

Coding Conventions for Igor Pro

Thomas Braun <thomas.braun@byte-physics.de>

21.01.2016

Version: 0.13

1 Procedures

- Always put code into external procedure files stored directly on disk
- Filenames are built from the characters [A-Za-z_-] and end with .ipf
- The file encoding is OS-dependent but the used charset should always be restricted to ASCII characters. Code parts exclusively used with Igor Pro 7 should use UTF-8 as text encoding and specify `#pragma TextEncoding = "UTF-8"`.
- The beginning of each procedure file has `#pragma rtGlobals=3` with optional comment.
- Always use UNIX (LF) end-of-line style

2 Whitespace and Comments

Comments

- Use doxygen for documenting files, functions, macros and constants
- Always add a space before a trailing comment as in

```
if(a < 0)
    b = 1
else // positive numbers (including zero)
    b = 4711
endif
```
- Prefer comments on separate lines instead of trailing comments

Doxygen

- Use `///` to start a doxygen comment and `///
for documentation after the definition`
- Align multiple `@param` arguments and document them in the same order as in the function signature:

```
/// @param pressure      Pressure of the cell
/// @param temperature   Outdoor temperature
/// @param length        Length of a soccer field
Function PerformCalculation(pressure, temperature, length)
    variable pressure, temperature, length

    // code
End
```

- Use in/out specifiers for `@param` if the function uses call-by-value and call-by-reference parameters.

```
/// @param[in] name      Name of the device
/// @param[out] type      Device type
/// @param[out] number    Device number
Function ParseString(name, type, number)
    string name
    variable &type, &number

    // code
End
```

- Optional parameters are documented as

```
/// @param verbose [optional, default = 0] Verbosely output
///               the steps of the performed calculations
Function DoCalculation([verbose])
    variable verbose

    // code
End
```

Whitespace

- Every function should be separated by exactly one newline from other code
- Indentation is done with tabs, a tab consists of four spaces (in case you are coding not in Igor Pro).
- Comments on separate lines have the same indentation level as the surrounding code

- Separate function parameters from local variables and local variables from the rest of the function body by a newline

```
Function CalculatePressure(weight, size)
    variable weight, size

    variable i, numEntries

    // code
End
```

- Add a space around mathematical/binary/comparison operators and assignments, and add a space after a comma or semicolon

```
a = b + c * (d + 1) / 5

if(a < b)
    c = a^2 + b^2
end

Make/O/N={1, 2} data

for(i = 0; i < numWaves; i += 1)
    a = i^2
endfor

if(myStatus && myClock)
    e = f
endif
```

- Try to avoid trailing whitespace, here space is `␣` and tab is `↵`
Good:

```
Function DoStuff()
    ↵print␣"Hi"

    ↵if(a␣<␣b)
    ↵    ↵c␣=␣a^2␣+␣b^2
    ↵end

    ↵Make/O/N={1,␣2}␣data
End
```

Bad:

```
Function DoStuff()␣
    ↵print␣"Hi"␣    ↵
```

```

%|
%|if(a<b)
%|    %|c=a^2+b^2    %|
%|end
%|
%|Make/O/N={1,2}data
End

```

- Surround blocks like `if/endif`, `for/endfor`, `do/while`, `switch/endswitch`, `strswitch/endswitch` with a newline if what they express is a logical group of its own

```

for(i = 0; i < numEntries; i += 1)
    // code
endfor

```

```

if(a > b)
    c = d
elseif(a == b)
    c = e
else
    c = 0
endif

```

```

switch(mode)
    case MODE1:
        a = "myString"
        break
    case MODE2:
        a = "someOtherString"
        break
    default:
        Abort "unknown mode"
        break
endswitch

```

According to that reasoning the following snippet has no newline before `for` and `if`

```

numEntries = ItemsInList(list)
for(i = 0; i < numEntries; i += 1)
    // code
endfor

```

```

NVAR num = root:fancyNumber
if(num < 5)

```

```

    // code
endif

```

When mutiple end statements match

```

for(i = 0; i < numEntries; i += 1)
    // code

    if(i < 5)
        // code
    endif
endifor

```

you should not add a trailing newline.

- There is no whitespace between different flags of an operation and no whitespace around = if used in a flag assignment.

Good:

```
Wave/Z/T/SDFR=dfr wv = myWave
```

```
Function/S DoStuff()
    // code
End

```

Bad:

```
Wave /Z /T /SDFR = dfr wv = myWave
```

- The & in a call-by-reference parameter is attached to the name

Good:

```
Function DoStuff(length, height, weight)
    variable &length, &height, &weight

```

```

    // code
End

```

Bad:

```
Function DoStuff(length, height, weight)
    variable& length, & height,& weight

```

```

    // code
End

```

3 Code

3.1 General

- Line length should not exceed 80 characters

- Use `camelCase` for variable/string/wave/dfref names and `CamelCase` for structures
- Prefer structure-based GUI control procedures over old-style functions
- The variables `i`, `j`, `k`, `l` are reserved for loop counters, from outer to inner loops
- Use free waves for temporary waves
- Write your code as much as possible without `SetDataFolder`. Properly document if your function expects a certain folder to be the current data folder at the time of the function call. Always restore the previously active current data folder before returning from the function.
- Although Igor Pro code is case-insensitive use the official upper/lower case as shown in the Igor Pro Help files for better readability

```
Make/N=(10) data
AppendToGraph/W=$graph data
WAVE/Z wv
SVAR sv = abcd
STRUCT Rectangular rect
print ItemsInList(list)
```

except for the following two cases:

```
variable storageCount
string name
```

- Variable and function definitions and references to them must also never vary in case
- Don't use variables for storing the result which is then returned.

Good:

```
if(someCondition)
    // code
    return 0
else
    // code
    return 1
endif
// if it is important to know that the returned value
// is a status, name the function something like GetStatusForFoo
// and/or use the @return doxygen comment for explaining its meaning
```

Bad:

```
variable status

// code

if(someCondition)
    // code
    status = 0
else
    // code
    status = 1
endif

return status
```

- Avoid commented out code
- Don't initialize variables and strings if not required and always initialize variables in their own line.

Good:

```
variable i = 1
variable numEntries, maxLength
string list
```

Bad:

```
variable i = 0, numEntries = ItemsInList(list), maxLength
string list = ""
```

- Don't use the default value for an optional argument

Good:

```
StringFromList(0, list)
```

Bad:

```
StringFromList(0, list, ";")
```

- Use parentheses sparingly

Good:

```
variable a = b * (1 + 2)
```

```
if(a < b || a < c)
    // code
endif
```

Bad:

```
variable a = (b * (1 + 2))
```

```
if((a < b) || (a < c))  
    // code  
endif
```

- Use parentheses when combining operators with the same precedence

Good:

```
if((A || B) && C)  
    // code  
endif
```

```
if(A == (B >= C))  
    // code  
endif
```

Bad:

```
if(A || B && C) // same as above as these are left to right  
    // code  
endif
```

```
if(A == B >= C) // same as above as these are right to left  
    // code  
endif
```

The reason is that remembering the exact associativity is too error-prone. See also [DisplayHelpTopic "Operators"](#).

3.2 Waves

- In multidimensional wave assignments always specify the exact dimension for each value:

```
Make/N=(1,1,2) data = NaN  
data[0][0][] = 0
```

In this example data will be set to 0 for both values. Each dimension is specified: p and q are fixed to 0 and both values in dimension r are set to 0.

```
Make/N=(1,1,2) data = NaN  
data[0][0] = 0
```

In this example the output will be 0 and NaN when using Igor Pro 7 (IP7). In Igor Pro 6 (IP6) the assignment will result in 0 for both values.

The IP6 behaviour can be triggered in IP7 by setting an Igor Option:

```
SetIgorOption FuncOptimize, WaveEqn = 1
```


To avoid confusing code always specify what value should go in which dimension (row, column, layer, chunk).

3.3 Constants

- Static constants, which are required only in one file, should be defined at the top of the file
- Global constants are named with all caps and underlines and are collated in a single file
- Explain magic numbers in a comment

```
static Constant DEFAULT_WAVE_SIZE = 128 // equals 2^8 which is
                                         // the width of the DAC signal
```

3.4 Macros

- Use Macros only for window recreation macros
- Try to avoid changing window recreation macros by hand. Write instead a function to reset the panel to the default state and let Igor Pro rewrite the macro by DoWindow/R.

3.5 Functions

- Try to keep their length below 50 lines (or half the screen height)
- Use CamelCase for function names (optionally prefixed by SomeString_ denoting the filename)
- Make them `static` if they are only required inside the same procedure file
- Define all variables at the top of the function body as in

```
Function CalculatePressure(weight, size)
    variable weight, size

    variable i, numEntries

    // code
End
```

The reason for this rule is that there is no block-scope in Igor Pro, i.e.

```
if(someCondition)
    variable a = 4711
end

print a
```

is valid code. And that certainly will confuse people coming from C/C++. Please also note that in the example above a blank line separates function argument definitions from general variable definitions. This will improve readability.

- Optional arguments should have defined default values:

```
Function DoCalculation(input, [verbose])
    variable input, verbose

    if(ParamIsDefault(verbose))
        verbose = 0
    endif

    // code
End
```

- Function Call with optional arguments:

```
DoCalculation(41, _verbose_=1)
```

When calling a function, each argument is separated by a comma followed by a whitespace. Optional arguments are set with surrounding white spaces before and after the equal sign.

- Boolean optional arguments should be forced to (0,1)

```
Function DoCalculation([overwrite])
    variable overwrite

    overwrite = ParamIsDefault(overwrite) ? 0 : !!overwrite

    if(overwrite)
        // Some Code
    endif

    if(!overwrite)
        // Negation will work as expected
    endif
End
```

The reason for this rule is that possibly unexpected behaviour should always be avoided. Without the double negation statement neither one of the above if statements would get triggered if `overwrite=NaN`.

To make this clear look at the following example: The function will print 2 as NaN can not get evaluated.

```
Function NaNisNotBool()
    if(NaN)
        print 0
    end
```

```
elseif(!NaN)
    print 1
else
    print 2
endif
End
```

4 Links and Literature

- ASCII: <https://en.wikipedia.org/wiki/ASCII>
- Doxygen: <http://www.stack.nl/~dimitri/doxygen/index.html>
- Git settings for Igor Pro code: <http://www.igorexchange.com/node/6013>
- Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall (2008)
- How to write good commit messages: <http://who-t.blogspot.de/2009/12/on-commit-messages.html>