The code base has four DLLs:

- PLJsAPL
- CommandLine
- AltKeys
- Tools

And two EXEs:

- ShellAPL
- EditAPL

DLL PLJsAPL provides the traditional APL functions and operators. While it is entirely a compiled product, all of the semantic variability is accounted for in this DLL. The key classes are:

APL Which controls the value vector and shape vector. An APL program will create one of these for every variable. It also provides the syntax structure for invoking dyadic functions in a pseudo infix manner. For more details on the latter, see the documentation in Statements.

_APL contains all of the built in functions and operators. The various types of functions are broken up into different files, but each are labeled a Partial Class so that one object instance handle will access all of them. The files included here are:

| | |
|---|---|
| _APL | contains the mathematical scalar functions. |
| _Compare | contains all ⎕CT aware functions. |
| _Random | contains all ⎕RL aware functions. |
| Conjunct | contains the four conjunctions and not. |
| Content | contains the functions which handle arrays as element containers. |
| Does | contains the built in operators. |
| Values | contains ⎕, ⎕AV and ⎕TS. It also provides Value for creating constants and Empty for indicating a missing term when subscripting. |

Method supports the execution of derived functions, i.e. what an operator returns.

OpMonadic provide an execution framework for monadic operators.

OpDyadic provides an execution framework for dyadic operators.

OpScalar provides a hidden operator to support non scalar and user defined functions in the domain of operators which assume scalar functions. N.B. All of the built in functions were written assuming their arguments were scalar dyadic functions.

Context provides the callback functions to provide Quad's Out, QuoteQuad's In, and Signal. The latter is much like IPSA QuadSignal.

ExceptionAPL is the exception type which QuadSignal throws.

Indexing computes which element is next when executing several content manipulating functions such as Sub, Select, Pivot and Transpose.

WorkSpace isn't strictly required, but provides a base class for writing classes which use PLJsAPL. For example, Inherits WorkSpace begins every class the code translator creates.

DLL CommandLine provides the non language functionality users expect from an APL system. The key class is Commands, where the method Commands analyzes each line it is sent to determine the four possible cases:

Begins with a comment, and is therefore logged and ignored.

Begins with a ∇ which attempts to create a method by the following name, and then invokes )EDIT on the result. It can also be used instead of )EDIT on an existing function.

APL statement which is translated from APL to .Net, then executed and the result displayed.

System command which starts with ). All work associated with fulfilling this request takes place within this object. In particular, this includes:

)SAVE which turns values into XML and writes this form of the workspace to the file system. If the workspace contains methods (which in PLJsAPL include Functions, Operators and Properties) they are translated into a class with appropriate .Net methods and then saved to the file system.

)LOAD which turns a file with XML content into a live workspace.

)COMPILE which ensures that the project file contains pointers to all the current workspaces which contain methods and then invokes the appropriate Visual Studio environment.

CommandLine also provides the system functions described in the user documentation.

For historical reasons, Execute lives in PLJsAPL, but gets all of its intelligence for how to provide APL functionality from CommandLine. At this point, I believe a proper implementation would extract Execute from PLJsAPL and put it in its own DLL. This would facilitate providing different versions of this facility to support translation into other languages than VB. I am only aware of two things which can be gained by making this move. One is the better press C# has received, even though it is still well behind VB in functionality. I believe that is because it looks so much like Java. Secondly, any APL user who is determined to use case sensitive names will prefer C#, since its names are case sensitive. Having used both APL and VB for decades, I much prefer having case preserved for readability and ignored when writing programs.

DLL AltKeys is a simple hack to provide entry of APL characters. It has two clear flaws. I've never learned how to avoid the bell warning, or prevent .Net from assuming I mean to invoke a menu item. As a result, this DLL also provides a very weak excuse for a text editor which all the functionality available via buttons and what windows users have come to consider traditional Control key shortcuts. In addition to having a built in translate table for Alt and Shift Alt key sequences, there is also the possibility of providing a control file called KeyPages.apl which overrides the built in table. A sample named KeyPagesX.apl is provided with the release.

DLL Tools is a slightly modified version of a tool kit I've been providing to VB programmers for years. The classes are as follows:

Files contains methods Lines and Bytes which are used by `⎕Lines` and `⎕Bytes` to provide access to UTF-8 and ANSI files respectively. There are also methods to handle such files as streams, but no APL interface is provided.

Host is expose by `⎕Host`. It provides paths to Desktop, MyDocuments and Temp. It also has methods to launch batch commands and fetch web pages.

Names provides several file name constructing functions and the basis for `⎕Paths` and `⎕Files`. Note to self, I can no longer remember what is achieved by providing `⎕Names`.

EXE ShellAPL provides the traditional shell script behavior many APL programmers have come to expect. Other than trying expressions and writing code in APL, it is not something I expect you would want to ship to customers who have come to expect from modern windowing products. It does only a small amount of the work itself.

It relies on the classes in the AltKeys for all APL typing behavior. It is probably unneeded in a windowing product.

During initialization, it supports two command line arguments:

Lib can establish an alternate path to locate workspaces as shell scripts. The default location is `(,⎕Host.MyDocuments),"\PLJ\PLJsAPL"`
Either the default, or user specified path are provided when It creates an instance of the Commands class in the CommandLine DLL.

`Start` if provided is the full path to a shell script which will be run first. Please note, since all types of lines are subject to scripting, the shell script can end with )OFF and provide a batch process.

It also determines if the last session was closed to do a )COMPILE. If so, it loads the continue workspace, restores the session log to the screen, and reestablishes the expected workspace name.

After initialization, it passes each line you type to the Commands method and relies on it for most of the behavior the user expects. It provides event handlers for the events CommanLine provides.

It also handles events for input ⍞ input and ⎕ output from PLJsAPL. Please note, APL programs written to work with windowing software probably won't use ⍞ and ⎕ and so they don't need to provide similar event handlers.

EXE EditAPL is little more than a call to the simple text editor in AltKeys. I provide it only because I like to look at files containing APL text directly rather than starting the language processor and using its tools. Shell scripts are presumed to have the .APL extension. In windows, it is relatively easy to indicate I wish to always view them with this program, so I do.