The Problem

In the past, Graphite font features were given mnemonic IDs such as ‘litr’, ‘viet’, and ‘dig4’. Now that OpenType supports character variants and stylistic sets, you may want Graphite font tables to use IDs that are compatible with OpenType’s, such as ‘ss01’ and ‘cv24’. However, we can’t simply remove the old IDs from the font since that would break backward compatibility.

The Solution

The solution is to allow GDL feature definitions to define multiple IDs. This in effect creates multiple features with identical behavior in the font. The new ID is offered to the user in UI mechanisms; the old ID is not offered but is still effective for existing data that might use it.

Example

An older Graphite font has a feature controlling which form of the uppercase eng is used.

```
eng {
      id = "Engs";
      name.1033 = string("Uppercase Eng alternates");
      default = descender;
      settings {
            descender {
                  value = 0;
                  name.1033 = string("Large eng with descender");
            }
            base {
                  value = 1;
                  name.1033 = string("Large eng on baseline");
            }
            capital {
                  value = 2;
                  name.1033 = string("Capital N with tail");
            }
            short {
                  value = 3;
                  name.1033 = string("Large eng with short stem");
            }
      }
}
```

We need to extend the code to use the feature ID ‘cv43’ as well as ‘Engs’.

Implementing this solution in a Graphite font has several aspects.

GDL Feature Definition

A feature definition can be extended to include multiple IDs. All but the first ID must be marked “hidden”.

```
featureGdlName {
      id = “cvXX”;     // ie, an ID that is compatible with OpenType
      id.hidden = “prev”;   // a mnemonic ID
      default = default_value;
      etc.
};
```

Features can be assigned multiple hidden IDs. The keyword following “id” is somewhat arbitrary but it must be unique and must start with the string ‘hidden’; e.g.,

```
featureGdlName {
      id = “cvXX”;          // OpenType compatible
      id.hidden = “prev”;
      id.hidden2 = “oldr”;
      id.hidden_obsolete = “obso”;
      default = default_value;
      etc.
};
```

Note that using a “hidden” ID can be used to simply create a feature with a single ID that is not shown in the user interface but may still be accessed by those who know the “secret code.”

Feature Testing

As indicated above, including a hidden ID actually causes the resulting Graphite font to have an extra feature with that ID. This means that the GDL code that tests a feature must explicitly test all the features that result from the single GDL definition--the public feature as well as any hidden features.

A special syntax is available to test hidden features, consisting of the GDL name followed by ‘__’ (that is, two underscores) and then the hidden ID.

So if your original GDL looked like

```
if (featureGdlName == 2)
```

the new code would be

```
if (featureGdlName == 2 || featureGdlName__prev == 2)
```

The ID can also be included for public features, so the following is also valid:

```
if (featureGdlName__cvXX == 2 || featureGdlName__prev == 2)
```

Testing Default Values

Special care must be taken when testing default values. For a non-default value, if either the public feature or the hidden feature is set to that value, the feature will be considered activated. But notice that the other feature will (most likely) still have the default value. This means that only if *both* the public feature and the hidden feature have the default value can the feature be considered to use the default setting. For this reason, default features should generally be tested using a logical AND operation rather than OR:

```
if (featureGdlName == 0 && featureGdlName_prev == 0)
// rules to run in the default case
```

Incompatible Values

This raises the question of how to handle incompatible values. For instance, what if cvXX = 2 and prev = 1? This could happen if the user has some existing data marked with prev = 1 and then uses the UI mechanisms to set cvXX to 2.

Here is a GDL test that takes this possibility into account:

```
if (featureGdlName == 2
    || (featureGdlName__prev == 2 && featureGdlName == 0))
```

In other words, the hidden feature is only taken into account when the public feature is set to the default.

Example

Here is what the GDL code to handle the uppercase eng feature would look like:

```
table(feature)
eng {
     id = “cv43”;
```

```
	id.hidden = "Engs";
	name.1033 = string("Uppercase Eng alternates");
	default = descender;
	settings {
		descender {
			value = 0;
			name.1033 = string("Large eng with descender");
		}
		base {
			value = 1;
			name.1033 = string("Large eng on baseline");
		}
		capital {
			value = 2;
			name.1033 = string("Capital N with tail");
		}
		short {
			value = 3;
			name.1033 = string("Large eng with short stem");
		}
	}
}
endtable;

table(substitution)

if (eng == descender && eng__Engs == descender)
// rules to produce large eng with descender
endif;
if (eng == base || (eng__Engs == base && eng__cv43 == descender))
// rules to produce large end on baseline
endif;
if (eng == capital || (eng__Engs == capital && eng__cv43==descender))
// rules to produce capital N with tail
endif;
if (eng == short || (eng__Engs == short && eng__cv43 == descender))
// rules to produce large eng with short stem
endif;

endtable;
```