Controlling the Daedal Systems

There are many, many programs built to control the Daedal positioning and data acquisition systems at the BRL. In this series of how-tos, I'll go over the different parts of controlling them that you'll need to know to make your own control programs.

Part 1: Understanding the Basic Setup

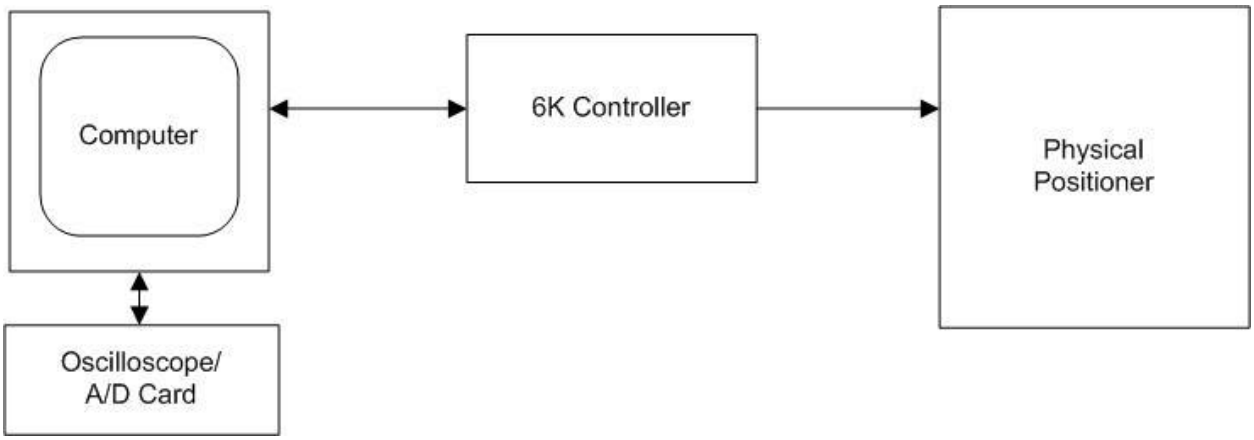

Figure 1: Basic Blocks and Dataflow

While the computer and the positioner may be the only parts you see, Figure 1 shows the four basic components involved in data acquisition. The computer, and more specifically suite of programs in Daedal Menu, is the "brain" of the whole operation.

- The Computer sends control commands to the 6K Controller
- The Computer sends control commands to the Oscilloscope or A/D card.
- The 6K Controller tells the Physical Positioner what to do and tracks its state.
- The 6K Controller responds to commands from the Computer, usually requests for position and movement status
- The Oscilloscope or A/D, while running, is constantly sending waveform information back to the computer.

If this doesn't all make sense right now, don't worry. We'll elaborate more on this in later sections.

Part 2: Control Commands

Before you worry about figuring out how to use LabVIEW or any other programming language, you'll want to get to know Compumotor's motion-control program: Motion Planner. This program is available on their website at http://www.parkermotion.com/literature/pg030_6k_software.htm

The main idea when controlling the Daedals is to send movement commands to the controller via LabVIEW. These commands could be sent with any other program, since they're just text commands sent over a serial cable, but LabVIEW allows for a great deal of customization with an easy-to-learn programming language. LabVIEW is not special in its ability to control the Daedals, it just happens to be the best method right now.

Here is an example of a basic movement command for Daedal 2:

```
-----
AXSDEF00000
CMDDIR00100
DRES50800,50800,50800,36000,36000
LH3,3,3,0,3
ERRLVL0
LS0,0,0,0,0
A5,5,5,5,5
AD5,5,5,5,5
V1,1,1,1,1
D10,10,10,10,10
DRIVE11111
GO10000
-----
```

While this might seem complicated, many of these lines you'll never really have to worry about:

- ***AXSDEF00000*** – tells the controller that all our movement axes are steppers, not servos (true for all Daedal systems)

- ***CMDDIR00100*** – means axes 1,2,4,5 move in the positive direction relative to their motors, axis 3 moves in the negative direction

- ***DRES50800,50800,50800,36000,36000*** – sets the resolution of the motor for each axis. Axes 1,2,3 are linear, with 50800 steps per revolution. Axes 4,5 are rotational, with 36000 steps per revolution. These values let us send command distances in millimeters and degrees, respectively, instead of having to scale our distances into motor steps

- ***LH3,3,3,0,3*** – hard-limits axes 1,2,3,5 so they stop if they hit an end. Axis 4 can rotate > 360 degrees, so it does not have to be limited. NOTE: hard limiting the axes is a good idea, but is not 100% effective in practice. Users still need to watch what theyre doing to avoid hitting the ends

- ***ERRLVL0*** – clears any errors before movement, in case something went wrong before but is fine now

- ***LS0,0,0,0,0*** – Do not "soft-limit" any axes. The soft-limit command checks for sensor input before starting motion, so the axes cannot move past its limit sensors. Daedal 2 does not have any limit sensors connected, so if these values were '1's motion would never begin.

These commands will always be the same per system. For example, Daedal 2 will ALWAYS use the values above, no matter how you're trying to move the system. However, Daedal 4 will require different values than above, since the two are set up differently. But Daedal 4 will have its own set values that should never change.

Each digit corresponds to one axis. So 00111 means "assign '0' to axes 1,2 and assign '1' to axes 3,4,5." For the commands with commas, the systems is the same, but with commas between axes. The commands without commas can only take values of 1 and 0; the commands with commas can take larger values.

The next set of commands are specific to the movement you're making. Save your questions until the end…

- ***A5,5,5,5,5*** – set acceleration on all axes to be 5 (mm/s^2 on 1-3, deg/s^2 on 4,5 thanks to our DRES scaling)
-
- ***AD 5,5,5,5,5*** – set deceleration on all axes to be 5 (mm/s^2 on 1-3, deg/s^2 on 4,5)
-
- ***V1,1,1,1,1*** – set velocity on all axes to 1 (mm/s on 1-3, deg/s on 4,5)
-
- ***D10,10,10,10,10*** – set distance to move on all axes to 10 (mm or degrees)
-
- ***DRIVE11111*** – power all axes. This is important because unpowered axes can slip/rotate even if you aren't moving on it
-
- ***GO10000*** – only move axis 1. This is how you can set all the movement parameters to be nonzero values and not move all axes at the same time.

This entire command moves axis 1 a total of 10 mm at a velocity of 1 mm/s, accelerating and decelerating at 5 mm/s^2. There is a lot of extra information going in, for example acceleration, deceleration, velocity, and distance don't need to be set on axes 2-4. But putting in zero values for these fields ends up being the same as putting in real values, so it doesn't really matter. In practice, you'll find it easier to write into your code "set all values to be the same on all axes" when moving on one axis so you don't have to make as many special cases. A visual example of this below:

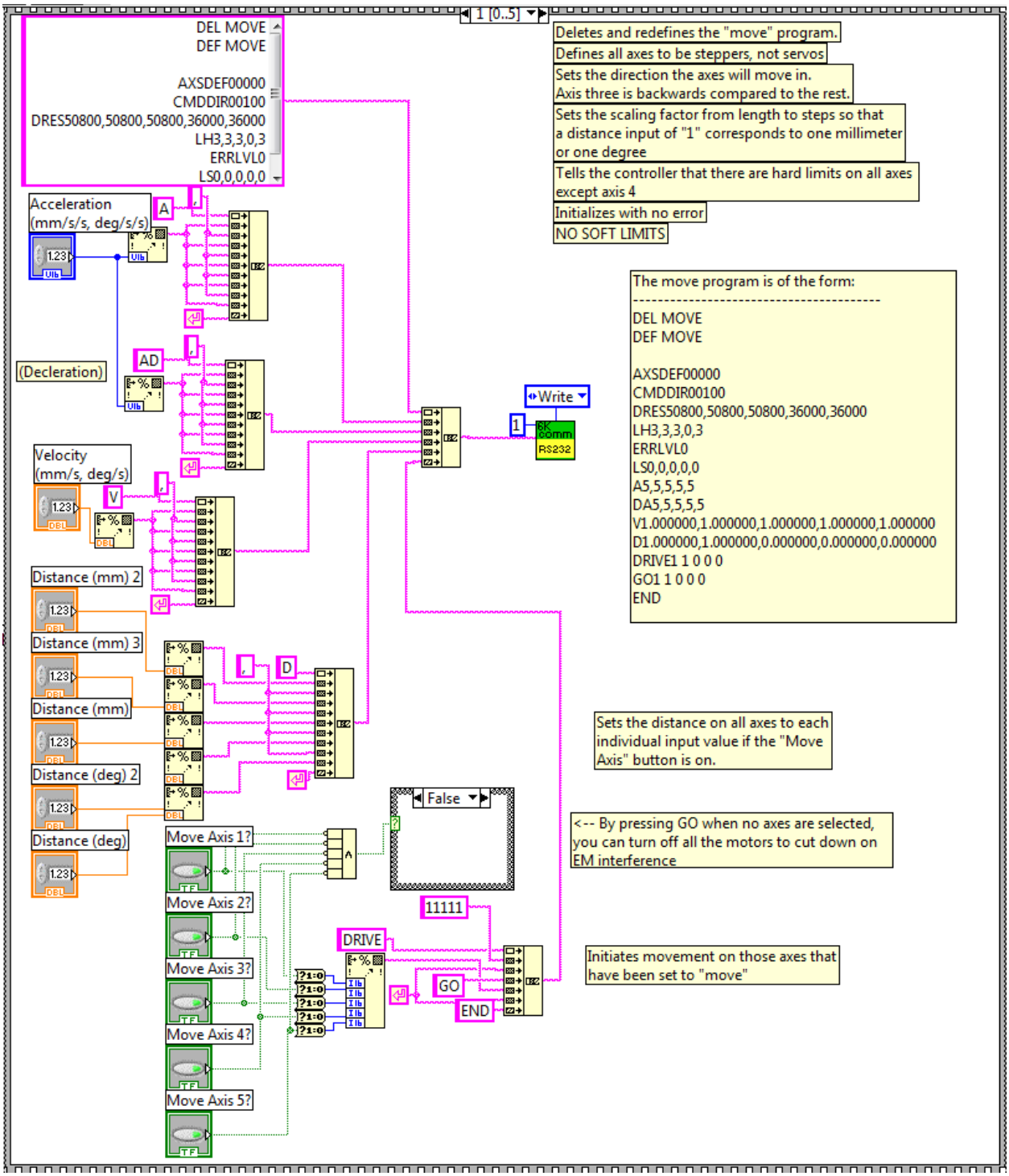

Figure 2: How "Position" Builds a Move Command

By setting the acceleration, deceleration, and velocity to be the same on all axes, we require fewer fields on the front panel and fewer operations. In fact, in the above example, acceleration and deceleration come from the same field, as there's no real need for them to be different for our uses. This practice will be discussed later in the program-building guidelines, but for now think of it as a way to make your life easier when writing code.

Part 3: Building and Using Motion Planner Scripts

In Figure 2, you'll notice that other than the commands we talked about before, there are three new lines:

```
DEL MOVE
DEF MOVE
…
END
```

These two commands don't have anything to do with the motion of the system, but they are very important to the general practice of sending commands to the 6K controller. While commands can be sent to the controller line by line, sending multiple commands slows down the controller's reaction time. By sending these commands packaged together into a script, we gain two advantages:

1. By packaging our commands this way, it speeds up processing of the 6K, as it doesn't have to register each line individually.
2. Now that we've defined the entire command into a script, if we want to repeat the same movement we don't have to redefine all the parameters – we just tell the controller to run the script again.

As for their usage, it's pretty self-explanatory:

- ***DEL MOVE*** – Delete the script named "MOVE" if it already exists. To edit/change an existing program, you must delete it first.
- ***DEF MOVE*** – Define a new script named "MOVE," containing all the lines that follow before "END." The name of program defined must be six characters or less by limitation of the controller.
- ***END*** - Tells the controller that this is the end of the script, so close and save it.

Now that we've built this script, we only have to tell the controller "MOVE" in the future to run all of the commands inside it. This is particularly useful for programs like "Two D," where the scanning process can involve repeating the same move thousands of times.

Part 4: Interrupting Motion

No program is perfect. Even more so, no user is perfect. Combine those two and your motion program will eventually move too far for a user, and possibly risk breaking thousands of dollars of equipment. To avoid this, every motion program you write should have a large, clearly labelled “STOP” button.

Figure 3: Salient and Satisfying

The most important thing about this stop button is that it has to be an interrupt: its function should be completely asynchronous from your motion program and should operate the millisecond that it is pushed. In the LabVIEW programming guide, we’ll talk about how to create this condition, but before we worry about any of that we have to know how to send an immediate-stop command to the controller.

Luckily, the 6K language uses the “!” character to mean “perform immediately.” The “K” command means “kill all motion,” so we’ve got that covered to. So just have the STOP button immediately send “!K” to the controller and you’re done!

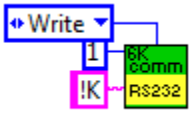

Figure 4: Send "!K" for an Emergency Stop

Questions?

For a full listing of what each of these commands does and how to use them, there is an index in the Compumotor Motion Planner software.

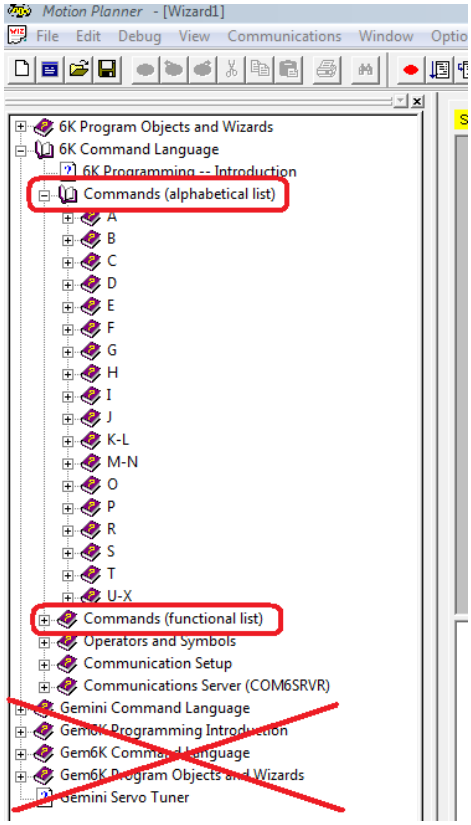

Figure 5: Commands Indexed Alphabetically and by Function

All of our positioning systems use 6K controllers, so don't worry about the Gemini command language.