Standards for User Defined File Loaders
A Working Proposal

Version 1.04(α)

J. J. Weimer

August 15, 2007

Distribution Details

Provides Packages: udFLStandardFunctions and udFLStandardStructure

Provides Experiment: none

Provides Procedure Files: SimpleStandardLoaderPanelDemo

Requires Packages: none

Developed with Igor Pro Version: 6.0.2

Abstract

This is a proposal to define standards for how user defined file loaders are designed to interface with other user defined packages in Igor Pro.

The proposed standards based on defining proper naming conventions and on using a STRUCTURE to carry information (processing parameters) between the file loader and the routines that interact with it.

Conventions and standards are proposed, the file loader structure is defined, and a basic application is presented in this document. A simple panel is also provided with the package distribution for users to test development of “standardized” file loaders.

This document was generated using LaTeX

Contents

1 Summary **1**

2 Background **1**
2.1 Loading Data Using Igor Pro Menus . . . 1
2.2 Loading Data Using Igor Pro Commands . . . 2
2.3 Loading Data Using User Defined File Loaders . . . 2
2.4 Loading Data Using Standard User Defined File Loaders . . . 2

3 Setup **3**
3.1 Requirements . . . 3
3.2 Package Contents . . . 3
3.3 Installation . . . 4

4 A Standardized File Loader **4**
4.1 Format . . . 4
4.1.1 Function Call, Name, and Parameters . . . 5
4.1.2 Structure Name . . . 6
4.1.3 Structure Contents . . . 7
4.2 General Use . . . 15
4.3 Programming Applications . . . 15
4.3.1 Using Templates for Basic Development . . . 15
4.3.2 Reading Input Parameters to Find Files . . . 18
4.3.3 Using Input Parameters to Refine Loader Processing . . . 21
4.3.4 Using Return Parameters to Assure Standard Behavior . . . 26

5 Some Standardized File Loader Processing Functions **29**
5.1 Zeroing the Standard Structure . . . 29
5.2 Assuring the File Paths . . . 30
5.3 Moving To and Creating New Data Folders . . . 31
5.4 Handling Error Codes and Messages . . . 31
5.5 Query Processing Modes . . . 31

6 A Simple Standardized File Loader Panel **31**

7 Acknowledgments **32**

8 Contact **32**

9 Legalize **32**

1 Summary

File loaders are an integral part of Igor Pro. They provide a way to input data from files into Igor Pro. A wide variety of data file formats exist outside of Igor Pro. No one file loader exists in Igor Pro to handle them all. Igor Pro provides a File Loader panel to input a number of common formats. Loaders for file formats outside of the common ones are generally developed by users of Igor Pro specific to their needs.

Two potential shortfalls exist in the current framework of development for user-defined file loaders. First, new users of Igor Pro are not always aware of the existence of file loaders other than those built-in to Igor Pro. Secondly, even when the existence of a specific user-defined file loader is well published and its use well-documented, it is at best "different" in its interaction than any other file loader and at worst "idiosyncratic" in how it is to be used. These problems can be a frustration for new users and seasoned programmers alike.

This document generates a proposed STANDARD for how user defined file loaders are to interact with other routines. It also outlines an example file loader and provides a simple file loader panel to test development of "standard" file loaders.

This document deals only with how files can be loaded in a standard manner. How the data are processed once they are loaded is left to the user to determine. In this regard, standardized file loaders should not presume to do any display or analysis of the data after they are loaded. Their sole task is to bring some type of file into Igor Pro in a consistent (standardized) manner.

In other words, when creating a "standardized" file loader, preparation of standardized routines for data processing should be left as an exercise for the user :-).

2 Background

Files can be loaded into Igor Pro in on of three ways.

2.1 Loading Data Using Igor Pro Menus

The top level method to load files is using the Load Waves menu option and associated sub-menus. These generally provide a dialog box or interface panel. Options are available to select the format of the files.

2.2 Loading Data Using Igor Pro Commands

A number of operations are provided in the Igor Pro command language to read files. The most common are outlined below. In writing a user-defined file loader within Igor Pro, you will likely use one or more of these commands as part of your file loader.

Open

This command can open a file, save a file, or just verify that a file is available to be opened. A related command is `OpenNotebook`. Any file that is opened should be closed using the `Close` command.

FBinRead

This command is used to read data from a file in a byte-wise manner. Commands that are associated with controlling the reading are `FSetPos` and `FStatus`.

FReadLine

This command is used to read binary data from a file. Commands that are associated with controlling the reading are `FSetPos` and `FStatus`.

LoadData

This command is used to read data from a different Igor Pro experiment into the current Igor Pro experiment or from a file-system folder containing Igor Pro type data.

2.3 Loading Data Using User Defined File Loaders

A wide range of user-designed file loaders have been created by the community of users of Igor Pro. Most are available at no additional cost. One distinct problem is, a central repository of file loaders for Igor Pro does not (yet) exist. Another problem is, because no standard exists for how a file loader should be designed and should operate, each file loader can be expected to be different from all others. Some file loaders contain well-designed panels or dialogs to guide the user. Others can require some external programming.

2.4 Loading Data Using Standard User Defined File Loaders

The purpose of this document is to put forward a standard paradigm for how user defined file loaders should work. It advances a sequence of proposals and provides a basic example of how such a standard can be used to create a single interface for a user to access all “standardized” file loaders.

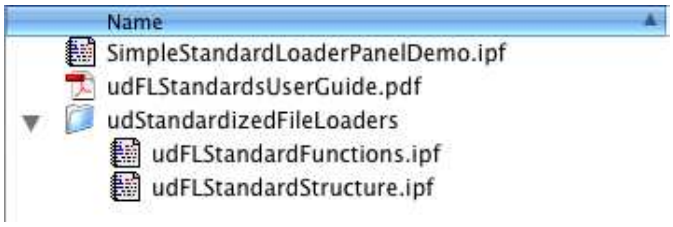
3 Setup

3.1 Requirements

The procedures with this distribution have only been tested on Igor Pro 6.0 and are defined with a minimum requirement of Igor Pro 6.0. Anyone who has success in using the procedures in this distribution under earlier versions of Igor Pro, please report such.

3.2 Package Contents

The procedure is provided in a ZIP archive. Unpacking the archive will reveal the primary (root) folder udFileLoadersStandardsN, where N is the version number. The directory structure inside this folder is shown below.



---

udFLStandardsWorkingProposal.pdf
This document.

SimpleStandardLoaderPanelDemo.ipf
This procedure file generates a panel that can be used to test development of a file loader against the proposed standards.

udStandardizedFileLoaders
This folder contains procedure files needed to implement the standard for file loader development and use. Given that this document is a proposal only, the naming convention is only presented to distinguish the two procedures as a distinct package (or module). The two required files inside this folder are

udFLStandardFunctions.ipf
This procedure file contains functions that may be used as part of a "standard" file loader. It is basically provided as a benefit for developers as a way to help avoid repeating the coding of what might be considered to be core functions for a file loader, especially in how it interacts with the standard loader structure defined in this document.

udFLStandardStructure.ipf
This procedure file contains the proposed standard structure.

3.3 Installation

The directory `udStandardizedFileLoaders` **MUST** be installed in ONLY one way. Move or copy the entire directory into the `User Procedures` directory of your local installation of Igor Pro. You must do this installation before you open any experiment that is to use the demo of the Simple Standard Loader Panel.

You can rename the directory after it is installed in the `User Procedures` directory (although this is not recommended!). You can also install future "standardized" file loaders into this directory as a way of keeping track of them (this is recommended).

The `SimpleStandardLoaderPanelDemo.ipf` procedure file can be installed in one of two ways.

- To have the simple loader panel available every time you start Igor Pro, move or copy the procedure file to the `Igor Procedures` directory of your local installation of Igor Pro.
- To have the simple loader panel available only when you wish to include it, first move or copy the file to the `User Procedures` directory of your local installation of Igor Pro. Then, when you wish to use the panel in a specific experiment, open the Macros window while in Igor Pro and put the line

  ```
  #include "simplestandardloaderpaneldemo"
  ```

  somewhere directly after the `#pragma rtGlobals=1` directive that usually appears in this window.

4 A Standardized File Loader

4.1 Format

The basic workings of the "standard" structure for programming purposes is illustrated by the simple coding for a "standardized" file loader that is shown below.

`Static Function udFLIinitStructure(udFL)`
initialize the structure

`Function udFileLoaderMYLOADER(udFL)`
do whatever is required by MYLOADER to input the file and store the data

The devil is in the details, though. To have proper operation of "standardized" functions requires that certain conventions be followed when developing them and when programming to use them.

4.1.1 Function Call, Name, and Parameters

The first conventions provide for consistent access to the file loader function.

> **Proposal 1**
> The file loader shall be accessed via a call of the form `Function XXX(YYY)`, where `XXX` is a name that is to be defined (by Proposal 2) and `YYY` are the parameters to be defined (by Proposal 3).

The second convention provides a consistent way to get the name of all standardized file loaders in use at any time.

> **Proposal 2**
> The name of the file loader function `XXX` shall be prefixed by `udFileLoader` followed by the name of the file loader (with or without capital letters).

When a user defined file loader conforms to the above convention, a call to the Igor Pro string function FunctionList("udFileLoader⋆",";","KIND:6") will return a separated list of all standardized file loader functions currently accessible in a given experiment. The prefix "udFileLoader" that appears on each loader name can be stripped via further processing as needed.

Note that, just because a file loader appears in a list of accessible files when searched using the method shown above does not mean it will work properly. The third convention therefore provides a consistent way to interact with routines that make function calls to the loader.

Proposal 3

The function itself shall take one parameter, a pointer to the standard file loader structure, as

```
Function udFileLoaderMYLOADER(sP)
STRUCT SSS sP
```

The use of `sP` to define the abbreviation for (pointer to) the structure is optional. The name `SSS` of the `STRUCT` is to be defined (by Proposal 4)

4.1.2 Structure Name

In order to be able to use the file loader structure for any file loader, it must be accessed by the same name.

Proposal 4

The name `SSS` of the structure shall be `udFLStandardStructure` as an notation for User-Defined File Loader Standard Structure. A suggested abbreviation for the pointer `sP` to the structure is `udFL` (as used throughout this document).

Examples

Accepting the above four proposals as conventions, the following examples show "standardized" file loaders:

```
Function udFileLoaderTabDelimitedXY(abCD)
STRUCT udFLStandardStructure &abCD

Function udFileLoaderXYZa(udFL)
STRUCT udFLStandardStructure &udFL
```

By comparison, the following examples show non-standardized file loaders:

```
Function udFileLoaderTabDelimitedXY(abCD,X)
STRUCT udFLStandardStructure &abCD
variable X

Function MyLoader(udFL)
STRUCT udFLStandardStructure &udFL

Static Function udFileLoaderMyLoader(udFL)
STRUCT udFLStandardStructure &udFL
```

4.1.3 Structure Contents

The remaining conventions deal with the internal operation of a standardized file loader. In discussing them below, the assumption is that, as a programmer, you are already familiar with how to use STRUCTURES in Igor Pro. The best example case is when you have written control procedures for panel controls and have worked through the examples to understand when structure parameters are “resident” in a function and when they disappear. If you are not to this point, then you may not yet be ready to write a ”standardized” file loader. You may however find encouragement from the somewhat tutorial approach the following discussion tries to take.

First, the standard structure `udFileLoaderStandardStructure` is a place to store parameters needed by the file loader to interact consistently with its “surroundings” (the calling functions). The structure is not a place where data from the data files are stored (they are stored directly into Igor Pro folders for example).

The contents of the structure should be established to provide consistent functionality and optimal utility. This interest establishes the thrust of the next proposal and its sub-proposals.

Proposal 5

The `udFileLoaderStandardStructure` shall contain parameters covering four types of information, as presented below.

Descriptors

The parameters in this section describe the "character" of the file loader.

Inputs

The parameters in this section provide a way for the programmer to input commands to control what the file loader is to do.

Localizers

The parameters in this section define where the file loader is to operate when it does its job.

Returns

The parameters in this section define the success or failure of the file loader when it completes its job.

Each type of information in the standard file structure is to be covered by defined parameters, as presented on the next few pages. In those cases where the parameter definition is UNDEFINED, further investigation is needed to establish a proper standard or standards for the parameter.

Proposal 5A

The following five parameters shall be specifically contained within the Descriptors section of the file loader:

```
string mimetype
```

This string defines the *mimetype* of files that can be loaded by the loader. An example is “TEXT” or “IGOR”.

```
string extensions
```

This string (list) defines the file extensions that can be loaded by the loader. It is a semi-colon separated list of all file extensions. An example is “.txt;.dat;”.

```
int32 itemsType
```

This numeric parameter defines the type of data that are stored by the loader. It is a bit-wise comparative number following the conventions:

BIT
0: global
1: keyword-list (using ”=” and ”;”)
2: wave
3: matrix
4: image
5: notebook
6: procedure file
others: user defined

When a loader stores more than one type of data simultaneouly, the resultant value for `itemsType` shall be a BIT-wise addition of values. For example, a loader that stores globals and images simultaneously would have an `itemType` of 1 + 16 = 17, while a loader that stored waves and a notebook would have an `itemType` of 4 + 32 = 36. When a loader stores more than one type of data, and the type it stores depends on the processing mode, the `itemType` should be set depending on on the processing mode and user control, as discussed later in this document.

Proposal 5A (continued)

```
int32 itemsDim[4]
```

This numeric parameters defines the "dimensionality" of the loaded data. It is a positive or negative integer number in an array storage that depends on `itemsType` following the conventions:

itemsType = 1 (global)
- itemsDim[0] = 1: variable
- itemsDim[0] = 2: string

itemsType = 2 (keyword-list)
- itemsDim[0] = 1: list uses "="
- itemsDim[0] = 2: list uses ":"

itemsType = 4 (wave)
- itemsDim[0] = -1: unscaled numeric wave
- itemsDim[0] = 1: scaled numeric wave
- itemsDim[0] = -2: unscaled text wave
- itemsDim[0] = 2: scaled text wave

itemsType = 8 (matrix)
- using N to indicate the matrix dimension
- $N = 0$: rows; $N = 1$: columns; $N = 2$: layers; $N = 3$: chunks
- itemsDim[N] = -1: unscaled numeric
- itemsDim[N] = 1: scaled numeric
- itemsDim[N] = -2: unscaled text
- itemsDim[N] = 2: scaled text

itemsType = 16 (image) – TO BE DEFINED!

itemsType = 32 (notebook)
- itemsDim[0] = -1: unformatted
- itemsDim[0] = 1: formatted

itemsType = 64 (procedure) – TO BE DEFINED!

Proposal 5A (continued)

```
string procModes
```

This string list defines the options available to a user to control the file loader. The first mode shall be “Auto”. All other modes are established by the programmer. An example for `procModes` for a file loader with four loading modes is “Auto;Via Dialog Box;Without Preprocess;Manual;".

Proposal 5B

The following five parameters shall be specifically contained within the Inputs section of the file loader:

```
int32 eventCode
```

This numeric parameter defines the “action” that is evoking the file loader. A value of `eventCode` = -1 shall mean, a request has been made to kill the file loader package (remove it from the current Igor Pro experiment). All other eventCodes are for the programmer to define.

```
int32 userCtrl
```

This numeric parameter is the process control variable for the file loader. All positive values of `userCtrl` map directly onto the index value of the mode in the `procModes` list, starting from a count of 1. Two special `userCtrl` values shall also apply.

Value
-1: initialize the loader, and query it for its Descriptors
0: initialize the loader only

A file loader with `procModes` = “Auto;Via Dialog Box;Without Preprocess;Manual;" would use mode “Without Preprocess” when `userCtrl` = 3.

Proposal 5B (continued)

```
string userData
```

This string parameter is an option to further refine a processing mode in the file loader.

```
int32 reportCtrl
```

This numeric parameter provides a way to define how the loader reports on its activities. Three specific modes shall be reserved.

Value
-1: silent
0: normal
1: verbose

The programmer is free to define the level of reporting provided by the file loader in each case.

```
string plugIns
```

This string list is an option to allow a programmer to enhance or supplement the loader through "plug-in" functions. A plug-in function is one that accepts the `udFileLoaderStandardStructure` and uses its parameters to speed up or enhance manipulations within the loader. A prime example is to make a file loader use a certain XOP routine that does something that otherwise is hard-coded into the loader. Providing a loader that is aware of the XOP is equivalent to making it "aware" of the XOP by adding the function name to a list in `plugIns`.

Proposal 5C

The following four parameters shall be specifically contained within the Localizers section of the file loader:

```
string pathStr
```

This establishes the path to the directory containing the file(s) to be loaded. The string uses the Igor Pro convention of path names being separated by ":" (colons).

```
string fileList
```

This establishes the list of files to be loaded. The list can contain the full path to the file(s), a partial path to each file(s), or the file name(s) without extension(s), as discussed later in this document.

```
int32 returnCtrl
```

This numeric value controls additional aspects of how the loader reports its progress. In particular, this value controls whether Igor Pro global parameters such as S_value and V_flag, that are normally created during Igor Pro `Open` processes, are also to be created by the file loader. Two specific values shall be reserved.

Value
0: normal (all return information is in the file loader structure)
1: enhanced (also create additional “global” parameters)

```
string toFolder
```

This string defines the Igor Pro folder where the data are to be stored. It shall always be defined relative to the `root` level.

Proposal 5D

The following four parameters shall be specifically contained within the Returns section of the file loader:

```
int32 itemsCount
```

This numeric value provides the total number of items loaded.

```
string itemsList
```

This string list provides the names of the items (globals, waves, ...) that have been loaded.

```
int32 errCode
```

This numeric value defines whether any error occurred during the execution of the file loader. Two return values shall be reserved.

Value
0: any error is only a warning
-1: any error is considered fatal (and should likely lead to an abort)

```
string errMsg
```

This string (list) returns any error messages generated by the file loader during processing. It shall accumulate warnings sequentially. Therefore, warnings can be dumped to a report. By comparison, any fatal error messages shall overwrite any prior contents, with the understanding that they are designed to be handled by an immediate `DoAlert` followed by an `Abort`.

In addition to the above parameters within the structure, the file loader itself shall return a 0 when it has executed properly and a -1 when some error (warning or fatal) has occurred.

4.2 General Use

As a general user of Igor Pro, you will typically have no direct interaction with the Structure procedure file that is part of this package distribution. Your only need is to maintain an updated copy of the file (and its companion Functions file) in your Igor Pro distribution. All procedures that conform to the standards proposed in this document will then be accessible to you, ideally through "plug-and-load" interfaces, such as the one provided by the Simple File Loader Panel Demo.

4.3 Programming Applications

4.3.1 Using Templates for Basic Development

A basic application of the file loader standard structure is illustrated in the two templates below. These templates are written in Igor Pro, they are not written as C/C++ for XOPs. Having someone provide such templates based would be welcomed as an addition to this document.

Initialization

The first function is the initialization routine for the loader.

```
Static Function udFLInitStructure(udFL)
  STRUCT udFLStandardStructure &udFL

  udFLZeroStructure(udFL) // zero the structure
  udFL.mimetype = "..."  // set the mimetype
  udFL.extensions = "..."  // set the extensions
  udFL.itemsType = ...  // set the itemType
  udFL.itemsDim[0] = ...  // set the itemsDim
  udFL.proModes = "Auto;..." // set the procModes
  if (udFL.userCtrl == -1)
    print udFL // show descriptors in history
  endif
  return 0
end
```

The function call to `udFLZeroStructure(udFL)` assures that all parameters in the file loader standard structure are properly defined (as explained later in this document).

The remaining lines define the character of your file loader. This information is passed back to the primary calling routine via your loader function.

Note that your initialization routine is defined as a STATIC function so that it is local to your procedure and only accessed by going through your file loader function.

File Loader

The second function is the entry point to your loader function. A basic template for a "standardized" file loader is shown below.

```
Function udFileLoaderMyLoader(udFL)
  STRUCT udFLStandardStructure &udFL

  if (udFL.eventCode==-1)
    ... procedure file is being killed
    ... do whatever is needed to clean up
    return 0
  endif

  switch(udFL.userCtrl)
    case -1:
    case 0:
      udFLInitStructure(udFL)
      break
    case 1:
      ... auto-load mode
      ... assume all parameters are defined in structure
      ... do auto-processing
      break
    case N:
      ... mode N
      ... do whatever processing
      break
    ...
  endswitch

  return 0
end
```

Each part of the template is presented in turn below.

Clean-up

The first portion of the above template provides a standard way to make your file loader aware of an event that is going to kill the procedure window containing your file loader. You can thereby write specific "clean-up" procedures within the `if ... endif` portion of the loader. The `return 0` command assures that no further processing is done by the loader after all the clean-up steps have completed.

In providing a standard `udFL.eventCode = -1` as a method to notify the file loader, the assumption is, any routine that attempts to kill your file loader procedure should pass `udFL.eventCode = -1` to it first. This may not always occur. One way to assure this may be through the use of Hook functions within your procedure itself. Further clarification of this process remains to be established.

Initialize

The initialization of the file loader occurs if `udFL.userCtrl = -1` or 0 when the file loader is invoked. In the first case, the initialization is followed by a report of the Descriptors of the file loader. In the example initialization routine, this is done by printing the entire structure to the history window. Further possible refinements to this are discussed later in this document. When `udFL.userCtrl = 0`, the file loader only initializes itself, nominally to its base set of Descriptors.

Auto-Process

The auto-process mode of the file loader occurs if `udFL.userCtrl = 1` when the file loader is invoked. A primary assumption is, all parameters in the Inputs and Localizers sections of `udFL` are pre-defined (by the user in some way) when auto-process mode is invoked. Your file loader should therefore not need to use dialog boxes or prompts for user input in the auto-load mode! It may (and indeed should) test that all parameters are indeed properly set and return an appropriate `udFL.errCode` and `udFl.errMsg` if they are not (as discussed later in this document).

Other Process Modes

Other processing modes are established by `udFL.userCtrl` values greater than 1. The value of `udFL.userCtrl` is to map exactly with the index value of the "mode" string in `udFL.procModes`. As an example, the table below shows how processing modes and user control values map for a file loader with three loading modes. The first menu selectable mode is "Auto" by definition. The second mode, "Pre-Select", is run when `udFL.userCtrl = 2`. The third mode, "Manual", is run when `udFL.userCtrl = 3`.

User Control	Mode	Notes
-1	Initialize and Query	typically not shown in a menu selection
0	Initialize (only)	typically not shown in a menu selection
1	Auto	first menu selection for all file loaders
2	Pre-Select	a specific mode for this file loader
3	Manual	a specific mode for this file loader

For the above table, to create the proper sequence for relating `udFL.procModes` and `udFL.userCtrl`, the value of `udFL.procModes` would be initialized as "Auto;Pre-Select;Manual;" (a string list).

4.3.2 Reading Input Parameters to Find Files

The use of input parameters `udFL.eventCode` and `udFL.userCtrl` have been illustrated in the example above. Two remaining parameters are essential to the "Auto" mode of processing. They are `udFL.pathStr` and `udFL.fileList` in the Localizers section of the file loader standard structure. These two parameters define where the files are located (and are therefore considered as localizers, although they also are required input parameters in "Auto" mode).

Your file loader can use the two parameters in any way you want. Some general ways are proposed below for further consideration.

Loading a Single File

When the "Auto" mode of loading is designed to load only a single file, a recommended convention is that `udFL.pathStr` will contain the full path to the file and `udFL.fileList` will contain the filename (with extension). In this case, your file loader could create a temporary data Path, load the file, and then delete the temporary data path. An example code to do this is shown below. In this code, the function `udFLAssureFilePaths(udFL)` is a function provided in the standard set, as discussed later in this document.

```
case 1:
  variable ic, nc = udFLAssureFilePaths(udFL), refNum
  string fname
  switch(nc)
    case -1:
      ... handle as error
      break
    case 1:
      fname = StringFromList(0,udFL.fileList)
      NewPath/O/Q tmpFPath $udFL.pathStr
      Open/R/Z=1/P=tmpFPath refNum as fname
      if (V_flag!=0)
        Close refNum
        KillPath/Z tmpPath
        ... process the error that occurred
      endif
      Close refNum
      ... process the data
      KillPath/Z tmpPath
      break
    default: // multiple files (see below)
      ...
  endswitch
  break
```

Loading Multiple Files

The "Auto" mode of loading is actually best designed when it can load many files of the same format, potentially from multiple directories. When all files are in the same directory, `udFL.pathStr` should contain the path and `udFL.fileList` just the file names. When the files are in multiple directories, `udFL.pathStr` is to be an empty string ("") and `udFL.fileList` is to contain the list of files with their full path names, including extensions. This convention is assured by the function `udFLAssureFilePaths(udFL)` (as discussed later in this document). An example code to handle one of the cases is shown below.

```
case 1:
   ...
   switch(nc)
      ... handle error and single file cases
      default:
        if (strlen(udFL.pathStr)!=0)
             ... all files are in the same directory
             ... loop on fname while in same path
        else
          do
            fname = StringFromList(ic,udFL.fileList)
            if (strlen(fname) == 0)
              break
            endif
            Open/R/Z=1 refNum as fname
            if (V_flag!=0)
              Close refNum
              ... process the error that occurred
            endif
            ... process the data
          while(1)
        endif
      break
   endswitch
   break
```

Handling Empty File Lists

You may wish to allow a user the option to select file(s) for loading by coding file selection routines *within your loader*. Of course, this MUST not happen in the "Auto" processing portion of your loader. You may decide that, an empty `udFL.fileList` is to be taken as a directive by the user to open a file selection dialog. Again, this MAY NOT happen when `udFL.userCtrl` = 1. In this case, an empty value of `udFL.fileList` is to be handled as an error from the user input (this will be assured by a call to `udFLAssureFilePaths(udFL)`.

In those cases where you do allow the user to select files by specific selection dialogs or prompts within your file loader, a good practice is to put the file selections properly

into udFL.pathStr and udFL.fileList following the conventions proposed above for loading a single file or loading multiple files. When your loader also processes the incoming data, this can in fact make your code easier to oversee, as the example below illustrates.

```
switch(udFL.userCtrl)
  case 1:
    break
  case 2:
    ... allow user to select files list
    ... store file names in udFL.fileList
    break
endswitch
... load file(s) in udFL.fileList
```

4.3.3 Using Input Parameters to Refine Loader Processing

The parameters in the Inputs section of udFL are to by defined by routines outside of your file loader (how this can be done in a "standard" way is part of the discussion in the section on the standardized file loader panel later in this document). You should NOT change them inside your file loader. You should use them as "switches" to define what your file loader does.

The use of udFL.eventCode and udFL.userCtrl have been illustrated in the preceding section. The examples below show how other input parameters can be used to further refine what your file loader does.

Defining Sub-modes

The string udFL.userData provides one way of having sub-modes of processing for any "main" processing mode. This is best illustrated by a programming example. In the section on the next page, only that part of the file loader that handles a mode when the value udFL.userCtrl = 2 is shown.

```
case 2: // my loader does "pre-processing" in this mode
  if (strlen(udFL.userData)==0)
    ... no special sub-mode pre-processing is requested
    ... do all pre-processing
    break
  endif
  strswitch(udFL.userData)
      case "preformat":
        ... only preformat mode is requested
        break
    case "overlay":
        ... only overlay mode is requested
        break
    default:
        ... the sub-mode string is not recognized
        ... handle this as an error
        return -1
  endswitch
  break
```

The sub-mode strings required from the user for your file loader can be as simple or as complex as you want. Verbose types of strings, as in the example above, are one example. Requiring strings that are more UNIX-like, for example “-p” for “preformat” and “-o” for overlay, is another example. You may even require numerical values (as strings) using `udFL.userData` and use spaces as separators to enable “multiple” or “sequential” sub-mode processing. Perhaps the only rule about using the `udFL.userData` string parameter is, the string required in `udFL.userData` at any point in your file loader should not be required to contain special characters such as $\star$ or ?.

Note also, you should be certain to handle cases when the input `udFL.userData` does not equal any of your desired selections. The example above shows, this is handled by the `default:` case of the string switch selection and is processed as a user input error.

Finally, the primary assumption when `udFL.userCtrl` = 1 is, your file loader will operate in “Auto” process mode. In particular, this means, no special sub-modes are to be selectable. Therefore, when `udFL.userCtrl` = 1, any string value that is contained in `udFL.userData` should be ignored within that portion of your file loader!

Localizing Processing

The objective of many file loaders is to store data in Igor Pro so that it can be further processed. This is certainly true for global, keyword-list, wave, matrix, and image data. In such cases, the strong recommendation within the Igor Pro community is to establish and use Igor Pro data folders as a place to store the data.

The discussion in this portion of the document assumes that you, as a programming, are familar with creating, setting, and changing data folder locations by appropriate Igor Pro coding within your file loader.

The file loader structure offers the parameter `udFL.toFolder` that is set by the user as a way to direct any file loader to a specific location where it MUST store the data. This directive is an IMPERATIVE, not an optional directive. In other words, when the parameter `udFL.toFolder` is not empty, your file loader is being commanded to change to the folder specified by the string and to load data only into that specific folder.

The value in `udFL.toFolder` is always a full folder path relative to the root folder. An empty value means, data are to be stored at the root folder level.

A standard function `udFLSetDataFolder(udFL,[setit])` has been provided to allow your loader to create and change to a specific folder based on the value pased in `udFL.toFolder`. Its use is illustrated in the example on the next page.

Your file loader MUST also return to the original data folder before the loader exits its processing. In general, a recommended way to assure that you always return to the original data folder is to include proper coding for `GetDataFolder(1)` and `SetDataFolder`, as is also shown in the example on the next page.

```
Function udFileLoaderMyLoader(udFL)
  STRUCT udFLStandardStructure &udFL

  string cdf = GetDataFolder(1)

  if (udFL.eventCode==-1)
    ...
    SetDataFolder $cdf // reset just in case the data folder was changed
    return 0
  endif

  udFLSetDataFolder(udFL)

  switch(udFL.userCtrl)
    ...
      SetDataFolder $cdf
      return -1
    ...
  endswitch

  SetDataFolder $cdf
  return 0
end
```

Within the confines of the above directives for `udFL.toFolder`, your file loader is otherwise free to create sub-folders in any of its loading "modes", including the "Auto" mode. At the end, your file loader will inform the user of any folders created by returning proper values in the `udFL.itemsList` parameter, as discussed later in this document.

Reporting Progress

The file loader standard structure provides one parameter to define the extent of reporting requested by the user during processing. The parameter `udFL.reportCtrl` can be used to define three (or more) "modes" of reporting information. In Silent mode, the processing proceeds with no reports. In Normal mode, the loader reports on certain stages of processing. Finally, in Verbose mode, the loader reports frequently throughout. The level of reporting in each mode is left entirely to the programmer. An example

of how this can be used to document the progress of the Open command is shown in the code below.

```
switch(udFL.reportCtrl)
  case -1:
    Open/R/Z=1/Q ...
    break
  case 0:
    Open/R/Z=1 ...
    break
  case 1:
    Open/R/Z=1 ...
    print "Just opened the file for reading ..."
    break
endswitch
```

Accessing Plug-Ins

Plug-ins should be understood to be routines that might be of general use to a wide range of file loaders or segments of code that are written in a more compact manner (ie, as an XOP). To be accessible as a plug-in, the plug-in function will be required to conform to conventions in its name (udFLPlugInXXX(YYY)) and parameters (YYY is the file loader standard structure).

After the file loader standard structure is zeroed using udFLZeroStructure(udFL), the parameter udFL.plugIns will contain a string list of all currently accessible plug-ins. This list can be checked for a particular plug-in and the plug-in used when available. The example code below shows how this could be designed.

```
if (strlen(ListMatch(udFL.plugIns,"udFLPlugInInvertData")==0)
  ... process by Igor Pro code
else
  udFLPlugInInvertData(udFL)
endif
```

4.3.4 Using Return Parameters to Assure Standard Behavior

Defining Character

The "character" of the file loader is an indication of what files it is designed to handle and how (in general) the results will appear after it is complete. The parameters in the Descriptors section of the file loader standard structure define this. Whenever an external routine queries your file loader, it should return proper values of the descriptor parameters. The initialization template shown previously in this document is one general way this can be done. Further comments about each descriptor parameter are given below.

No convention applies when the file loader can handle more than one mime type. One proper method of return is to set the mime type to the most common value. Alternatively, the mimetype can be set to an empty value. Finally, the mime type can be set dependent on other user inputs, such as `udFL.userCtrl` or `udFL.userData`, if this is appropriate to how the loader behaves. In no event (as of this version) should the mime type be set to more than one value. This option is reserved for future versions of the structure.

When more than one file extension can be loaded, the convention is to create a string list of all possible extensions. The understanding is, the search for file extensions takes precedence over the search for mimetype when searching for files to load, and, as needed, all files containing all possible file extensions will be collected and displayed for selection by a "standardized" input routine.

When the file loader can return more than `udFL.itemsType` value, and absent any other qualifiers, it should return the BIT-wise addition of all possible values. The example in Proposal 5A illustrates this convention. For such "multi-type" file loaders, when the value of `udFL.itemsType` depends specifically on `udFL.userCtrl`, the value of `udFL.itemsType` must be returned appropriate to the value of `udFL.itemsType`. The example code on the next page shows such behavior in an outline.

```
... within initialization function
udFL.itemsType = 17 // loader returns globals and images

... within the actual loader function
switch(udFL.userCtrl)
  case 1:
    ... load both globals and images
    udFL.itemsType=17
    break
  case 2:
    ... load only images
    udFL.itemsType=16
    break
endswitch
```

When only one data type is stored by the file loader, the conventions for returning proper values in`udFL.itemsType` also apply for the value of `udFL.itemsDim[N]`. When more than one type is stored by the file loader, no convention exists for setting the value of `udFL.itemsDim[N]`. An appropriate return value may be a value of ZERO (0), to indicate a state that requires further study by the user.

With the exception of the convention that "Auto" be the first value, the proper string names to return in `udFL.procModes` are left entirely to the programmer. The only suggestion is, the names should be short (to fit in a popup menu list) and descriptive (to inform the user of their function). By the way, the language used for the string is also entirely up to the programmer. One method of preparing language localizations for this menu is to use `StrConstant` values in a localization file. The example below shows a section to define the values for English or German, with the current localization being for German.

```
... in the loader preamble
#define DEUTSCH // this is a German localization

#ifdefine ENGLISH
StrConstant udFLprocModes = "Auto;Select the Files;Test;"
#endif

#ifdefine DEUTSCH
StrConstant udFLprocModes = "Auto;Files Auswahlen;Prufen;"
#endif

... in the initialization routine
udFL.procModes = udFLprocModes
```

Pre-processing Data

A "standardized" file loader ONLY loads the file as data. It is NOT to be designed to display or analyze the data after it is loaded. This extends into a gray area however, because some file loaders may also provide "pre-processesing" types of routines.

A valid pre-processing routine is understood to be a routine that transforms one of the data types into a different data type. A function that takes a (previously loaded) notebook and translates it into a keyword-list of values is a valid pre-processing routine. A function that transforms a matrix to an image is also valid. A function that displays an image for the user is not a valid part of a file loader.

Whenever a pre-processing (or transformation) routine is applied to data by the file loader, the value of `udFL.itemsType` must reflect the value of the data AFTER the transformation has been applied. This also applies to the value of `udFL.itemsDim[N]`.

Returning Item Details

The two parameters, `udFL.itemsCount` and `udFL.itemsList`, provide additional documentation of what the file loader has stored.

The value of `udFL.itemsCount` is the total count of all items stored by the file loader. The value of `udFL.itemsList` is a string list of the items loaded using full path name designations starting from the `root:` data folder.

In one sense, `udFL.itemsCount` is redundant information because it can be directly obtained by counting the items in `udFL.itemsList`. In another, it provides a direct value for a user to quickly confirm everything has been loaded properly, for example when the items are the same type but may be loaded across different sub-folders.

Handling Errors

The values of `udFL.errCode` and `udFL.errMsg` should be used to inform the user of problems during the loading. A standard function has been provided to "set" the values, as the example below shows.

```
case 1:
  if (udFLAssurePaths(udFL)<0)
    udFLSetError(udFL,"File names are not proper",-1)
    SetDataFolder $cdf
    return -1
  endif
  ... continue processing in Auto mode
```

Errors that cause aborts should cause a return from the loader with a value of -1. Otherwise, successful loading should return a value of 0.

5 Some Standardized File Loader Processing Functions

A collection of functions is provided as part of a standard package. They have been referenced in the examples above. Their specific use is described in more detail below.

5.1 Zeroing the Standard Structure

The command below will fill the file loader standard structure with default (blank or otherwise) values, as shown. In particular, the value of `udFL.plugIns` will contain the list of available plug-in modules that follow the proper naming convention. This does not test the validity of the plug-in!

```
Function udFLZeroStructure(udFL)
  STRUCT udFileLoaderStandardStructure &udFL

  udFL.mimetype = ""
  udFL.extensions = ""
  udFL.itemsType = 0
  udFL.itemsDim[0] = 0
  udFL.itemsDim[1] = 0
  udFL.itemsDim[2] = 0
  udFL.itemsDim[3] = 0
  udFL.procModes = "Auto;"
  udFL.eventCode = 0
  udFL.userCtrl = 0
  udFL.userData = ""
  udFL.reportCtrl = 0
  udFL.plugIns = FunctionList("udFLPlugIn*",";","KIND:6")
  udFL.pathStr = ""
  udFL.fileList = ""
  udFL.returnCtrl = 0
  udFL.toFolder = "root:"
  udFL.itemsCount = 0
  udFL.itemsList = ""
  udFL.errCode = 0
  udFL.errMsg = ""
end
```

5.2 Assuring the File Paths

```
Function udFLAssureFilePaths(udFL)
```

The command above will assure that `udFL.pathStr` and `udFL.fileList` have proper values. It returns a -1 when they are incorrectly formated. Otherwise, it returns the number of files to be loaded. It returns a blank string in `udFL.pathStr` when multiple files are to be loaded from multiple directories.

5.3 Moving To and Creating New Data Folders

```
Function udFLSetDataFolder(udFL,[setit])
```

The command above will move to a specific folder defined in the `udFL.toFolder` directive. If the folder does not exist, the function creates it. The optional variable parameter `setit` will determine whether to stay in the specified folder or return to the folder of origin. When `setit` = 0, the function returns to the folder of origin. This is equivalent to creating a new data folder but not moving into it. When `setit` = 1, the function also moves to the new or existing folder. The default is `setit` = 1, so a call to `udFLSetDataFolder(udFL)` will create (as needed) the new data folder and move to it directly while a call to `udFLSetDataFolder(udFL,0)` will create (as needed) the new data folder but will not change folder locations.

5.4 Handling Error Codes and Messages

```
Function udFLSetError(udFL,errStr,errCode)
```

The command above will set the `udFL.errMsg` and `udFL.errCode` parameters to the string `errStr` and the numeric parameter `errCode`, respectively. It always returns ZERO.

5.5 Query Processing Modes

```
Function/S udFLQueryprocModes(LoaderFunction)
```

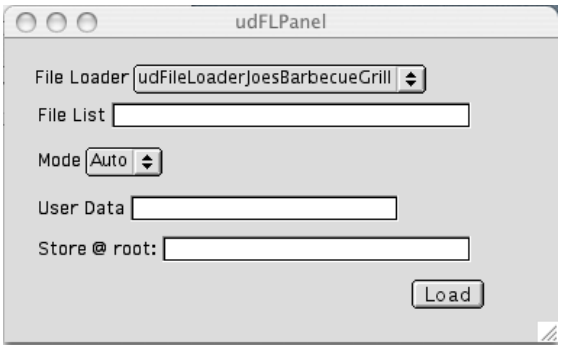
The command above will return a string list of the available processing modes for the loader function named `LoaderFunction` (as a string).

6 A Simple Standardized File Loader Panel

The Igor Pro procedure file to handle “standardized” file loaders is provided as a benefit to users and programmers alike. It is a very basic design and should be simple enough

to understand that no detailed explanation is provided here.

Install the procedure file, open Igor Pro, and select Macros:Compile (unless Auto-Compile mode is on, whereby this is not needed). The Macros menu should have the command for the simple panel demo. This will generate the panel below.



The file loaders that initially appear are included with the demo as a proof-of-concept. Clicking on the Load button only prints out the entire contents of udFL.

To test a “standardized” file loader, first close the existing simple panel. Then install the procedure file for a standardized file loader according to its instructions and recompile in Igor Pro. Now, select the panel demo under the Macros menu, and the installed file loader should appear in the list of file loaders. You can test various inputs (file list, processing mode, user data, and the value for toFolder) by using the panel.

7 Acknowledgments

Thanks Wavemetrics! The ability to use STRUCTURES and FuncRef make this entire process much easier.

8 Contact

Suggestions and comments should be sent to me at jjweimer@matsci.uah.edu. Please use the phrase "Igor Pro Software" somewhere in the subject line so I can sort and reply promptly.

9 Legalize

This software is free to use as per the terms of any other publicly released software. Enjoy!