Document Type: NI News
NI Supported: Yes
Publish Date: Dec 3, 2012

Top 3 Design Decisions When Programming an Embedded System

Table of Contents

Engineers working on large, complex project with short deadlines will face many design tradeoffs and decisions. Deciding where to focus time during the design phase is a big challenge. This paper identifies three important design decisions you should make when programming an embedded system, and provides guidance given your application requirements. Taking time to consider each of these three items will put you on the right track to designing a reliable, scalable, and maintainable embedded application.

What Software Architecture Should You Use?

When working on a project with a short deadline, code is often not designed to be scalable an d maintainable because it is not the primary goal of the project. However, a scalable application will pay off significantly in the long term as new features are added and software bugs are addressed. For any application that goes beyond a prototype, a scalable and maintainable architecture is strongly recommended.

When deciding what type of software architecture to implement, you can save a significant amount of development time by starting with a prebuilt NI LabVIEW for CompactRIO Sample Project available in LabVIEW 2012. These sample projects were designed to be used as architectural starting points that can scale to a wide variety of embedded control and monitoring applications. By starting from a sample project, you can ensure you create a reliable, scalable, maintainable application.

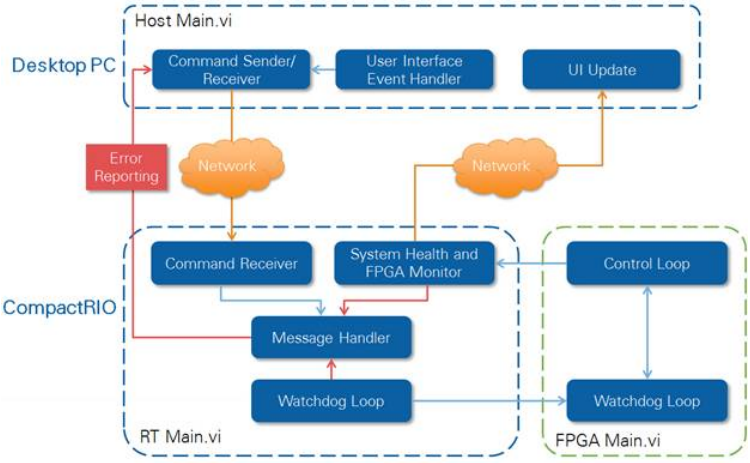

Figure 1. LabVIEW FPGA Control Sample Project for CompactRIO Software Architecture

Use the guidelines in Table 1 to select a sample project that meets your application requirements. If your application requirements differ from those Table 1, you can design your own architecture by following best practices outlined in the Introduction and Basic Architectures section of the LabVIEW for CompactRIO Developers Guide.

Does my application require control rates of at least 1 kHz and/or hardware-based reliability?	LabVIEW FPGA Control Sample Project for CompactRIO
Does my application require control rates less than a few hundred Hz?	LabVIEW Real-Time Control Sample Project for CompactRIO
Does my application require streaming analog data and logging it to disk?	LabVIEW FPGA Waveform Acquisition and Logging Sample Project for CompactRIO

Table 1. Use your specific application requirements to select an architecture included in LabVIEW 2012.

What Data Communication Models Should You Implement?

Data communication is often the most challenging piece of any embedded software architecture,but it's also one of the most foundational pieces. This means selecting the right data communication models up front is critical. If you're using an NI sample project as an application starting point, the core communication models have already been implemented. However, if you are expanding a sample project or have decided not to use one, you'll need to consider three data communication models that are common across embedded control and monitoring applications: command based, current value data storage, and stream. Often times, your application will require a combination of these models This section provides guidance on when to use one model over another.

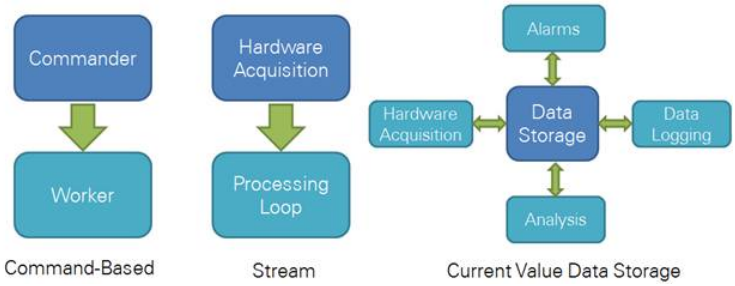

Figure 2. Common data communication models used in embedded control and monitoring applications.

Command Based

It is common for embedded applications to operate in a headless configuration, performing specific actions in response to commands coming from a user interface running on a remote host. These commands are often transferred from the remote host to the embedded system over Ethernet, and then distributed to the appropriate process on the embedded system where the command is executed. This type of data can be transferred over Ethernet using a LabVIEW mechanism such as Network Streams or a Simple TCP/IP Messaging (STM) reference library, which is a free download from ni.com.

Stream

Streaming communication involves the high-throughput transfer of every data point, where throughput is more important than latency. Typically, one process writes data to another process that reads, displays, or processes that data. An example is an in-vehicle data logger that sends data from the FPGA to the real-time controller for data logging. Streaming can be implemented using a FIFO-based mechanism in LabVIEW such as Network Streams for streaming data across the network, real-time FIFOs for streaming between real-time processes, or DMA FIFOs for streaming between an FPGA and a real-time target.

Current Value Data Storage

Machine control, automation, and application monitoring are typically all developed as independent processes running on one or more systems. Each process performs a distinct task but shares a set of common data or variables that allow successful completion of all application tasks. When designing a scalable application, it's often best to store this common data in a central location. This type of data can be stored and accessed with mechanisms in LabVIEW including shared variables and the Current Value Table (CVT) reference library, which is a free download on ni.com.

What Techniques Should You Use to Ensure Reliability?

NI embedded platforms offer high reliability by incorporating both a processor running a real-time operating system (RTOS) and an FPGA. Depending on your application requirements, you may need to spend some additional development time ensuring that your system is as reliable as it can be. If your system will be deployed with a required uptime of days, weeks, or years, you should minimize dynamic memory allocations within the real-time application. The memory on the real-time processor will become fragmented over time, and if the memory manager cannot find a contiguous block of memory large enough to support a requested allocation, the application will terminate. Systems that can handle regular reboots or maintenance can use dynamic memory freely as long as they monitor the memory status.

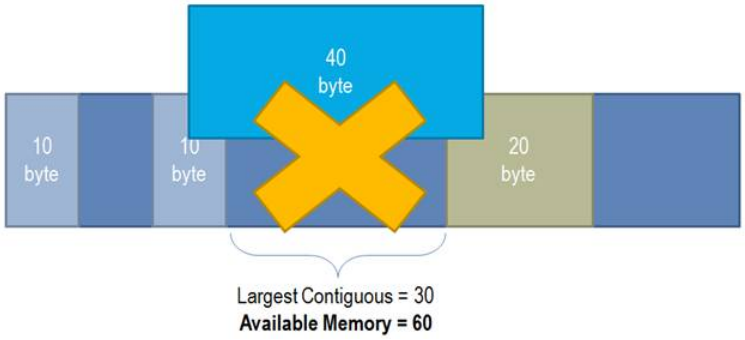

Figure 3. Eliminating dynamic memory allocations helps prevent the memory on an RTOS from becoming fragmented, which could potentially cause the real-time application to terminate.

Another technique to ensure the reliability of deployed embedded systems is to implement a watchdog timer. A watchdog timer is a hardware counter that interfaces with the embedded software application to detect and recover from software failures. An example of a software failure is your application running out of memory, causing your application to hang or crash. Even if you followed best practices for eliminating dynamic memory allocations, it's always important to have a backup plan. All CompactRIO and NI Single-Board RIO controllers include a built-in watchdog timer that you can access from the LabVIEW Real-Time Module. You can find tips for minimizing dynamic memory allocations and taking advantage of watchdog timers in the Designing a LabVIEW Real-Time Application section of the CompactRIO Developer's Guide.

» **For a comprehensive overview of NI recommended best practices and the many other techniques to consider when designing an embedded application, see the recently updated LabVIEW for CompactRIO Developer's Guide.**