

The “AplFunction” Attribute

Visual APL supports both the **method** (standard C#) and the **operator** (traditional APL) function signatures.

In VisualAPL or (C#) the **method** function signature encloses the argument list in parentheses, separates the arguments by commas, uses the “return” keyword to define the function result, and often strong-types the arguments and result. The “function” keyword or the “Alt+f” (“f”) keyboard short-cut may be used.

```
function result_type My_Fn(arg1_type arg1, arg2_type arg2,...)
```

```
{
```

```
// Body of function defining result variable
```

```
return result
```

```
}
```

The **operator** function signature is unique to VisualAPL. The function arguments (optional left and right) are separated from the function name by a space. The function may use the “return” keyword to define the function result or may explicitly describe the result variable in the function signature. The result may be dynamically- or strongly-typed, but strong typing is not supported in the function signature. The “operator” keyword or “Alt+g” (“v”) keyboard short-cut may be used. Dyadic (both left and right arguments present), monadic (only a right argument present) and niladic (no arguments present) function signatures may be created. A method declared by the operator header syntax in a VisualAPL .Net assembly produces a method which is completely CLS compliant. Other .Net languages which do not recognize the operator signature will still be able to call the function using the method function signature calling syntax.

```
Operator left_arg My_Fn right_arg (or vresult←left_arg My_Fn right_arg)
```

```
{
```

```
//Body of function defining result variable
```

```
}
```

Visual APL implements user-defined operators, with the traditional APL function signature syntax, by use of the “AplFunction” attribute. When a programmer declares a VisualAPL function using the operator signature, the VisualAPL method created has the “AplFunction” attribute automatically applied to it.

The “AplFunction” attribute can be used in non-VisualAPL .Net assemblies, for example those created using C# or VB.Net. For example, a C# function using the standard .Net function signature which also has the “AplFunction” signature applied to it, can be called from a VisualAPL .Net assembly using the VisualAPL operator function signature calling syntax. There is no restriction on the return and argument types. A

reference to the VisualAPL "APLNext.APL.DLL" .Net assembly must be made in the C# .Net assembly and a "using directive" for the APLNext.APL.Objects namespace must be included in the C# .cs file. For such a C# function to be consumed using the traditional operator function signature calling syntax by Visual APL, the C# method must include the public keyword in the function signature.

When VisualAPL calls a method in a non-VisualAPL class which does not include the "AplFunction" attribute, the C# standard function signature calling syntax is necessary, e.g. the arguments are enclosed in parentheses and separated by commas.

If VisualAPL is calling an "instance", i.e. non-static, C# method with the "AplFunction" attribute, the VisualAPL variable which is assigned the instance of the C# class must be strongly-typed, e.g. "MyCsNamespace.MyCsClass cs_instance = new MyCsNamespace.MyCsClass()" or else the C# method must be called using the C# function signature calling syntax. The strong-typing of the instance of the C# class is required, because the VisualAPL class must "know in advance" that the instance of the C# method can be called using the traditional APL function signature calling syntax. This will prevent ambiguity in the call of the C# method which could be erroneously interpreted as .Net property reference instead. A benefit of using an instance method is that dynamically-created properties in the C# class can be accessed and modified from VisualAPL using the standard .Net syntax, e.g. "MyCsNamespace.MyCsClass.MyCsProp1 = ..." .

Here are C# methods which can be called by VisualAPL using the traditional APL dyadic, monadic and niladic operator function signature calling syntax:

```
using APLNext.APL.Objects;
...

namespace MyCsNamespace{

    public class MyCsClass{
        ...

        [AplFunction]

        public static result_type FnD(right_arg_type right_arg, left_arg_type left_arg){...

        //^Dyadic
        ...

        }

        [AplFunction]

        public static result_type FnM(right_arg_type right_arg){...

        //^Monadic
        ...
```

```

    }

    [AplFunction]

    public static result_type FnN(){...

    //^Niladic
    ...

    }

```

To call this function from VisualAPL a reference to the C# .Net assembly must be made in the VisualAPL class library and a "using directive" to the C# class must be included in the VisualAPL .apl file.

```

Using MyCsNamespace.MyCsClass;
...

resD = left_arg FnD right_arg

resM = FnM right_arg

resN = FnN

```

The C# method using the "AplFunction" attribute may specify the argument data type(s). For example if the C# method specifies an argument type as "string" then this will insure that VisualAPL will convert, if possible, data passed from the VisualAPL class to the C# class method argument as a string datatype. If the C# method will support argument(s) of different data types, "overloads" may be used, or the "object" data type may be specified by the C# method.

If a C# class publishes to methods with the "AplFunction" attribute with the same function name but different arguments or data types, the appropriate one will be used when that C# method is called by VisualAPL depending on the calling syntax used by VisualAPL.

The C# method must be static because operators do not themselves have a persistent state. Instead, they rely on the state of the VisualAPL class which called the operator.

Using the C# methods with the "AplFunction" attribute can optionally receive the "module" of the VisualAPL class calling the C# method. Using the module, the C# method can query class state information, such as Index Origin (IO), Random Link (RL), Divide by Zero (DBZ), etc. The C# method using the "module" argument may also access any dynamic variables, methods, properties, events, etc, in the VisualAPL class calling the C# method. To have the VisualAPL calling class module passed to the C# method with the "AplFunction" attribute, the first argument of the C# method must be of type "module".

Here are C# methods incorporating the "module" argument which can be called by VisualAPL using the traditional APL dyadic, monadic and niladic operator function signature calling syntax:

```

using APLNext.APL.Objects;
...

namespace MyCsNamespace{

    public class MyCsClass{
        ...

        [AplFunction]

        public static result_type ModFnD(module mod_arg, right_arg_type right_arg,
left_arg_type left_arg){...

            //^Dyadic
            ...

        }

        [AplFunction]

        public static result_type ModFnM(module mod_arg, right_arg_type right_arg){...

            //^Monadic
            ...

        }

        [AplFunction]

        public static result_type ModFnN(module mod_arg){...

            //^Niladic
            ...

        }
    }
}

```

To call this function from VisualAPL a reference to the C# .Net assembly must be made in the VisualAPL class library and a "using directive" to the C# class must be included in the VisualAPL .apl file.

```

Using MyCsNamespace.MyCsClass;
...

resD = left_arg ModFnD right_arg

resM = ModFnM right_arg

```

resN = ModFnN

Note that the module argument is not included in the VisualAPL call to the C# method using the "AplFunction" attribute and the module argument. It is the C# method with the module argument which may query the VisualAPL "state" when it is called from the VisualAPL class. The C# method accesses the "state" of the VisualAPL calling class using the properties of the module argument, e.g. "mod_arg.IO" for the current value of the APL index origin in the VisualAPL calling class.

Additional information and more examples of this topic may be found by searching for "AplFunction" in the "...\\AplNext\\VisualAPLProfessional\\Documentation\\Visual_APL_Help.chm" file installed with VisualAPL.