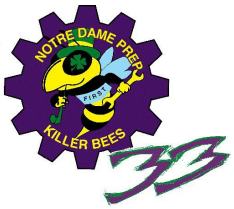

FRC #33 – The Killer Bees
2013 Software – WPIlib total rewrite
Design and architecture

Introduction:

Over the past few years, we've steadily noticed increases in compile, download, boot, and other significant development times in the LabVIEW development environment. Starting in 2012 we also began noticing CPU bandwidth issues, and we began to 'run out of' CPU. This seems strange, since we're using a 400mhz PowerPC which should be more than enough power for any control algorithm we could write in FRC, but we were hitting 100% CPU utilization, and realtime performance significantly suffered. Loops would loose timing determinism and miss entire iterations, and the robot would lock up or trip the watchdog when realtime displays and probes were open. We were also unable to download code without the help of the No-App DIP switch and later the imaging tool on the 4-slot cRio models, because our code ran at such high processor utilization that the LV bootloader would sometimes be unable to work properly, and the processor would reboot. In 2012, our we heavily investigated methods of improving code efficiency in LabVIEW, and learned many tricks (including use of subroutines, inlined VI's, and such) to speed up runtime execution. We found that the entire WPIlib appeared to completely ignore most of these rather simple execution changes, and the general layout (which required as many as 12 VI calls layered to write a single motor value to a PWM channel) did not lend itself to efficiency. We rewrote the Motor Set VI with a highly optimized one, at the cost of support of any controller except the Victor, and streamlined many of the function calls by copy-and-pasting low level code into the first level VI call. We also looked at many other functions in the WPIlib and our own code and inlined many of them, drastically improving code execution. We were able to run most of our code in 20ms loops with rather poor timing, at a low enough CPU usage to guarantee around 25ms worst case execution, and were still unable to download code without the no-app switch. The code ran, but barely.

This year, we dug through the majority of the WPIlib in LabVIEW and found many more cases of inefficient design, mostly caused by high-level design decisions that we did not support or requirements that we did not care about. For example, to set a relay or solenoid (which is stored in the FPGA as 4 uint8 bit registers, relay_fwd, relay_rev, and two solenoid modules), the VI determines which bit register the desired output is in, reads that register from the FPGA (slow operation), changes the one bit it wants, and writes it back (slow operation). For the relays, it performs this twice, once for the fwd and once for the rev channel. There are also calls to functions which validate that the object being written to was created, and that the module in question exists and has been configured. All of these conditions are checked for every iteration of every relay or solenoid in use.

Our solution was to essentially delete (not actually delete, but not include in our code) the entire WPIlib for LabVIEW and write our own interface layer to the FPGA, focused on efficiency at the expense of features we do not use. We implemented the entire library as five key VI's: Init, Postinit, Read FPGA, Read Control, and Write FPGA. There are a few other logical inlined functions which deal with data type conversions and ADC sampling, but the entire library contains only 30 VI's, most of which deal with scaling and are inlined for runtime performance.

Pal Lib:

Initial thoughts were to call this 'Buzz Lib', but we already named our 'useful VI' toolbox 'Buzz Lib', so I named this library 'Pal Lib'. The architecture is simple.

First, we initialize everything in Init. While Init is not terribly flexible, I setup the system for 1 ADC card with 1 gyro configured on channel 0, sampling all 8 channels at 1khz, two single-wire counters, two quadrature encoders, and all of the GPIO/PWM/Relay on a single Digital Sidecar, and a single Solenoid card (no config required during init). It is possible to enable up to 2 additional encoder4x and 6 additional counters, this is up to the user. It is also possible to configure a counter as dual-wire for quadrature decoding 1x or 2x, I have not done so. I also assume all GPIO are configured as inputs. If you need outputs, the easiest option is to use Relay outputs, as each output provides 2 buffered 5v logic signals (595-style driver updated at 5ms). The ports and calibrations for the encoders/counters/gyro are configurable via the front panel. The PWM frequency multipliers (5.05ms, 10.1ms, 20.2ms) are also configurable for the DIO module. The default WPIlib configuration uses 1x (5.05ms) for the Jaguar, 2x (10.1ms) for the Talon and Victor, and 4x (20.2ms) for a servo, but we have successfully used 1x for Talons and Victor 884's without issue. The configuration of Victor or Talon (or Jaguar even) is not done here, that is left up to the user code. I also configured the FPGA watchdog for 50ms timeout. We do not use any watchpuppies, only the FPGA's single watchdog. The previous WPIlib 'watchpuppies' were purely software constructs, in a single-threaded program a single FPGA watchdog is sufficient and significantly reduces CPU load over the watchpuppy implementation.

After Init, we run Postinit. Postinit simply executes the code which must run after Init but before the loop starts. Notably, we reset two Encoders and a gyro, call observeUserProgramStarting, and begin the auton parser (which runs asynchronously once at boot). I removed the auton parser call from the pal_lib_default.zip release as it is user-code specific, but I left the encoder/gyro reset.

The code then executes from a single 10ms RT loop. I have measured timing accuracy and jitter and the numbers are quite good, averaging around 20us of jitter in our code project. I believe based on the CPU loading currently that I could run the 2013 Buzz18 code as fast as 7ms, but this leaves little overhead for probes and realtime debugging, which consume fairly significant CPU resources. As part of the single RT loop, we store 4 data structures between iterations, as the primary method of data storage. bus_state contains the state of subsystems and is the primary method for communicating between them, bus_inputs and bus_outputs contain IO data scaled, and control_data stores the latest data from the Driver Station (it is received asynchronously via UDP but read from the intermediate buffer in Netcomm and processed every iteration). At the beginning of the loop, we read the inputs using read_fpga, scale them to engineering units in input_scale.vi (part of user code), execute the user code, calculate the raw outputs from engineering values in output_scale.vi (part of user code), including building bit registers for bit outputs, and write the values to the FPGA in write_fpga. The watchdog is also set in read_fpga. All of the Netcomm interaction happens in control data, including writing error and dashboard return data. We use the 'old style' binary string dashboard method instead of NetworkTables, since it is so easy to send the three busses (bus_state, bus_inputs, and bus_outputs) to the dashboard directly.

Usage:

Usage is simple. There is an example Main.vi which contains an example program. The write_fpga block requires uint8 motor values, you can use motor_talon.vi, motor_victor.vi, etc. to scale a 'standard' decimal double float to uint8 appropriate for that controller. We used all Talons, as does this example, but used several Victors on our practice robot with the Talon block without issues (the pulse profile of a Talon and Victor is almost identical). The Jaguar requires significantly different pulse profile than the Talon or Victor, I recommend using a Jaguar block when using a Jaguar or calibrating the Jaguar to the other pulse profile. We do not support CAN at all (the current WPIlib CAN stack is highly not recommended, as it contains many blocking nodes and other garbage) and have no plans to support it ever.

Example program functionality:

The example program provided (main.vi) and mapping code (code.vi) contains the following code:

-Tank Drive from a Logitech gamepad to Talons
-Mapping several additional buttons and joysticks on the primary and secondary joystick to Talons, Relays, and Solenoids (in several cases to multiple outputs).
-Control of the compressor
-No provisions for autonomous code, however a VI to determine if autonomous enabled is the current state is provided.

The example program is intended to be used as a baseline for code structure using this framework, or as complete (but not supported) code to run a relatively simple robot (analogous to the Default Code provided with the IFI control systems). I have not tested this code, it may contain unforeseen bugs.