**Motivation**

Like other major software development environments, the LabVIEW® IDE has long included a variety of debugging tools to help developers troubleshoot their own code. In addition to the standard debuging tools expected of any decent software development environment, such as Breakpoints, Single Stepping, and [Conditional] Probes (commonly called “Watch Variables” and “Watchpoints” in other languages), the graphical nature of LabVIEW also includes graphical-code specific debugging facilities, in the form of Execution Highlighting. Introduced with LabVIEW 2013, the *Event Inspector* added an ability to gain new understandings into the inner workings of your event structures. While all of these built-in tools are extremely useful during troubleshooting of individual VIs or small collections of VIs, particularly when you already know which VIs are the troublesome ones, the LabVIEW® IDE still lacks a debugging tool to provide insights with a “bigger picture” view of an entire project. *VI Analyzer* does its job well, but it intends to perform a **static** analysis of VIs, not a **dynamic** analysis of code while running. The separate *Desktop Execution Trace Toolkit (DETT)* product is also very good at what it was designed to do – but while execution traces can provide a fantastic wealth of evidence when event timing and/or sequence information is critical, they were never meant to provide other (non-event) kinds of comprehensive awareness into the instantaneous state of your dynamic code.

Using only the native troubleshooting tools, reentrant VIs are especially difficult to debug, since each preallocated clone will have its own data space in memory. Dynamically launched VIs, even if not reentrant, also present their own particular frustrations during a debug session, since the developer is unable to place probes BEFORE executing the code. How to defeat these shortcomings, and many other debuging related challenges, have long been the topics of many user community discussions; and indeed several ad-hoc solutions and processes, although usually quite limited in scope, have been developed over the years by the community – but they all lack the coherence of a unified tool.

**Solution**

The *LabVIEW Task Manager* seeks to be that missing unified debuging tool – to provide a dynamic & big-picture view of all VIs currently in memory, conquering those difficulties concerning reentrancy, clones, dynamic launching and other sticky complications, and allow for remotely interacting with VIs in various ways that make sense during debuging.

**History**

In September of 2011, a few community members discussed the need to create such a tool. Only a few days later a fledgling implementation was put forth by one very active community member, and then enhanced only two days after that by yet another highly respected community member. Soon several others would contribute code modules which were then rolled into the tool by the original author. Over the course of the next seven months the tool received some feature enhancements, optimizations, and improvements to error handling. In October 2012, the original author released a plugin which allowed LabVIEW Task Manager to be used from inside a built executable. LabVIEW 2013 included an under-the-hood change which broke the tool’s ability to detect clones. This was fixed in March of 2014, along with the addition of yet a few more modest improvements and bug fixes; plus the tool was at that time packaged up using VI Package Manager (VIPM) for easier distribution, installation, and use.

**What Is the LabVIEW Task Manager (LVTM)?**

*LabVIEW Task Manager* is a debugging tool for use during LabVIEW® code development. An expandable/collapsible tree diagram displays detailed information on all VIs in memory, belonging to a selected project/target. It allows for interacting with single or multiple selected VIs at a time, and includes the following major features:

* A Look & Feel similar to Windows Task Manager
* Selection of project/target
* Lists all VIs in memory, grouped by class/library
* Searches for and enumerates clones in memory
* DropIn VI for including dynamically referenced clones (Clone Beacon)
* 'Refresh Now' (F5) re-reads all VIs in memory and adds new ones to the tree
* Displays VI name, owning class/library, state, path, data size & code size
* Displays VI FP Behavior, Reentrant?, Reentrancy Type, Paused? & Highlight?
* Sort by any column, including by library name
* Filter by item types vi, ctl, and vit/ctt
* Filter out vi.lib and global VIs
* Tracking of, and ability to toggle, execution highlighting on multiple selected VIs
* Tracking of paused VIs with ability to Pause/Resume/TogglePause multiple selected VIs
* DropIn VI for pausing on a condition (a Conditional Breakpoint)
* If a clone initiates a pause, a different pause symbol is used for all clones of that same reentrant original VI
* Select multiple VIs and open or close their FPs or BDs
* Double Click a VI from the tree to bring the BD (first choice) or FP to front, if already open
* Select multiple top-level VIs and Abort them

**How Do I Use It?**

During LabVIEW® code development, invoke the LabVIEW Task Manager from your "Tools > LAVA" menu. Use the "App Instance" drop-down to select which of your loaded projects/targets you wish to manage. A tree diagram will show detailed information on all VIs in memory. You may interact with single or multiple selected VIs at a time.

**Helper Functions**

There are helper functions available to assist with your debugging effort, and these are made available from Functions Palette > Addons > LAVA > LabVIEW Task Manager.

* **Pause**

The Pause function allows for optionally initiating a pause from within any VI. Just drop this function into any VI that you wish to be able to pause, on an optional condition, **but only if LVTM is running** (so this pause will NOT occur if not debugging). If the pause request is generated from a clone of a reentrant VI, it will pause all clones of that VI. Leaving this function in your VIs, even when deploying your application, will not affect the performance, as it has a very small footprint.

* **Clone Beacon**

Asynchronously called reentrant VIs are not automatically recognized by the LabVIEW Task Manager, because they (by design) run in their own threads, independent from the rest of the project. This function remedies that problem. Just drop the "Clone Beacon" into any asynchronously called reentrant VI, to force its clones to be seen by LabVIEW Task Manager. For each reentrant clone that is created and run, it will add its own name to this Functional Global. LabVIEW Task Manager will then display "All VIs in Memory" **PLUS** the clones listed in this FG. Leaving this function in your VIs, even when deploying your application, will not affect the performance, as it has a very small footprint.