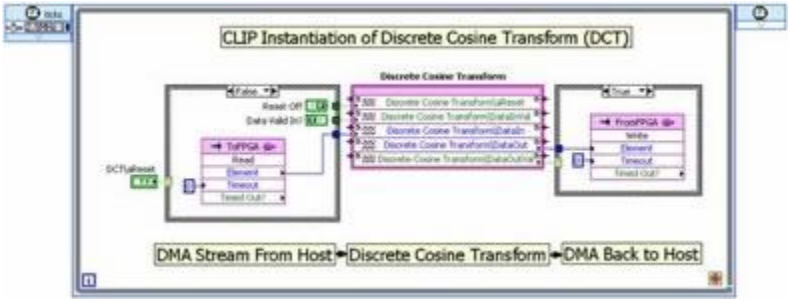

Figure 11. This LabVIEW statechart diagram shows the CLIP Node.

Because NI uses Xilinx FPGAs and the Xilinx toolchain as part of the LabVIEW FPGA Module, you can easily use the Xilinx core generator to create compatible cores. You can also use the Xilinx Embedded Development Kit to create any type of soft microprocessor. In addition, many third-party IP vendors can provide all types of signal processing, bus solutions, or application-specific cores.

Learn more about the NI LabVIEW FGPA Module and the CLIP Node.

Incorporating C-Based Syntax

You can incorporate sequentially executed text-based syntax into a VI block diagram using one of several techniques. The Formula Node offers an inline structure that supports a syntax similar to traditional C programming. Much like C, every line ends with a semicolon and variables must have a defined scope.

The Inline C Node is similar to the Formula Node with additional support and functionality for low-level programming and header files without the overhead of a function call. You can use the Inline C Node for any C code, including assembly directives and #defines, that syntactically is between the curly braces in a C file.

The Inline C Node is available only for targets that use generated C code. The Inline C Node is not supported for desktop Windows targets.

Interfacing with Built Assemblies

Instead of importing source code to a LabVIEW block diagram, you may want to call into built assemblies or reuse built LabVIEW applications in other environments. Applications written in LabVIEW can easily reuse existing code and algorithms developed in other languages or programming approaches. Additionally, you might need to build an assembly from LabVIEW code, which includes the programming approaches discussed above, to be called by a different environment.

LabVIEW offers multiple solutions for both scenarios. LabVIEW can call external code in DLLs or shared libraries and code exposed through ActiveX or .NET interfaces. In addition, you can reuse LabVIEW code in other programming languages by building a LabVIEW DLL or shared library, or by using ActiveX.

If you have existing C code and need to reuse it in LabVIEW, one technique is to build the code as a DLL and call it using the Call Library Function Node. In fact, based on your C application architecture, you can use simple LabVIEW parallel programming to run two or more existing C routines in parallel without the additional complexity of C-based multithreaded programming. To make importing external libraries simple, LabVIEW includes the Import Shared Library Wizard, which automatically creates or updates a LabVIEW wrapper VI project library for Windows .dll file, Mac OS .framework file, or Linux .so file functions.

Interfacing with the command-line is also possible with the System Exec.vi, which provides OS-specific interfaces for calling executables and other build libraries.

Taking Advantage of Flexible Programming

The combination of multiple programming approaches in a single development environment offers the advantage of reusing existing code and algorithms developed in other languages. It also makes it possible to combine simple, high-level abstractions with lower-level code that gives you more visibility and control of your application. These abstraction layers represent highly complex operations in simple, easy-to-read representations, but can be coupled with functions that give low-level control over application behavior and hardware interfaces. Thanks to tight integration with I/O, you can combine these approaches with real-world signals to take advantage of the most recent hardware technology such as multicore CPUs, FPGAs, and embedded

processors.

For any problem, there are multiple ways to solve it – and LabVIEW gives you the flexibility to choose from multiple programming approaches.

[1] Not all programming approaches are available for the listed targets, depending on resource availability and feature support