Introduction to ANSYS Fluent scripting – Part 3 – Creating keyboard shortcuts, menu items and panels

If you have questions regarding this document, please try to contact the author, first:
akram.radwan@ansys.com

Note that the content of this document is not covered by ANSYS support contracts. It is provided as self-help content for your convenience in addition to the contents of the ANSYS Fluent documentation. Always remember that ANSYS can decline to provide technical support if your project relies on Scheme scripts even if your problem description has nothing to do with Scheme.

Description

The Scheme programming language is mentioned in the ANSYS Fluent documentation and also in some self-help knowledge materials available on the ANSYS Customer Portal. But what are Scheme scripts and what is the difference to journal files? How to access Fluent cell zone or boundary conditions within a script? How to create user-defined menu items, keyboard shortcuts or panels?

These questions are answered in this three-part series.

The first document [1] describes Scheme. In principal you can find the content also in several digital books but not all of the commands and procedures described there can be used in ANSYS Fluent. The goal of this first document is to give you a basic understanding of the Scheme programming language. It is not a complete reference guide but you can write most of your scripts with what you learn here.

The second document [2] describes Fluent specific commands and procedures. You learn how to process text output and some convenience procedures. That document can only scratch the surface but it should give you the tools that you need to develop versatile scripts for ANSYS Fluent. You should be familiar with the contents of the first document before you start with the second part.

With this third document you can learn how to create your own menu items, keyboard shortcuts and panels. Most of this document is also covered in the official documentation [3] but the examples might make it easier to understand the concept. You should be familiar with the contents of the first document before you start with the third part. Knowledge about the second part is not required.

Everything in this document has been tested with Fluent 18.0 on Windows and some parts also on Linux. Still, it is possible that you get unexpected results. Scheme is not documented and Fluent procedures and commands can change the behavior or stop working completely with every new minor release.

Important

Scheme is poorly documented and not covered by ANSYS support contracts. ANSYS can refuse to provide support if you have trouble with simulations that rely on Scheme scripts. ANSYS can also refuse to provide support for journal files that contain Scheme code.

Nevertheless, with Scheme you have a mighty tool at your disposal. You can automate simulation setup, post-processing and case modifications. You can even create your own menu items and panels to have quicker access to panels and settings you need most often.

Note: If you are interested in streamlining your processes in ANSYS Fluent you should also consider learning ACT (ANSYS Customization Toolkit). Starting with Fluent 17, ACT wizards can be used as beta features. There are demonstrations available in the ACT library of the ANSYS Customer Portal. ACT is not covered in this document.

Important

This document has not been reviewed to meet the quality standards of ANSYS Best Practice Guides. Please get in touch with the author if you find issues with the document. However, ANSYS does not guarantee that this document is updated in the future.

Introduction

In this third part of Introduction to ANSYS Fluent scripting you learn about menu items, keyboard shortcuts and panels.

In the first chapter you learn how to record journal files efficiently. A recorded journal will be used as example for keyboard shortcuts and menu items which are covered in chapters two and three.

The largest part of this document is chapter four. It covers the creation of your own panels. This can be complex and confusing. Therefore, you find a description about each element on its own before everything is put together with an example configuration. Fluent reads that file automatically and you have access to your customization all the time.

All code examples are available for download in a single zip file.

Recording journal files

In general, it is not recommended to record journal files. If you can you should write them using text commands. However, recorded journal files can be great to do a partial setup of a panel and leave it open for interactive completion. You can even combine them with other Scheme commands.

It is easy to record a journal that is either incomplete or not robust. Therefore, you should think about what you want to do and how your actions depend on other settings before recording. These questions can guide you through a recording:

- What do I want to record in one go?
  - Write down all steps for complex recordings that you don't miss anything.
- What are the default settings of the panels?
- Are there dependencies on active models?
  - How can they be eliminated?

Example 1: Surface integral

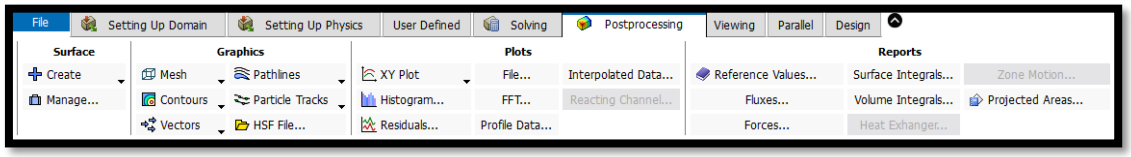
Calculating a surface integral is easy but it requires a number of clicks to get to something like the mass weighted average of the temperature of a secondary phase. This can be reduced by adding a recorded journal to a user-defined menu or even defining a keyboard shortcut to it. Both possibilities are covered later, this chapter is just about the recording.

Important

These steps can be different depending on the version of Fluent. The graphical user interface is changing starting with Fluent 16 to improve usability. Unfortunately, this has negative effects on recorded journals. Still, they can be of use but you should make notes of your steps in case you have to fix a journal recorded with a previous version.

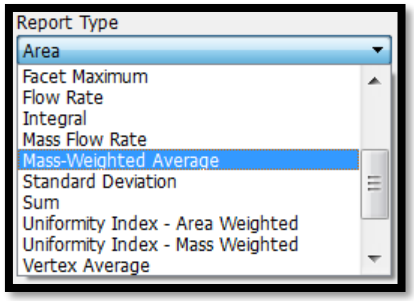
Before you start a recording, go through the steps that need to be recorded. In this case:

1. Select the Postprocessing ribbon

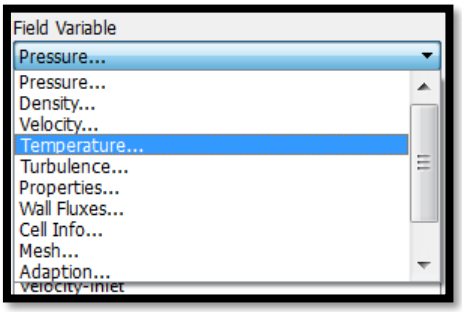


2. Click on "Surface Integrals" from the list.

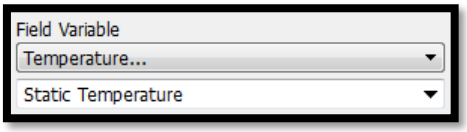
3. Select the report type "Mass-Weighted Average" which requires two clicks and some scrolling.



4. Select the field variable “Temperature” which also requires two clicks and some scrolling.

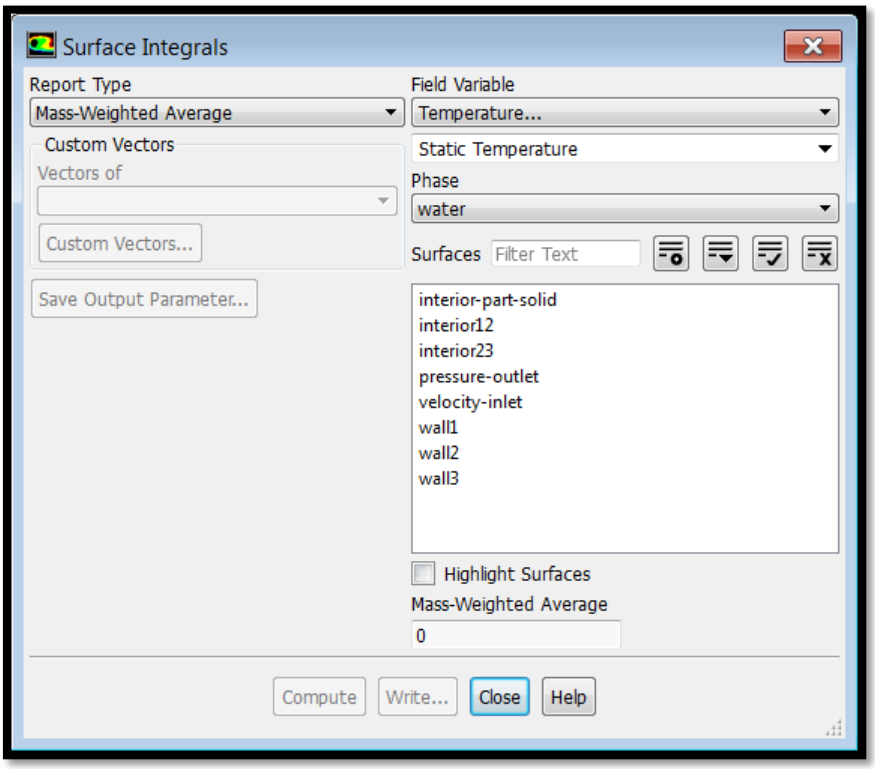


5. Select the field variable “Static Temperature” which should be the default.



6. Select the phase which can also require two clicks.

7. The panel should stay open that you can select the desired surfaces from the list.



Of course you could also get to the panel directly in the tree. The tree structure changed more than the ribbon over the last versions which made recorded journals less reliable compared to the ribbon.

Now that you know the steps, make sure you select something else in the panel, e.g. area-weighted average of the static pressure of the mixture phase.

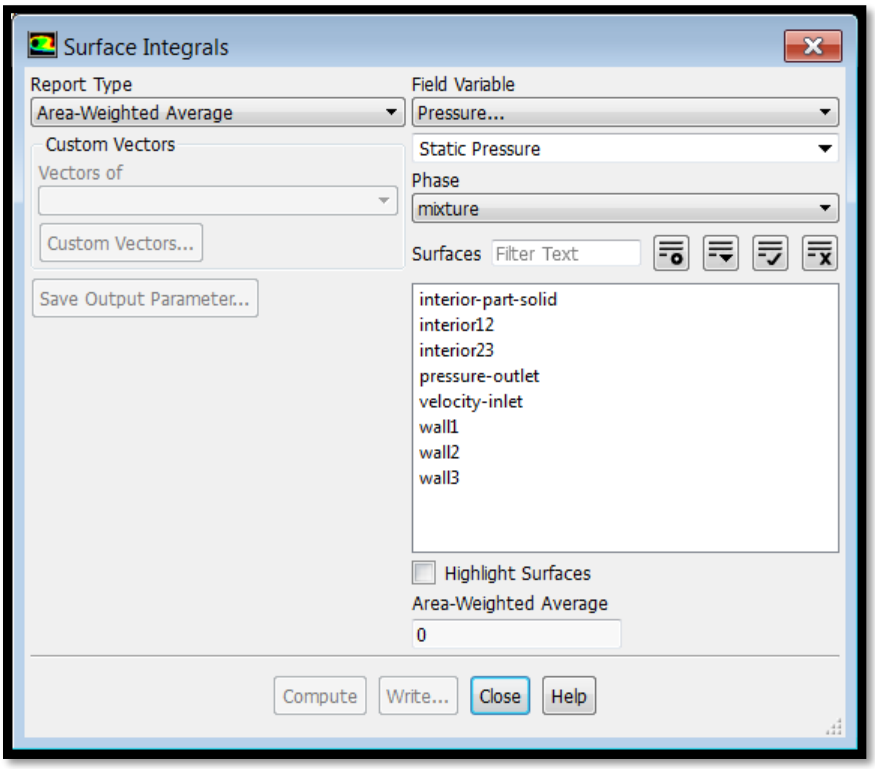


Figure 1: Initial state of the surface integrals panel to start journal recording

Then close the panel and select a different ribbon.

It is important that nothing of your desired workflow is pre-selected because this can make the recorded journal useless and you have to start from the beginning.

Then start the recording from the menu File > Write > Start Journal… and provide a name (e.g. 01-recording.jou).

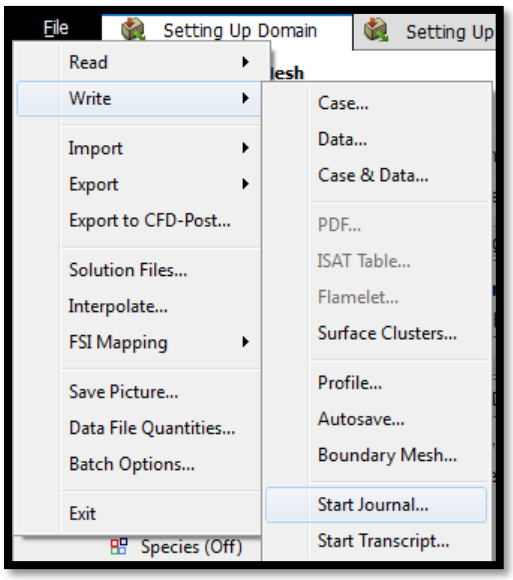


Figure 2: Start journal recording from File > Write > Start Journal...

Record all steps mentioned above and stop the recording from the menu File > Write > Stop Journal.

You can find a video of all steps on the ANSYS Customer Portal.

Let's look at the recording and figure out how it works.

```
001 (cx-gui-do cx-activate-item
    "Ribbon*Frame1*Frame6(Postprocessing)*Table1*Table3(Reports)*PushButton4(Su
    rface Integrals)")
002 (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table1*DropDownList1(Report Type)" '( 11))
003 (cx-gui-do cx-activate-item "Surface Integrals*Table1*DropDownList1(Report
    Type)")
004 (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList1(Field Variable)" '( 11))
005 (cx-gui-do cx-activate-item "Surface Integrals*Table2*DropDownList1(Field
    Variable)")
006 (cx-gui-do cx-set-list-selections "Surface Integrals*Table2*DropDownList2"
    '( 0))
007 (cx-gui-do cx-activate-item "Surface Integrals*Table2*DropDownList2")
008 (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList3(Phase)" '( 1))
009 (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList3(Phase)")
010 (cx-gui-do cx-activate-item "MenuBar*WriteSubMenu*Stop Journal")
```

Code fragment 1: Recorded journal without edits (01-recording.scm)

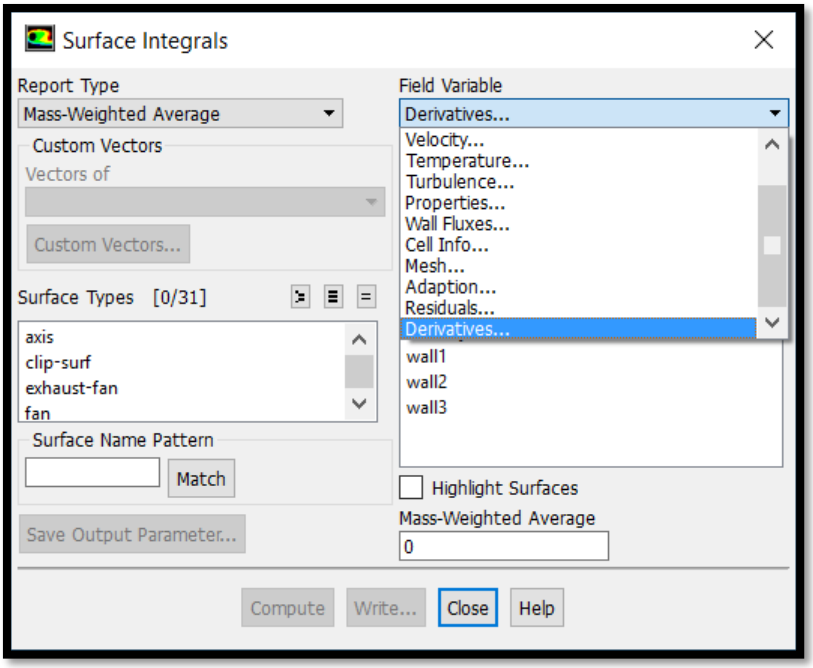
`cx-gui-do` are actions you did with the mouse. Each line starts with this Scheme command.

`cx-set-…` and `cx-activate-…` are selections or button clicks. You learn more about these in the chapter about panel creation.

Line 1:	Open the “Surface Integrals” panel by selecting a certain button on the ribbon. Note the structure with ribbon, frame, table or button and some number. If additional buttons are added to the ribbon in future versions, this might break this line of Scheme code.
Line 2-3:	Select the 12th item in the “Report Type” list and evaluate its content.
Line 4-5:	Select the 12th item in the “Field Variable” list and evaluate its content.
Line 6-7:	Select the first item in the second list in table two of the panel structure and evaluate its content.
Line 8-9:	Select the second item in the “Phase” list and evaluate its content.
Line 10:	Stop the recording of the journal.

There are three possible problems with this journal:

1. Selecting a specific item from a list with the index is dangerous. It would be better to use a name in case the index changes in future versions or due to a different choice of models. For example, if multiphase is not active temperature is the fourth item in the list, not the 12th. Therefore, the journal selects something else.



2. If the energy equation is not active temperature does not exist at all. This can be a problem when working with names in lists.
3. If Euler-Euler is not active, it is not possible to select the phase.

Making the journal robust

To make the journal more robust you can replace the numbers with names like they appear in the drop-down lists. The changed line numbers are marked in purple. You can also delete the last line because it serves no purpose during the execution of the journal.

For more information about the used macros you can refer to solutions 2042681 [1] and 2042682 [2].

```
001 (cx-gui-do cx-activate-item
    "Ribbon*Frame1*Frame6(Postprocessing)*Table1*Table3(Reports)*PushButton4(Su
    rface Integrals)")
002 (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table1*DropDownList1(Report Type)" '( "Mass-Weighted Average"))
003 (cx-gui-do cx-activate-item "Surface Integrals*Table1*DropDownList1(Report
    Type)")
004 (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList1(Field Variable)" '( "Temperature..."))
005 (cx-gui-do cx-activate-item "Surface Integrals*Table2*DropDownList1(Field
    Variable)")
006 (cx-gui-do cx-set-list-selections "Surface Integrals*Table2*DropDownList2"
    '( "Static Temperature"))
007 (cx-gui-do cx-activate-item "Surface Integrals*Table2*DropDownList2")
008 (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList3(Phase)" '( 1))
009 (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList3(Phase)")
```

Code fragment 2: Fix list selections (02-lists.scm)

Remember:

- Replace the number exactly like it is written in the panel or the script selects nothing at all.
- You need both, `cx-set-list-selections` and `cx-activate-item`. The first one makes the selection and the second one calls the associated callback procedure. (See panel creation.
- The index of the phase should not be replaced with the name because this is likely to change with each project. To select the secondary phase, always use index 1.
- The lists start with `'(` including the whitespace character behind the opening parenthesis. Instead of `'( 1)` you can also write `(list 1)`.
- The file extension can be either *.jou or *.scm. Since it contains only Scheme statements the provided examples have the extension *.scm which can be loaded in Fluent from the backstage menu File > Read > Scheme…
- Save the script with a new name for each change in case modifications break something in the code that you might notice at a much later stage.
- Test the script after each change to track down errors early.

Note that lines 6 and 7 might not be recorded automatically for versions before Fluent 18. You would need to make a separate recording and add them to be sure the static temperature is selected. In Fluent 18 you can just open the dropdown and click on the default value to include it in a recorded journal.

Now it's time to add some conditionals to rule out certain errors.

First, if no data is available Fluent reports an error. This can be avoided by adding two lines at the top. Don't forget to close the parentheses in the last line. At the end you can add an error message if you like.

```
001 (if (data-valid?)
002   (begin
003     (cx-gui-do cx-activate-item
    "Ribbon*Frame1*Frame6(Postprocessing)*Table1*Table3(Reports)*PushButton4(Su
    rface Integrals)")
...
011     (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList3(Phase)")
012   )
013   (display "\n Error! Data not available!\n Please load the data file for
    post-processing.\n\n"))
```

Code fragment 3: Adding check for valid data (03-data-check.scm)

Next you can rule out the phase. The panel should still open if Euler-Euler is not active but it should select either no phase or the mixture phase depending on the active models. The previous lines 10 and 11 moved to 12 and 13 in the following code. Note the added parentheses at the end of line 13 to close the begin-statement of line 11.

```
009     (cx-gui-do cx-activate-item "Surface Integrals*Table2*DropDownList2")
010     (if (eqv? (sg-mphase?) 'multi-fluid)
011       (begin
012         (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList3(Phase)" '( 1))
013         (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList3(Phase)"))
014       (if (or (eqv? (sg-mphase?) 'vof) (eqv? (sg-mphase?) 'drift-flux))
015         (begin
016           (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList3(Phase)" '( 0))
017           (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList3(Phase)")))))
018   (display "\n Error! Data not available!\n Please load the data file for
    post-processing.\n\n"))
```

Code fragment 4: Multiphase considerations (04-multiphase.scm)

Line 10: Check if Euler-Euler is active.

Line 11-13: Select the secondary phase from the phase drop-down and evaluate it.

Line 14: Check if VOF or Mixture model is active

Line 15-17: Select the mixture phase from the phase drop-down and evaluate it.

Do nothing if the multiphase model is not recognized.

Next is the energy equation. The if-statement goes into line 6 and is closed in the new line 18. If the energy equation is disabled, the panel still opens and selects the report type. But the remaining settings are left out.

```
005     (cx-gui-do cx-activate-item "Surface
        Integrals*Table1*DropDownList1(Report Type)")
006     (if (rf-energy?) (begin
007       (cx-gui-do cx-set-list-selections "Surface
        Integrals*Table2*DropDownList1(Field Variable)" '( "Temperature..."))
...
017             (cx-gui-do cx-set-list-selections "Surface
        Integrals*Table2*DropDownList3(Phase)" '( 0))
018             (cx-gui-do cx-activate-item "Surface
        Integrals*Table2*DropDownList3(Phase)"))))))
019   (display "\n Error! Data not available!\n Please load the data file for
        post-processing.\n\n"))
```

Code fragment 5: Check for energy equation (05-energy.scm)

If you play this through all possible combinations, you might recognize that it is still not robust enough. It can happen that the static temperature is not selected correctly because it is not available for the phase that is pre-selected when the panel opens. Therefore, it is best to move the selection of the temperature from lines 9 and 10 to the end of the script.

```
008       (cx-gui-do cx-activate-item "Surface
        Integrals*Table2*DropDownList1(Field Variable)")
009       (if (eqv? (sg-mphase?) 'multi-fluid)
...
016             (cx-gui-do cx-activate-item "Surface
        Integrals*Table2*DropDownList3(Phase)"))))
017       (cx-gui-do cx-set-list-selections "Surface
        Integrals*Table2*DropDownList2" '( "Static Temperature"))
018       (cx-gui-do cx-activate-item "Surface
        Integrals*Table2*DropDownList2"))))
019   (display "\n Error! Data not available!\n Please load the data file for
        post-processing.\n\n"))
```

Code fragment 6: Fix temperature selection when a phase is pre-selected (06-fix-temperature-selection.scm)

Finally, you can wrap everything into a procedure that you can reuse it easily.

```
001 (define (my-surface-integral-temperature-secondary-phase)
002   (if (data-valid?)
003     (begin
004       (cx-gui-do cx-activate-item
"Ribbon*Frame1*Frame6(Postprocessing)*Table1*Table3(Reports)*PushButton4(Su
rface Integrals)")
005       (cx-gui-do cx-set-list-selections "Surface
Integrals*Table1*DropDownList1(Report Type)" '( "Mass-Weighted Average"))
006       (cx-gui-do cx-activate-item "Surface
Integrals*Table1*DropDownList1(Report Type)")
007       (if (rf-energy?) (begin
008         (cx-gui-do cx-set-list-selections "Surface
Integrals*Table2*DropDownList1(Field Variable)" '( "Temperature..."))
009         (cx-gui-do cx-activate-item "Surface
Integrals*Table2*DropDownList1(Field Variable)")
010         (if (eqv? (sg-mphase?) 'multi-fluid)
011           (begin
012             (cx-gui-do cx-set-list-selections "Surface
Integrals*Table2*DropDownList3(Phase)" '( 1))
013             (cx-gui-do cx-activate-item "Surface
Integrals*Table2*DropDownList3(Phase)"))
014           (if (or (eqv? (sg-mphase?) 'vof) (eqv? (sg-mphase?) 'drift-flux))
015             (begin
016               (cx-gui-do cx-set-list-selections "Surface
Integrals*Table2*DropDownList3(Phase)" '( 0))
017               (cx-gui-do cx-activate-item "Surface
Integrals*Table2*DropDownList3(Phase)"))))
018         (cx-gui-do cx-set-list-selections "Surface
Integrals*Table2*DropDownList2" '( "Static Temperature"))
019         (cx-gui-do cx-activate-item "Surface
Integrals*Table2*DropDownList2"))))
020     (display "\n Error! Data not available!\n Please load the data file for
post-processing.\n\n")))
```

Code fragment 7: Complete surface integral code (07-si-temp-secondary-phase.scm)

Of course you can also choose a shorter name if you want to access it from the TUI. However, using describing names is recommended. You can always create an alias to have quick access.

When hardening recorded journals, it is best to observe how the Fluent panels change with the different models you use. But you should always keep in mind that making a script rock solid can be very time consuming.

To have permanent access to this procedure you can save it in a text file. Name it ".fluent" and put it in your user profile folder.

You can find your user profile by typing

`cd ~`

in a Linux shell or

`%userprofile%`

in the address bar of a Windows Explorer (not Internet Explorer!).

Fluent reads the .fluent file each time it starts which gives you permanent access to this procedure. Only Scheme commands are allowed in this file.

See also ANSYS Fluent User's Guide [4], The .fluent File for a brief description of that configuration file.

Note: For Windows systems it can be difficult to write files that start with a period. Not all text editors support it and the Explorer denies renaming a file like that. If your favorite text editor and Explorer don't work, you can always revert to the Windows command prompt or Windows PowerShell to rename that file.

Creating keyboard shortcuts

You can create keyboard shortcuts for any Scheme procedure or text command with the variable *cx-key-map*.

```
001 (set! *cx-key-map*
002   (append *cx-key-map*
003     '(
004       ("control shift e" . "exit ok")
005      )
006   )
007 )
```

Code fragment 8: Example code to define keyboard shortcuts (01-keyboard-exit.scm)

The dotted pair in line 4 is the definition of the keyboard shortcut. The first string of the pair is the shortcut itself as string. The second string contains the command or commands. You can put a large number of text commands or multiple Scheme procedures in that string.

To add multiple shortcuts, simply add more dotted pairs after line 4.

In this example the keyboard shortcut CTRL+SHIFT+E ends the Fluent session immediately. You have to specify all arguments because you cannot start a text command with a keyboard shortcut and fill in the rest in the text interface.

You can use the keywords control and shift. The alt key cannot be used. But you can use all function keys (F1, F2…).

There are some shortcuts defined already. Using the code above overwrites the definition for the current Fluent session. You can show all defined keyboard shortcuts with

```
(display *cx-key-map*)
```

Important

Keyboard shortcuts only work if the graphics window is active. If you clicked in the console, the ribbon or any other Fluent window, the shortcut does not work. When in doubt, simply left-click once in the graphic area and try the shortcut again.

To add a keyboard shortcut for the procedure from the last chapter and the quick exit, you can modify your .fluent file in the following way.

```
001 (define (my-surface-integral-temperature-secondary-phase)
002   (if (data-valid?)
003     (begin
004       (cx-gui-do cx-activate-item
    "Ribbon*Frame1*Frame6(Postprocessing)*Table1*Table3(Reports)*PushButton4(Su
    rface Integrals)")
005       (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table1*DropDownList1(Report Type)" '( "Mass-Weighted Average"))
006       (cx-gui-do cx-activate-item "Surface
    Integrals*Table1*DropDownList1(Report Type)")
007       (if (rf-energy?) (begin
008         (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList1(Field Variable)" '( "Temperature..."))
009         (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList1(Field Variable)")
010         (if (eqv? (sg-mphase?) 'multi-fluid)
011           (begin
012             (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList3(Phase)" '( 1))
013             (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList3(Phase)"))
014           (if (or (eqv? (sg-mphase?) 'vof) (eqv? (sg-mphase?) 'drift-flux))
015             (begin
016               (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList3(Phase)" '( 0))
017               (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList3(Phase)"))))
018         (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList2" '( "Static Temperature"))
019         (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList2"))))
020     (display "\n Error! Data not available!\n Please load the data file for
    post-processing.\n\n")))
021
022 (set! *cx-key-map*
023   (append *cx-key-map*
024     '(
025       ("control shift e" . "exit ok")
026       ("control shift m" . "(my-surface-integral-temperature-secondary-
    phase)"))))
```

Code fragment 9: Example of a .fluent file to map new keyboard shortcuts

This gives you permanent access to your recorded journal with the keyboard shortcut CTRL + SHIFT + M from the graphics window.

The order in the file is important. You need to define your procedures before using or referencing them. Therefore, you have all procedures at the top of the .fluent file, all keyboard bindings and menu items at the end.

Note: If you are on Linux and want to access the provided .fluent file remember that files that start with a dot are usually hidden. This is not the case on Windows systems.

Creating menu items

You can find a description of how to create menu items in the ANSYS Fluent Customization Manual [3], Part II, Adding Menus to the Right of the Ribbon.

You can add a new top level menu with

```
(cx-add-menu "name" #f)
```

Example:

```
> (cx-add-menu "UD Post" #f)
243588616
```

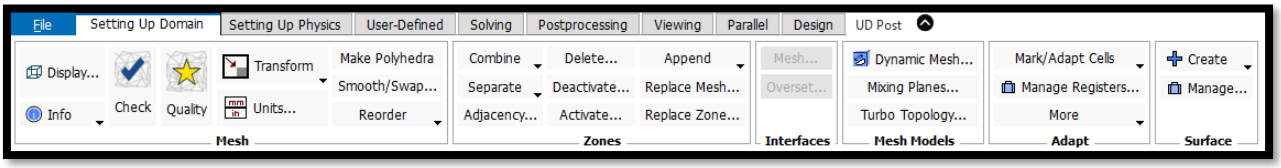


Figure 3: New menu item added right of the Fluent Ribbon

There are two possibilities to identify a menu to create menu items and sub menus: name and id. The id is the number you see in the screenshot. It is more robust than using the name because it can happen that you use a name twice. To avoid such conflicts, it's best to assign a menu to a variable.

Important

When experimenting with menu items, restart Fluent often to avoid conflicts.

```
> (define menu-udpost (cx-add-menu "UD Post" #f))
menu-udpost
```

Note: You can use local variables to store the ids of menu items. This will prevent you from extending your menu easily therefore it is recommended to use global variables.

The #f in the procedure call is a shortcut that could be used together with the ALT key to access the menu. This does not work anymore with Fluent 17 and above. Therefore, this should always be `#f`.

Now that you have a menu, you can also add sub menus.

```
(cx-add-hitem parent "submenu" #f)
```

“parent” is either the name of the parent (sub) menu as string or the variable that holds the id of the parent item.

```
> (define menu-udpost-udreport (cx-add-hitem menu-udpost "UD Report" #f))
menu-udpost-udreport
```

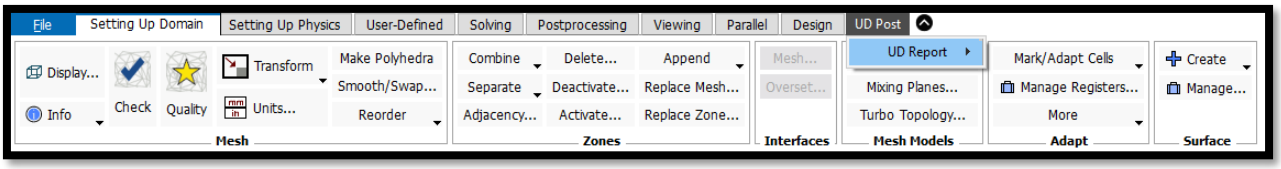


Figure 4: Submenu item added to the existing user-defined menu

You add menu items that execute procedures with

```
(cx-add-item parent "item" #f #f #t my-procedure)
```

```
> (cx-add-item "UD Report" "Temperature Second Phase" #f #f #t my-surfaceintegral-temperature-secondphase)
372959216
```

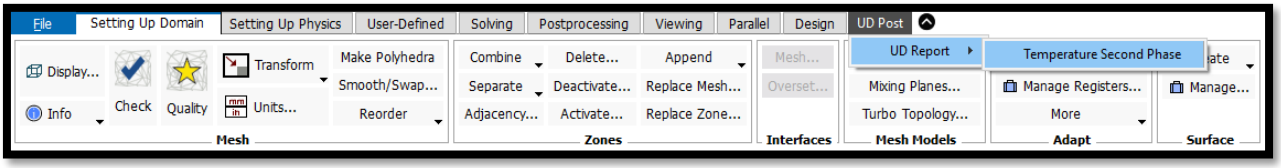


Figure 5: Item to call a procedure added to the user-defined menu

In this case you should not enclose the procedure with parentheses. Just provide the name itself like you use a variable. The procedure has to exist before you add the menu item.

Actually, Fluent makes a copy of the procedure for the menu item. This means if you change the procedure by redefining it, you can still execute the version that you used to create the menu.

Of course you can put a menu also in the .fluent file.

The order in which you place keyboard shortcuts and menu items is not important. All required procedures must be above both but there is no dependency between shortcuts and menus.

```
001 (define (my-surface-integral-temperature-secondary-phase)
002   (if (data-valid?)
003     (begin
004       (cx-gui-do cx-activate-item
    "Ribbon*Frame1*Frame6(Postprocessing)*Table1*Table3(Reports)*PushButton4(Su
    rface Integrals)")
005       (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table1*DropDownList1(Report Type)" '( "Mass-Weighted Average"))
006       (cx-gui-do cx-activate-item "Surface
    Integrals*Table1*DropDownList1(Report Type)")
```

```
007       (if (rf-energy?) (begin
008         (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList1(Field Variable)" '( "Temperature..."))
009         (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList1(Field Variable)")
010         (if (eqv? (sg-mphase?) 'multi-fluid)
011           (begin
012             (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList3(Phase)" '( 1))
013             (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList3(Phase)"))
014           (if (or (eqv? (sg-mphase?) 'vof) (eqv? (sg-mphase?) 'drift-flux))
015             (begin
016               (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList3(Phase)" '( 0))
017               (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList3(Phase)"))))
018         (cx-gui-do cx-set-list-selections "Surface
    Integrals*Table2*DropDownList2" '( "Static Temperature"))
019         (cx-gui-do cx-activate-item "Surface
    Integrals*Table2*DropDownList2"))))
020     (display "\n Error! Data not available!\n Please load the data file for
    post-processing.\n\n")))
021
022 (set! *cx-key-map*
023   (append *cx-key-map*
024     '(
025       ("control shift e" . "exit ok")
026       ("control shift m" . "(my-surface-integral-temperature-secondary-
    phase)"))))
027
028 (define menu-udpost (cx-add-menu "UD Post" #f))
029 (define menu-udpost-udreport (cx-add-hitem menu-udpost "UD Report" #f))
030 (define menu-udpost-udreport-temperature (cx-add-item menu-udpost-udreport
    "Temperature Second Phase" #f #f #t my-surface-integral-temperature-
    secondary-phase))
```

Code fragment 10: Example of a .fluent file with the definition of new menu items

Creating ribbon items

The ribbon works similar to menu items.

Adding new tabs

You can add a new tab with cx-add-ribbon-tab:

```
(define my-tab (cx-add-ribbon-tab "My tab")
```

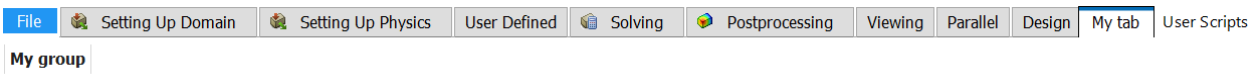
Ensure that the name of the tab is unique. Although the id is stored in the variable, groups are created with the name, not with the id.

You cannot add tabs in Fluent Meshing but you can add groups to the ribbon.

Adding new groups to existing tabs

You add groups with cx-add-ribbon-group:

```
(define my-group (cx-add-ribbon-group "My tab" "My group"))
```



Unlike menus, groups are always added with the name of the tab. It doesn't seem to be possible to use the id of a ribbon tab to add groups.

To add groups in Fluent Meshing, use the tab name "Task Page".

Adding buttons to existing groups

Once you have a group, you can add buttons.

It is good practice to embed the buttons either in a table or a frame, using three rows and any number of columns. Frames and Tables are discussed in more detail later in this document.

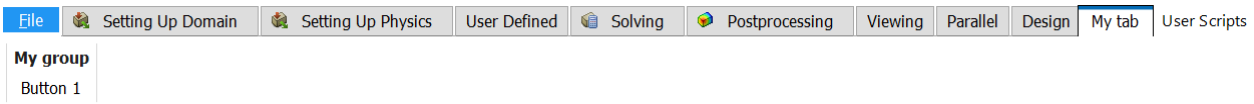
A table is added with cx-create-table:

```
(define my-table (cx-create-table my-group "" 'border #f))
```

It is invisible, but you can use it to place the buttons with cx-create-button which is also described in more detail in chapter Push button.

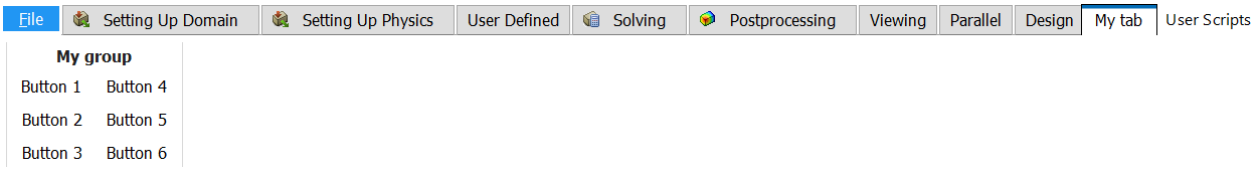
```
(define my-button-1 (cx-create-button my-table "Button 1" 'activate-callback my-procedure-button-1 'row 1 'col 1))
```

Of course, the procedure "my-procedure-button-1" must be defined before it is used during button generation.



The complete code for a new ribbon tab with one group and six buttons could look like this:

```
001 ; Button callback procedures
002 (define (my-procedure-button-1) (display "Button 1 pressed\n"))
003 (define (my-procedure-button-2) (display "Button 2 pressed\n"))
004 (define (my-procedure-button-3) (display "Button 3 pressed\n"))
005 (define (my-procedure-button-4) (display "Button 4 pressed\n"))
006 (define (my-procedure-button-5) (display "Button 5 pressed\n"))
007 (define (my-procedure-button-6) (display "Button 6 pressed\n"))
008 ;
009 ; New tab
010 (define my-tab (cx-add-ribbon-tab "My tab"))
011 ; New group
012 (define my-group (cx-add-ribbon-group "My tab" "My group"))
013 ; Buttons
014 (define my-button-1 (cx-create-button my-group "Button 1" 'activate-
    callback my-procedure-button-1 'row 1 'col 1))
015 (define my-button-2 (cx-create-button my-group "Button 2" 'activate-
    callback my-procedure-button-2 'row 2 'col 1))
016 (define my-button-3 (cx-create-button my-group "Button 3" 'activate-
    callback my-procedure-button-3 'row 3 'col 1))
017 (define my-button-4 (cx-create-button my-group "Button 4" 'activate-
    callback my-procedure-button-4 'row 1 'col 2))
018 (define my-button-5 (cx-create-button my-group "Button 5" 'activate-
    callback my-procedure-button-5 'row 2 'col 2))
019 (define my-button-6 (cx-create-button my-group "Button 6" 'activate-
    callback my-procedure-button-6 'row 3 'col 2))
```



Essentially, you can use all container and primitive elements in a ribbon that are described in chapter Creating panels.

Creating panels

Designing a panel (=window) for Fluent can be confusing. It is best to use a text editor that can keep track of parentheses. Place the Scheme file with your code in your Fluent working directory. For quick access, define an alias (see Introduction to Fluent scripting, part 2) to read the file. In the following example the file name is "`panel.scm`" which can be quick-loaded by typing "`lp`" in the Fluent TUI.

```
001 (alias 'lp
002   (lambda ()
003     (load "panel.scm")))
```

Code fragment 11: Defining an alias for quick loading a Scheme file (01-alias-definition.scm)

When you are finished with your panel, you can put it into your .fluent file and create menu items or keyboard shortcuts like described in the previous chapters for easy access.

You can also find all examples as *.scm files in the zip file available on the ANSYS Customer Portal.

Structure of panels

Panels and task pages follow a strict hierarchy:

1. At the top level you have the panel which is identical with a window.
2. Inside the panel you have at least one container. They are used to give the panel structure.
3. Inside container you have either more container or primitive objects.
4. Primitive objects provide a graphical interface to interact with your scripts and UDFs by providing user-data or showing return values (results).

You can control most of the parameters for all of these objects with six types of procedures:

- `(cx-create-… parent-id label options-list)`
- `(cx-enable-… id flag)`
- `(cx-activate-… id)`
- `(cx-set-… id value)`
- `(cx-show-… id)`
- `(cx-hide-… id)`

Replace the three dots of the first four types of procedures by the type, for example with "panel" to modify a window or "text" for simple text:

- `(cx-create-panel label options-list)`
- `(cx-create-text parent-id label options-list)`

In the following chapters you learn about all of the common objects and the available options.

Panels

A panel is a window that runs in the context of the Fluent Cortex process. You define a panel with `cx-create-panel` followed by its name and some options.

```
(cx-create-panel label options-list)
```

The label is the title of the window. You provide it with a simple string or with a variable that holds a string.

Three options are available:

- `'apply-callback`
  - Procedure that is executed when you click on the OK button.

- 'update-callback
  - Procedure that is executed when you open the panel.
- 'close-callback
  - Procedure that is executed when you either click on the X or on the Cancel button

Note: There is also a help button that provides a hard link to the ANSYS Java help system. You can't change the behavior. Therefore, this button is useless for user-defined panels but it can't be removed either.

Panels have ids just like menu items. It is good practice to assign this id to a variable for easy access. Most of the examples use local variable definitions but if you want to put your panel in the .fluent file, you have to use a global variable for the panel itself. Furthermore, global variables allow you to modify your panels and allow interaction between them.

```
001 (let ((my-panel-id (cx-create-panel
002                                "My Panel"
003                                'apply-callback #t
004                                'update-callback #f
005                                'close-callback #f)))
006   (cx-show-panel my-panel-id)
007 )
```

Code fragment 12: Basic definition of a panel without content (02-basic-panel.scm)

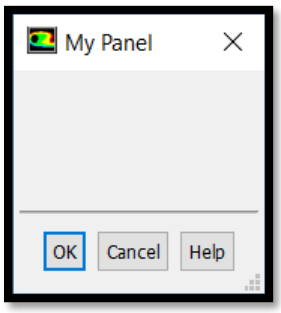


Figure 6: User-defined panel without content

Line 1: Create a local variable with the name "my-panel-id". It holds the unique identifier of the new panel that is created with "cx-create-panel". The parameters for that procedure are split up into several lines.

Line 2: Name of the panel as string. This name is used as title of the new window.

Line 3: You can define an optional procedure that executes when the OK button is pressed. In this case this procedure just returns #t. If you specify #f, the OK button disappears.

Line 4: You can define an optional procedure that executes when the panel opens. This is usually used to read RP variables. In this case nothing happens because the panel just gets the Boolean value #f.

Line 5: You can define an optional procedure that executes when you press the Cancel button. In this case nothing happens.

Line 6: Show the panel with the id stored in the variable "my-panel-id".

Objects inside a panel can be placed using relative orientation. This will become clearer in one of the following examples. The available positions are:

- 'below
- 'right-of

- `'above`
- `'left-of`

In general, you should only use below and right-of because it is difficult enough already to keep track of all the objects inside a panel when you use a structured layout.

You can execute three different callback procedures for a panel. They are executed when you open a panel, close it or click on the OK or Apply buttons. Of course, you have to define the callback (cb) procedures before you define the panel. You can use global or local procedure definitions.

```
001 (let* ((cb-apply (lambda () (display "You clicked OK.\n")))
002        (cb-upd (lambda () (display "You just opened this panel.\n")))
003        (cb-close (lambda () (display "You closed the panel without applying
    changes.\n")))
004        (my-panel-id (cx-create-panel
005                                "My Panel"
006                                'apply-callback cb-apply
007                                'update-callback cb-upd
008                                'close-callback cb-close)))
009   (cx-show-panel my-panel-id)
010 )
```

Code fragment 13: Panel definition with callback procedures (03-panel-with-callback.scm)

Note that the cb procedures don't use parentheses because they are not executed during the definition of the panel. You provide only the text without any quotes.

If you don't need the cb procedure for close you can neglect the options and provide the names of the cb procedures in the order apply > update.

```
001 (let* ((cb-apply (lambda () (display "You clicked OK.\n")))
002        (cb-upd (lambda () (display "You just opened this panel.\n")))
003        (my-panel-id (cx-create-panel "My Panel" cb-apply cb-upd)))
004   (cx-show-panel my-panel-id)
005 )
```

Code fragment 14: Short definition of callback procedures for panels (04-short-panel-callback.scm)

If you want to close a panel with a script you can use

`(cx-hide-panel panel-id)`

Remember that the panel-id must be accessible which most likely means that the definition must be global, not local like in all of these examples.

Task pages

Task pages are very similar to panels. But they don't have an apply callback procedure and the close callback procedure is mostly useless because task pages are usually not closed. They are replaced by opening other task pages on top of them. Most of the time it's more convenient to use panels instead of task pages.

`(cx-create-taskpage label options-list)`

```
001 (let* ((cb-upd (lambda () (display "You just opened this task page.\n")))
002        (cb-close (lambda () (display "You closed the task page.\n")))
003        (my-tp-id (cx-create-taskpage
004                                "My Panel"
005                                'update-callback cb-upd
006                                'close-callback cb-close)))
007   (cx-show-taskpage my-tp-id)
008 )
```

Code fragment 15: Example definition of a task page with callback procedures (05-task-page.scm)

Note that the update cb procedure is called twice when a task page is opened.

The rest of this document concentrates on panels. Most of the techniques apply to task pages, too.

Container

Container are required to place so-called primitive objects like buttons. Although strictly speaking, containers are not required unless you want to control the layout of you panel in detail.

Fluent knows three types of containers:

- Frame
- Table
- Button Box

All three are used in the same way. They differ in how you can place objects inside them.

Frame

A frame allows relative placement of primitives. You create a frame with:

`(cx-create-frame parent-id label options-list)`

The parent-id is the id of the panel or container the new container is placed in. The label is a string for the headline. You can add one, multiple or no additional options:

- `'tabbed`
  - Boolean value, default: `#f`
- `'border`
  - Boolean value, default: `#t`

In addition to these two you can also define the position using the placement options of the parent object. If you do not specify the location of the frame, Fluent places it to the top left corner. If there is another object already it might be placed on top of it. Therefore, you should always specify the location.

Example 1:

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002       (my-frame-id))
003   (set! my-frame-id (cx-create-frame my-panel-id "Frame" 'below 0 'right-of
    0))
004   (cx-show-panel my-panel-id)
005 )
```

Code fragment 16: Example of a simple frame without content (06-basic-frame.scm)

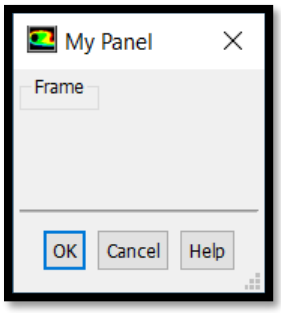


Figure 7: A single frame container with border

Example 2:

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002         (my-frame-id1)
003         (my-frame-id2)
004         (my-frame-id3))
005   (set! my-frame-id1 (cx-create-frame my-panel-id "Frame" 'border #f 'below
    0 'right-of 0))
006   (set! my-frame-id2 (cx-create-frame my-frame-id1 "Inner Frame 1" 'border
    #t 'below 0 'right-of 0))
007   (set! my-frame-id3 (cx-create-frame my-frame-id1 "Inner Frame 2" 'border
    #t 'below my-frame-id2))
008   (cx-show-panel my-panel-id)
009 )
```

Code fragment 17: Frame definition with border and placement but without content (07-frame-border.scm)

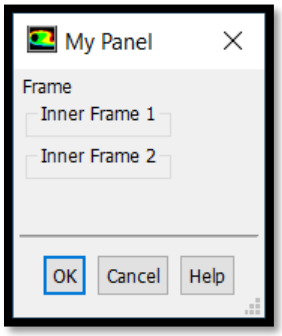


Figure 8: Three frame containers nested and placed

Line 2 - 4: Names of three variables that hold the unique identifiers of frames.

Line 5: Create a new frame inside the panel identified by "`my-panel-id`". It has the headline "`Frame`" and no border.

Line 6: Create a new frame inside the first frame. It has the headline "`Inner Frame 1`" and a border.

Line 7: Create a new frame inside the first frame. It has the headline "`Inner Frame 2`", a border and it is located below the frame defined in line 10.

For orientation inside a frame you can use the options `'below`, `'right-of`, `'above` and `'left-of`.

You can also use frames to create tabs.

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002       (my-frame-id1)
003       (my-frame-id2)
004       (my-frame-id3))
005   (set! my-frame-id1 (cx-create-frame my-panel-id "" 'tabbed #t))
006   (set! my-frame-id2 (cx-create-frame my-frame-id1 "tab1"))
007   (set! my-frame-id3 (cx-create-frame my-frame-id1 "tab2"))
008   (cx-show-panel my-panel-id)
009 )
```

Code fragment 18: Convert frames to tabs (08-tabs.scm)

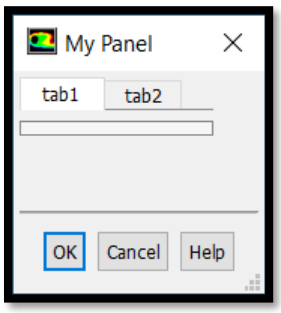


Figure 9: Tabs without content

Line 2 - 4: Names of three variables that hold the unique identifiers of frames.

Line 5: Create a frame to create tabs. A name is not required. In this case the position can be ignored because it is the only top level container but it is possible to place tabs below other objects.

Line 6: Define another frame to be the first tab with the name “`tab1`”.

Line 7: Define a third frame to be the second tab with the name “`tab2`”.

There are no options to define the position of the individual tabs. Each frame that has the tabbed frame as parent object is placed in the same order it is written in the code.

Table

You create tables with

```
(cx-create-table parent-id label options-list)
```

In addition to the position you have the following options available:

- `'border`
  - Boolean value, default: `#t`
- `'visible-rows`
  - Integer value, default: `0`

You cannot use relative positions inside a table. Instead you can specify row and column with the keywords:

- `'row`
- `'col`

Rows and columns start to count at 1. Usually the top left cell is row 1 and column 1. However, if you use the option `'visible-rows` Fluent counts the rows from bottom to the top. Columns are still from left to right, though.

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002         (my-table-id)
003         (my-frame-id1)
004         (my-frame-id2)
005         (my-table-id3)
006         (my-frame-id4))
007   (set! my-table-id (cx-create-table my-panel-id "Table with 3 elements"
    'border #t 'below 0 'right-of 0))
008   (set! my-frame-id1 (cx-create-frame my-table-id "1-1" 'row 1 'col 1))
009   (set! my-frame-id2 (cx-create-frame my-table-id "2-2" 'row 2 'col 2))
010   (set! my-table-id3 (cx-create-table my-table-id "3-3 - Table with 2
    visible rows" 'visible-rows 2 'row 3 'col 3))
011   (set! my-frame-id4 (cx-create-frame my-table-id3 "3-1-1" 'row 1 'col 1))
012   (cx-show-panel my-panel-id)
013 )
```

Code fragment 19: Nested tables (09-table.scm)

Line 2 - 6: Names of variables that hold the unique identifiers of tables and frames.

Line 7: Create a new table inside the panel identified by "`my-panel-id`". It has the headline "`Table with 3 elements`".

Line 8: Create a new frame in the first row and first column of the table with the headline "`1-1`".

Line 9: Create a new frame in the second row and second column of the table with the headline "`2-2`".

Line 10: Create a new table in the third row and third column of the main table with the headline "`3-1-1`". It has two visible rows.

Line 11: Create a new frame in row 1 and column 1 of the table created in line 14.

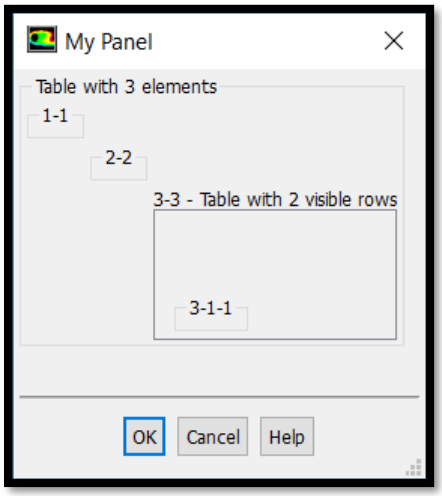


Figure 10: Nested tables and frames

Button box
You can think of button boxes as simplified tables that contain only one row or one column. You create them with

```
(cx-create-button-box parent-id label options-list)
```

In addition to the position you can use:

- `'radio-mode`
  - Boolean value, default: `#f`
- `'vertical`
  - Boolean value, default: `#t`

You cannot use relative or absolute positioning of objects inside a button box. Fluent handles the positioning for you.

In order to use button boxes, you need the primitive object “button”. Therefore, you can find an example in the chapter about buttons.

You can hide frames, tables and button boxes with

```
(cx-hide-item container-id)
```

Remember that hiding an item might break the layout because the relative placement of other objects might no longer be valid.

Primitives

There are ten commonly used primitives:

- Text
- Push button
- Toggle button
- Text entry
- Integer entry
- Real entry
- List
- Drop-down
- Scale
- Dial

Like container, you should assign the id of primitives to a variable that you can use it for relative placement of other objects (container or primitives).

Text
Text objects display text like the label of a container.

```
(cx-create-text parent-id label options-list)
```

Besides the position you can use the option:

- `'border`
  - Boolean value, default: `#f`

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002         (my-frame-id)
003         (my-text-id1)
004         (my-text-id2)
005         (my-text-id3))
006   (set! my-text-id1 (cx-create-text my-panel-id "Headline without frame"
      'border #t 'below 0 'right-of 0))
007   (set! my-frame-id (cx-create-frame my-panel-id "Frame" 'border #t 'below
      my-text-id1 'right-of 0))
008   (set! my-text-id2 (cx-create-text my-frame-id "First text in a frame"
      'below 0 'right-of 0))
009   (set! my-text-id3 (cx-create-text my-frame-id "Below first text" 'below
      my-text-id2 'right-of 0))
010   (cx-show-panel my-panel-id)
011 )
```

Code fragment 20: Text primitives and frames (10-text-primitives.scm)

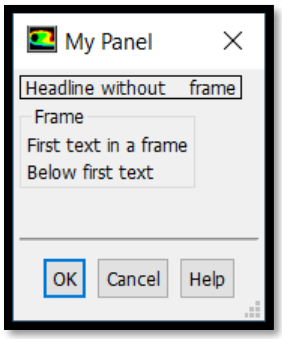


Figure 11: Text primitives placed in panel and frame

Line 2 - 5: Names of variables that hold the unique identifiers of frames and text.
Line 6: Define a text with a border and place it at the top of the panel.
Line 7: Define a frame with border below the first text.
Line 8: Define text as first element inside the frame.
Line 9: Define another text object and place it below the first text object in the frame.

Push button
You use push buttons to execute procedures on demand. The callback functions have to be declared before you define the button.

```
(cx-create-button parent-id label options-list)
```

Besides the position you can use the option:

- `'activate-callback`
  - Procedure (without parentheses), default: none

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002         (my-frame-id)
003         (my-button))
004   (set! my-frame-id (cx-create-frame my-panel-id "Frame" 'border #t 'below
    0 'right-of 0))
005   (set! my-button (cx-create-button my-frame-id "Push me" 'activate-
    callback #f 'below 0 'right-of 0))
006   (cx-show-panel my-panel-id)
007 )
```

Code fragment 21: Push button without cb procedure (11-button.scm)

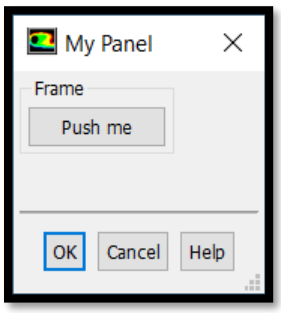


Figure 12: Push button inside frame

Line 2 - 3: Names of variables that hold the unique identifiers of frame and button.

Line 4: Define a frame with the headline "`Frame`".

Line 5: Define a push button with the text "`Push me`". The callback function is `#f` which means that nothing happens when you click on the button.

A button can execute any global or local procedure. It can even modify other GUI elements.

```
001 (let* ((my-panel-id (cx-create-panel "My Panel"))
002         (my-frame-id)
003         (my-button)
004         (button-cb (lambda ()
005           (display "Closing panel\n")
006           (cx-hide-panel my-panel-id))))
007   (set! my-frame-id (cx-create-frame my-panel-id "Frame" 'border #t))
008   (set! my-button (cx-create-button my-frame-id "Close panel" 'activate-
    callback button-cb ))
009   (cx-show-panel my-panel-id)
010 )
```

Code fragment 22: Push button with cb procedure (12-button-close.scm)

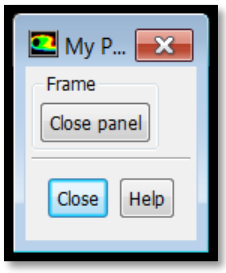


Figure 13: Push button inside Frame that calls a procedure to close the panel

Line 2 - 3: Names of variables that hold the unique identifiers of frame and button.

Line 4 - 6: Local callback procedure for the button which prints a text to the console and closes the panel.

Line 7: Define a frame with the headline "`Frame`".

Line 8: Define a push button with the text "`Close panel`" and the callback procedure that was defined in line 4.

Toggle button

You can use toggle buttons for checkboxes and radio buttons. The difference is made with the button box container, not with the button object itself. Toggle buttons have the same options as push buttons.

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002          (my-frame-id)
003          (my-button)
004          (bcb (lambda () (display "Toggled button\n"))))
005   (set! my-frame-id (cx-create-frame my-panel-id "Frame" 'border #t))
006   (set! my-button (cx-create-toggle-button my-frame-id "Toggle me"
    'activate-callback bcb))
007   (cx-show-panel my-panel-id)
008 )
```

Code fragment 23: Toggle button in a frame (13-toggle-button.scm)

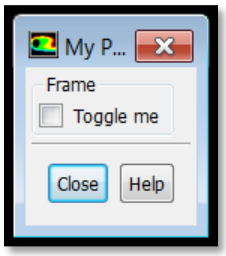


Figure 14: Toggle button inside frame

Line 2 - 3: Names of variables that hold the unique identifiers of frame and button.

Line 4: Callback procedure for the button that prints some text to the console.

Line 5: Define a frame with the label "`Frame`" inside the panel.

Line 6: Define a toggle button with the title "`Toggle me`". When you click on it the procedure from line 4 runs.

There is no 'deactivate-callback option although Fluent does not complain if you use this option. The same procedure is executed regardless of the status of the button. You have to make sure that the procedure tracks the changes and the correct status.

You can get the current status of the toggle button with
`(cx-show-toggle-button id)`

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002         (my-frame-id)
003         (my-button)
004         (bcb (lambda ()
005           (if (eqv? (cx-show-toggle-button my-button) #t)
006               (display "Toggle active now\n")
007               (display "Toggle inactive now\n")))))
008   (set! my-frame-id (cx-create-frame my-panel-id "Frame" 'border #t))
009   (set! my-button (cx-create-toggle-button my-frame-id "Toggle me"
    'activate-callback bcb))
010   (cx-show-panel my-panel-id)
011 )
```

Code fragment 24: Toggle button with callback procedure (14-toggle-cb.scm)

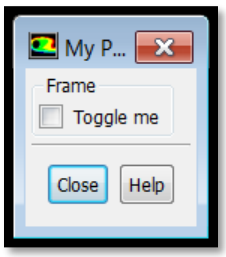


Figure 15: Toggle button with callback procedure

Line 4 – 7: Definition of the procedure for the button which prints the new state to the console.

Buttons and button boxes
Button boxes are special container for buttons. As described earlier, they act like a simple table with only one column (default) or one row.

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002         (my-bb-id)
003         (my-b1)
004         (my-b2)
005         (my-b3))
006   (set! my-bb-id (cx-create-button-box my-panel-id "Button box" 'border
    #t))
007   (set! my-b1 (cx-create-button my-bb-id "Button 1" 'activate-callback #f))
008   (set! my-b2 (cx-create-toggle-button my-bb-id "Button 2" 'activate-
    callback #f))
009   (set! my-b3 (cx-create-toggle-button my-bb-id "Button 3" 'activate-
    callback #f))
010   (cx-show-panel my-panel-id)
011 )
```

Code fragment 25: Buttons inside a button box (15-button-box.scm)

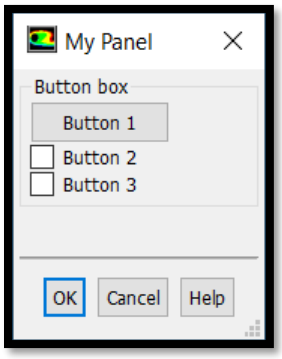


Figure 16: Push and toggle buttons in a button box

Note that there are no position options used in lines 11 to 13. If you use an option like `'below` Fluent reports an error “Object does not recognize attribute” but it shows the panel nonetheless.

To change the orientation, add the option `'vertical #f` to line 6:

```
(set! my-bb-id (cx-create-button-box my-panel-id "Button box" 'border #t 'vertical #f))
```

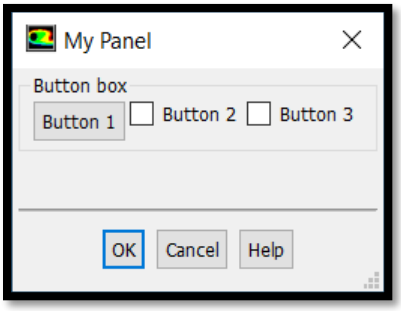


Figure 17: Horizontal orientation of push and toggle buttons in a button box

You can convert toggle buttons to radio buttons with the option `'radio-mode #t`. Push buttons ignore this option.

```
(set! my-bb-id (cx-create-button-box my-panel-id "Button box" 'border #t 'radio-mode #t))
```

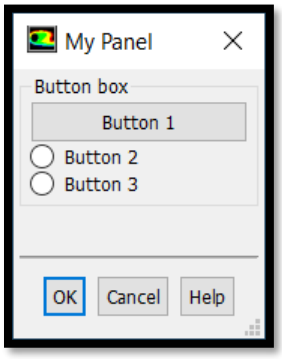


Figure 18: Radio mode for toggle buttons in a button box

If you want to read the value of radio buttons, you can use the same procedure described before for the toggle buttons. Even though only one of the buttons can return true, you might need to check all buttons to find the one that is true.

Text entry

You can use text entries to input strings for your scripts. Callback procedures are called when you hit return but not when you click outside of the text field. Make sure you evaluate all the inputs with the callback function for the OK button of the panel.

```
(cx-create-text-entry parent-id label options-list)
```

There are many options available. Most of them are also available for integer and real entries:

- 'editable
  - Boolean value, default: #t
- 'label-position
  - Symbol, default: 'above, possible other values: 'below, 'right-of, 'left-of
- 'width
  - Integer, default: 10
- 'activate callback
  - Procedure, default: none
- 'value
  - String, default: “”
- 'password-mode
  - Boolean value, default: #f
- 'symbol-mode
  - Boolean value, default: #f

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002         (my-frame-id)
003         (my-text1) (my-text2)
004         (my-text3) (my-text4)
005         (my-text5) (my-text6)
006         (my-text7) (my-text8)
007         (my-text9) (my-text10)
008         (my-frame2) (my-text11)
009         (my-cb (lambda () (display "cb\n"))))
010   (set! my-frame-id (cx-create-frame my-panel-id "Text fields" 'border #t
    'below 0 'right-of 0))
011   (set! my-text1 (cx-create-text-entry my-frame-id "Text 1" 'activate-
    callback my-cb 'below 0 'right-of 0 'width 20))
012   (set! my-text2 (cx-create-text-entry my-frame-id "Text 2" 'activate-
    callback my-cb 'below 0 'right-of my-text1 'border #t))
013   (set! my-text3 (cx-create-text-entry my-frame-id "Text 3" 'below my-text1
    'editable #f))
014   (set! my-text4 (cx-create-text-entry my-frame-id "Text 4" 'activate-
    callback my-cb 'below my-text2 'label-position 'below))
015   (set! my-text5 (cx-create-text-entry my-frame-id "Text 5" 'activate-
    callback my-cb 'below my-text3 'label-position 'left-of))
016   (set! my-text6 (cx-create-text-entry my-frame-id "Text 6" 'activate-
    callback my-cb 'below my-text4 'label-position 'right-of))
017   (set! my-text7 (cx-create-text-entry my-frame-id "Text 7" 'activate-
    callback my-cb 'below my-text5 'width 5))
018   (set! my-text8 (cx-create-text-entry my-frame-id "Text 8" 'activate-
    callback my-cb 'below my-text6 'value "pre-filled value"))
019   (set! my-text9 (cx-create-text-entry my-frame-id "Text 9" 'activate-
    callback my-cb 'below my-text7 'value "pre-filled value" 'password-mode
    #t))
```

```
020   (set! my-text10 (cx-create-text-entry my-frame-id "Text 10" 'activate-
      callback my-cb 'below my-text8 'value "pre-filled value" 'symbol-mode #t))
021   (set! my-frame2 (cx-create-frame my-panel-id "Width" 'border #f 'below
      my-frame-id))
022   (set! my-text11 (cx-create-text-entry my-frame2 "Text 11" 'activate-
      callback my-cb 'below 0 'right-of 0 'width 5))
023   (cx-show-panel my-panel-id)
024   )
```

Code fragment 26: Different options for text entry fields (18-text-entry-options.scm)

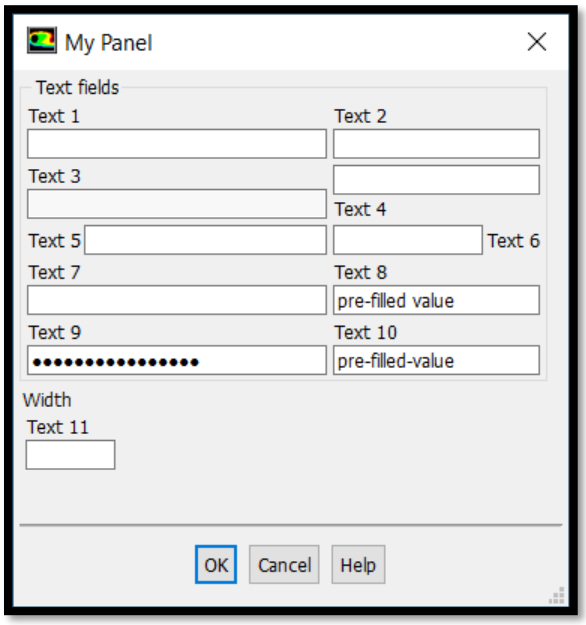


Figure 19: Examples for the different options of text entry primitive objects

Line 2 – 8: Definition of the local variables for frames and text entry fields

Line 9: Definition of the callback procedure which prints "`cb`" to the Fluent console. It is executed when you hit enter after filling in a text field.

Line 10: Definition of a frame with the title "`Text fields`".

Line 11: First text field with a width of 20 symbols.

Line 12: Second text field with a border and the default width. You can use the keyword 'border but it has no effect on text fields.

Line 13: A non-editable text field located below the first one. It has the same width although no width is specified here. Non-editable text fields are usually used to show string values of variables.

Line 14: The fourth text field has the title below the entry. It is located below the second one. Note that it is not necessary to specify 'below and 'right-of. One of both is sufficient to identify the location without conflict.

Line 15: The fifth text field has the title left of the entry field. The width is reduced accordingly.

Line 16: The sixth text field has the title right of the entry field. The width is reduced accordingly.

Line 17: Although the width is reduced, the text field is still 20 characters long because of the definition of the first text field in the frame.

Line 18: A text field which is pre-filled with some text.

Line 19: A text field with password mode which is pre-filled with some text. All characters are masked with dots.

Line 20: A text field with symbol mode which is pre-filled with some text. Symbol mode converts the input to a valid string that can be used as Scheme symbol. Characters that are not allowed (like the white space) are converted to a minus.

Line 21: Definition of another frame below the first one.

Line 22: A last text entry field with reduced width. This time it works because it is the first text field in the frame.

You can read the input with

```
(cx-show-text-entry id)
```

Integer entry

You can get integer inputs for your scripts with integer entry fields. They work more or less like text entry fields.

```
(cx-create-integer-entry parent-id label options-list)
```

Most options are identical with the text field:

- `'editable`
  - Boolean value, default: `#t`
- `'label-position`
  - Symbol, default: `'above`, possible other values: `'below`, `'right-of`, `'left-of`
- `'width`
  - Integer, default: `10`
- `'activate callback`
  - Procedure, default: none
- `'value`
  - Integer, default: `0`
- `'increment`
  - Integer, default: `1`
- `'maximum`
  - Integer, default: none
- `'minimum`
  - Integer, default: none

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002          (my-frame-id)
003          (my-int1) (my-int2)
004          (my-int3) (my-int4)
005          (my-int5) (my-int6)
006          (my-cb (lambda () (display "cb\n"))))
007   (set! my-frame-id (cx-create-frame my-panel-id "Integer fields" 'border
      #t))
008   (set! my-int1 (cx-create-integer-entry my-frame-id "Integer 1" 'activate-
      callback my-cb 'width 3))
009   (set! my-int2 (cx-create-integer-entry my-frame-id "Integer 2" 'activate-
      callback my-cb 'right-of my-int1 'width 4))
010   (set! my-int3 (cx-create-integer-entry my-frame-id "Integer 3" 'activate-
      callback my-cb 'below my-int1 'width 5))
011   (set! my-int4 (cx-create-integer-entry my-frame-id "Integer 4" 'activate-
      callback my-cb 'below my-int2 'increment 5 'value 3 'width 4))
```

```
012   (set! my-int5 (cx-create-integer-entry my-frame-id "Integer 5" 'activate-
      callback my-cb 'below my-int3 'maximum 4))
013   (set! my-int6 (cx-create-integer-entry my-frame-id "Integer 6" 'activate-
      callback my-cb 'below my-int4 'minimum -3 'width 4))
014   (cx-show-panel my-panel-id)
015   )
```

Code fragment 27: Options for the integer entry fields (19-integer-entry-options.scm)

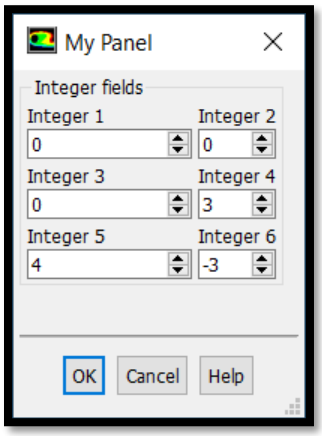


Figure 20: Example for options of integer entry primitive objects

Line 2 – 5: Definition of the local variables for frames and text entry fields

Line 6: Definition of the callback procedure which prints "`cb`" to the Fluent console. It is executed when you hit enter after typing in a number or each time you click on the increment/decrement buttons on the right side of the text field.

Line 7: Definition of a frame with the title "`Integer fields`".

Line 8: First integer entry field with a width of 3. This is ignored because not all items in this column have the same width.

Line 9: Second integer entry field with a width of 4. All the items in this column have the same width (lines 11, 13), therefore the width is respected.

Line 11: Although an increment of 5 is specified for field 4, it is possible to start with other values. It is also possible to type in different values. The increment is only active for the increment/decrement arrows. If you want to make sure only certain increments are entered, you have to consider this in the different callback functions.

Line 12/15: For minimum and maximum values you cannot enter values outside of the range. Of course you can combine both options for a single integer field.

You get the value of an integer entry with `(cx-show-integer-entry id)`.

Real entry

A real entry is very similar to an integer entry. You don't have the increment/decrement buttons, though.

`(cx-create-real-entry parent-id label options-list)`

You have the following options available:

- `'unit-quantity`
  - Symbol, default: none, use the symbols as written in the units panel (quantities) in Fluent
- `'editable`
  - Boolean value, default: `#t`
- `'label-position`

    - Symbol, default: `'above`, possible other values: `'below`, `'right-of`, `'left-of`
- `'width`
    - Integer, default: `10`
- `'activate callback`
    - Procedure, default: none
- `'value`
    - Number, default: `0`
- `'maximum`
    - Integer, default: none
- `'minimum`
    - Integer, default: none

```
001 (let ((my-panel-id (cx-create-panel "My Panel"))
002         (my-frame-id)
003         (my-real1) (my-real2)
004         (my-real3) (my-real4)
005         (my-real5) (my-real6)
006         (my-cb (lambda () (display "cb\n"))))
007   (set! my-frame-id (cx-create-frame my-panel-id "Real fields" 'border #t))
008   (set! my-real1 (cx-create-real-entry my-frame-id "Real 1" 'activate-
    callback my-cb 'width 3))
009   (set! my-real2 (cx-create-real-entry my-frame-id "Real 2" 'activate-
    callback my-cb 'right-of my-real1 'width 4))
010   (set! my-real3 (cx-create-real-entry my-frame-id "Real 3" 'activate-
    callback my-cb 'below my-real1 'width 5))
011   (set! my-real4 (cx-create-real-entry my-frame-id "Real 4" 'activate-
    callback my-cb 'below my-real2 'value 3e5 'width 4 ))
012   (set! my-real5 (cx-create-real-entry my-frame-id "Real 5" 'activate-
    callback my-cb 'below my-real3 'units-quantity 'length))
013   (set! my-real6 (cx-create-real-entry my-frame-id "Real 6" 'activate-
    callback my-cb 'below my-real4 'minimum -3 'width 4))
014   (cx-show-panel my-panel-id)
015 )
```

Code fragment 28: Different options for real entry fields (20-real-entry-options.scm)

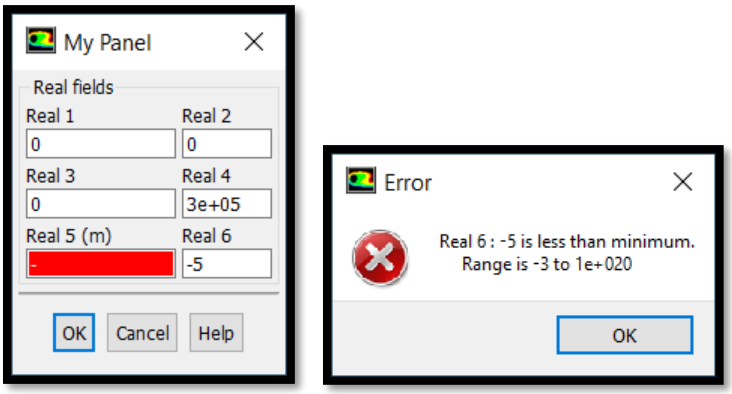


Figure 21: Example of options for real entry primitive objects with error message

Line 7: Definition of a frame with the title “`Real fields`”.

Line 8: First real entry field with a width of 3. This is ignored because not all items in this column have the same width.

Line 9: Second real entry field with a width of 4. All the items in this column have the same width (lines 11, 13), therefore the width is respected.

Line 11: Field 4 is pre-filled with the value `3e5`. The format is always the same.

Line 12: The length unit is added to the title of the field. In this case the Fluent case has the default unit meter assigned to the quantity length. The field itself is red because the input is invalid.

Line 15: For minimum and maximum values you cannot enter values outside of the range. The test is only done when you hit return. Fluent shows an error message that the value is out of range. If you don't hit enter the fields are evaluated before the callback function of the panel (click on OK) is called.

You can read the value of a real entry field with `(cx-show-real-entry id)`.

List

List selections require two procedures. The first one defines the list, the second fills the list.

```
(cx-create-list parent-id label options-list)
(cx-set-list-item parent-id list)
```

The following options are available:

- `'activate-callback`
  - Procedure, default: none
- `'mixed-case`
  - Boolean, default: `#f`
- `'multiple-selections`
  - Boolean, default: `#f`
- `'selectable`
  - Boolean, default: `#t`
- `'sorted`
  - Boolean, default: `#f`
- `'visible-lines`
  - Integer, default: `10`
- `'width`
  - Integer, default: `10`

```
001 (let* ((my-panel-id (cx-create-panel "My Panel"))
002          (my-frame-id)
003          (my-list1) (my-list2)
004          (my-list3) (my-list4)
005          (my-list5)
006          (my-cb (lambda () (display "cb\n"))))
007   (set! my-frame-id (cx-create-frame my-panel-id "Lists" 'border #t))
008   (set! my-list1 (cx-create-list my-frame-id "List 1" 'activate-callback
    my-cb 'width 30 'visible-lines 5))
009   (cx-set-list-items my-list1 (inquire-surface-names))
010   (set! my-list2 (cx-create-list my-frame-id "List 2" 'activate-callback
    my-cb 'right-of my-list1 'width 30 'visible-lines 3 'mixed-case #t))
011   (cx-set-list-items my-list2 (inquire-surface-names))
012   (set! my-list3 (cx-create-list my-frame-id "List 3" 'activate-callback
    my-cb 'below my-list1 'width 30 'visible-lines 3 'multiple-selections #t))
013   (cx-set-list-items my-list3 (inquire-surface-names))
```

```
014   (set! my-list4 (cx-create-list my-frame-id "List 4" 'activate-callback
      my-cb 'right-of my-list3 'width 30 'visible-lines 3 'sorted #t))
015   (cx-set-list-items my-list4 (inquire-surface-names))
016   (set! my-list5 (cx-create-list my-frame-id "List 5" 'below my-list3
      'width 40 'multiple-selections #t 'selectable #f))
017   (cx-set-list-items my-list5 (inquire-surface-names))
018   (cx-show-panel my-panel-id)
019 )
```

Code fragment 29: Options for list primitives (21-list-options.scm)

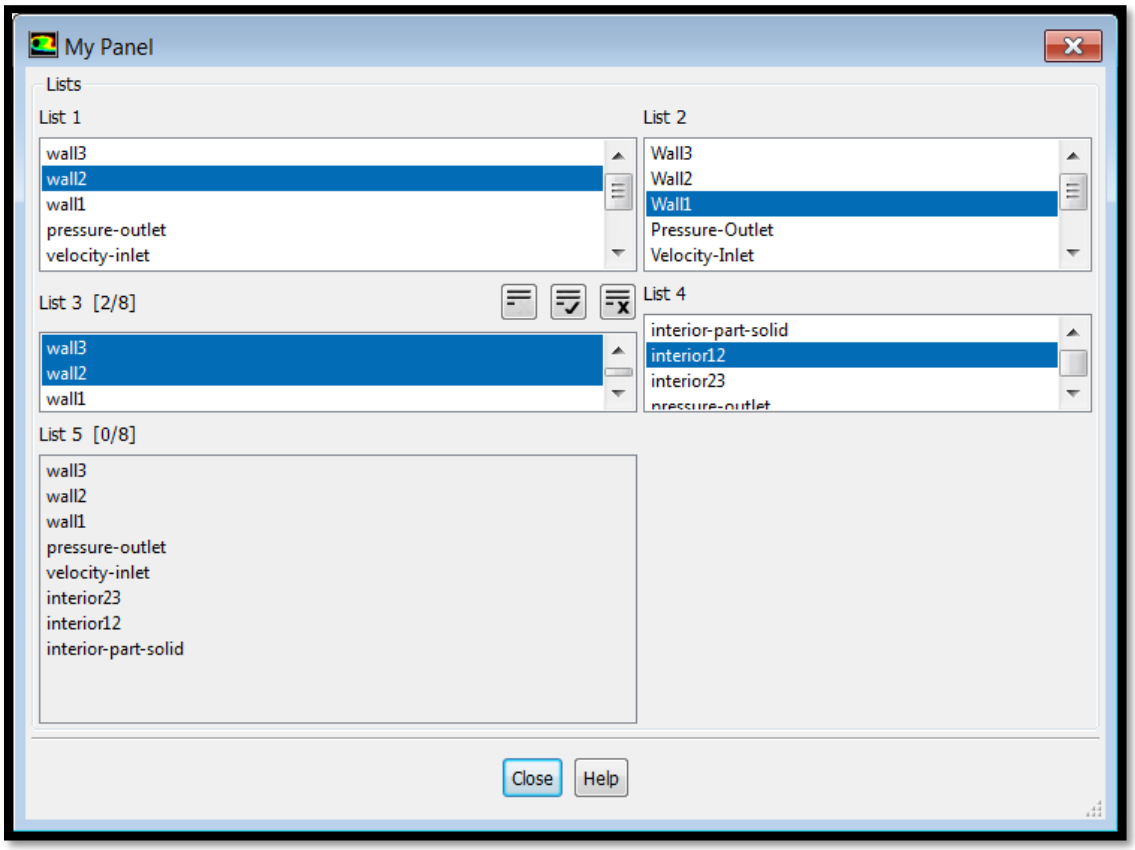


Figure 22: Lists with different selection types

Line 7: Definition of a frame with the title “`Lists`”.

Line 8: First list with a width of 30 and 5 visible lines.

Line 9: Display the list of all post-processing surfaces in “`my-list1`”.

Line 10: Second list with a width of 30 and only 3 visible lines. The number of lines of list 1 overwrites this setting. The mixed case setting makes the first letter of each word uppercase.

Line 12: Third list with a width of 30 and 3 visible lines. Multiple selections are allowed in this list which also shows the selection buttons.

Line 14: Fourth list with a width of 30 and 3 visible lines. The list is alphabetically sorted regardless of the order in the list.

Line 16: Fifth list with a width of 40 which overrides the width of the lists on top of it. Although multiple selections are allowed in theory, nothing can be selected due to the 'selectable option.

Note that the callback procedure is called every time an item is selected or deselected. Usually it is better to use the callback procedure of the panel (ok button) instead of assigning a callback procedure to each list.

To get a list of all the selected items you can use `(cx-show-list-selections id)`.

If you want to pre-select something use `(cx-set-list-selections id '(items))`. With `'(items) = '(1 2)` for the second and third element of the list or `'(items) = '(wall1 wall2)` to select the items with the names "`wall1`" and "`wall2`". If you use a name that does not exist, you don't get error messages.

Selecting an item doesn't call the associated callback procedure. A generic way to call this procedure without knowing its name is `(cx-activate-item id)`.

```
001 (let* ((my-panel-id (cx-create-panel "My Panel"))
002          (my-frame-id)
003          (my-list1)
004          (my-button1) (my-button2) (my-button3)
005          (my-cb (lambda () (display "Callback called\n")))
006          (my-button-select (lambda ()
007            (cx-set-list-selections my-list1 '( 0))))
008          (my-button-clear (lambda ()
009            (cx-set-list-selections my-list1 '( "**blank**"))))
010          (my-button-select-cb (lambda ()
011            (cx-set-list-selections my-list1 '( 0))
012            (cx-activate-item my-list1))))
013   (set! my-frame-id (cx-create-frame my-panel-id "Lists" 'border #t))
014   (set! my-list1 (cx-create-list my-frame-id "List 1" 'activate-callback
      my-cb 'width 30 'visible-lines 5 'multiple-selections #t))
015   (cx-set-list-items my-list1 (inquire-surface-names))
016   (set! my-button1 (cx-create-button my-frame-id "Select 1" 'below my-list1
      'activate-callback my-button-select))
017   (set! my-button2 (cx-create-button my-frame-id "Clear" 'below my-button1
      'activate-callback my-button-clear))
018   (set! my-button3 (cx-create-button my-frame-id "Select 1 CB" 'below my-
      button2 'activate-callback my-button-select-cb))
019   (cx-show-panel my-panel-id)
020 )
```

Code fragment 30: Example of using list selections with buttons (22-list-selections.scm)

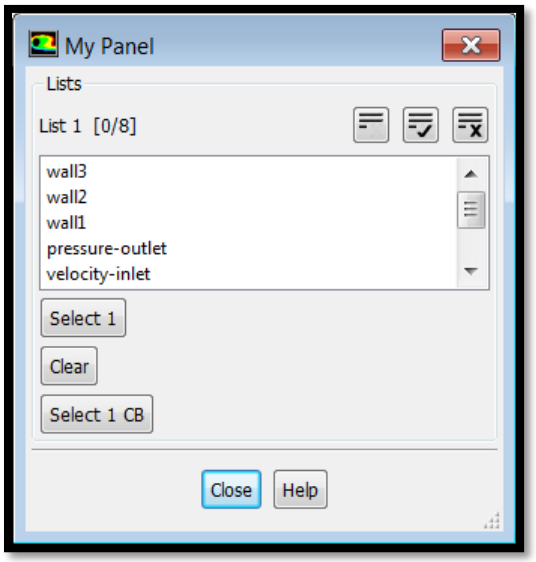


Figure 23: Combination of lists and buttons

Line 5: Callback procedure that is called from the list.

Line 6-7: Callback procedure for the first button to select the first item in the list.

Line 8-9: Callback procedure to clear the list selection by selecting an item that (most likely) doesn't exist.

Line 10-12: Callback procedure for the last button to select the first item in the list and call the cb procedure of the list.

Line 13: Definition of a frame with the title "`Lists`".

Line 14: Definition of a multi-selection list

Line 15: Populate the list with the names of the post-processing surfaces for the current case (works only if mesh is available).

Line 16-18: Definitions of buttons that call the different procedures from lines 6 to 12.

When you click on the first button you can see that the first item in the list is selected. But there is no output to the Fluent console from the cb procedure of the list. However, when you click into the list yourself the cb procedure is called.

The second button clears the selection by selecting an item that doesn't exist. At least it is not very likely that a list of post-processing surfaces contains an item with the name "**blank**". Alternatively, you could use a GUI procedure to virtually click on the "deselect all" button of the list by replacing line 9 with:

```
(cx-gui-do cx-activate-item "My Panel*List 1*PushButton3(fl_DeselectAll)")))
```

Note that you have to provide the full "path" to the button within the string using the names you used, starting with the name of the panel.

The last button of the panel also selects the first item of the list but in addition it calls the callback procedure by activating the list.

In principle, you can activate any primitive to call its cb procedure. This is not restricted to lists.

Drop-down

Drop-down lists are related to the lists of the last chapter.

```
(cx-create-drop-down-list parent-id label options-list)
(cx-set-list-item parent-id list)
```

The following options are available:

- 'activate-callback
  - Procedure, default: none
- 'mixed-case
  - Boolean, default: #f
- 'sorted
  - Boolean, default: #f
- 'width
  - Integer, default: 10

```
001 (let* ((my-panel-id (cx-create-panel "My Panel"))
002          (my-frame-id)
003          (my-list1) (my-list2)
004          (my-list3) (my-list4)
005          (my-cb (lambda () (display "cb\n"))))
006   (set! my-frame-id (cx-create-frame my-panel-id "Integer fields" 'border
      #t))
007   (set! my-list1 (cx-create-drop-down-list my-frame-id "List 1" 'activate-
      callback my-cb 'width 30))
008   (cx-set-list-items my-list1 (inquire-surface-names))
009   (set! my-list2 (cx-create-drop-down-list my-frame-id "List 2" 'right-of
      my-list1 'width 30 'mixed-case #t))
010   (cx-set-list-items my-list2 (inquire-surface-names))
011   (set! my-list3 (cx-create-drop-down-list my-frame-id "List 3" 'below my-
      list1 'width 40))
012   (cx-set-list-items my-list3 (inquire-surface-names))
013   (set! my-list4 (cx-create-drop-down-list my-frame-id "List 4" 'right-of
      my-list3 'width 30 'sorted #t))
014   (cx-set-list-items my-list4 (inquire-surface-names))
015   (cx-show-panel my-panel-id)
016 )
```

Code fragment 31: Example of the options available for drop-down lists (24-drop-down-options.scm)

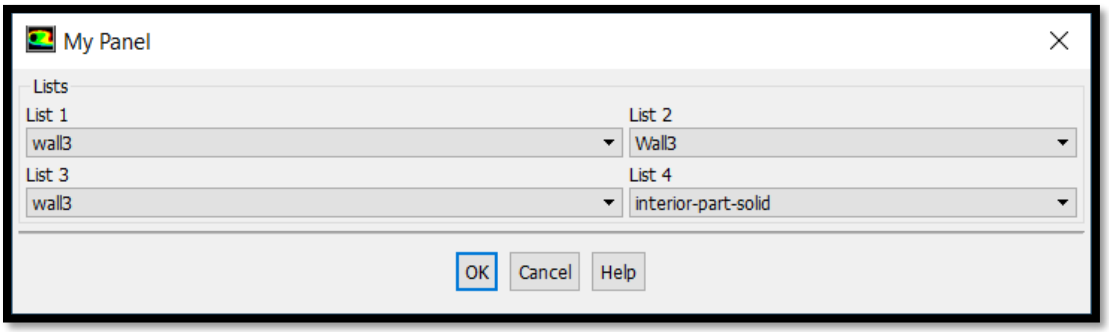


Figure 24: Drop-down lists with different display styles

Line 6: Definition of a frame with the title "Lists".

Line 7: First drop-down list with a width of 30 and a callback procedure. It is called after you select something from the drop-down.

Line 8: Populate the drop-down “`my-list1`” with a list of all post-processing surfaces.

Line 9: Second drop-down that capitalizes each word of the provided list.

Line 11: Third drop-down which overrides the width of the first one.

Line 13: Last drop-down with alphabetically sorted contents.

Like with selection lists you can use `(cx-show-list-selections id)` and `(cx-set-list-selections id '(item))` to read the selected item or to set the selection. Although only one item can be selected for dropdown lists, both macros return and accept only lists with a single item.

Scale

A scale is a horizontal slider to select integer values.

You have four options available:

- `'activate-callback`
  - Procedure, default: none
- `'minimum`
  - Minimum value for the scale, default: `0`
- `'maximum`
  - Maximum value for the scale, default: `10`
- `'value`
  - Integer, default: `0`

```
001 (let* ((my-panel-id (cx-create-panel "My Panel"))
002          (my-frame-id)
003          (my-scale1) (my-scale2)
004          (my-cb (lambda () (display "cb\n"))))
005   (set! my-frame-id (cx-create-frame my-panel-id "Scales" 'border #t))
006   (set! my-scale1 (cx-create-scale my-frame-id "Scale 1" 'activate-callback
    my-cb 'minimum -2 'maximum 5 'value 2))
007   (set! my-scale2 (cx-create-scale my-frame-id "Scale 2" 'below my-scale1))
008   (cx-show-panel my-panel-id)
009 )
```

Code fragment 32: Creating scales (25-scale-options.scm)

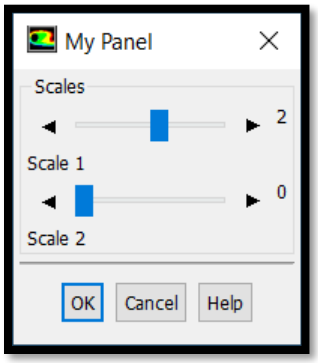


Figure 25: Two scales with default and user-defined bounds

Line 5: Definition of a frame with the title “`Scales`”.

Line 6: First scale with a minimum, maximum and default value. The callback procedure is executed every time you move the slider one increment.

Line 7: A default scale.

It is not possible to place the label of a scale at another location. It is always below the slider. Also, you cannot define the width of the slider. It's best if you have only a small number of increments.

You get the value selected by the scale with `(cx-show-scale id)`.

Note that the callback procedure is called for each increment. It can be more convenient to process the input value of a scale with the callback procedure of the panel.

Dial

You can think of a dial as a radial scale.

You have the same four options available:

- `'activate-callback`
  - Procedure, default: none
- `'minimum`
  - Minimum value for the scale, default: `0`
- `'maximum`
  - Maximum value for the scale, default: `360`
- `'value`
  - Integer, default: `0`

```
001 (let* ((my-panel-id (cx-create-panel "My Panel"))
002          (my-frame-id)
003          (my-dial)
004          (my-cb (lambda () (display (cx-show-dial my-dial)) (newline))))
005   (set! my-frame-id (cx-create-frame my-panel-id "Dial" 'border #t))
006   (set! my-dial (cx-create-dial my-frame-id "Dial 1" 'activate-callback my-
    cb 'minimum 0 'maximum 10 'value 3))
007   (cx-show-panel my-panel-id)
008 )
```

Code fragment 33: Using a dial for integer selection (26-dial-options.scm)

Figure 26: Dial with user-defined bounds

Line 4: Definition of a callback function that prints the current value to the Fluent console.

Line 5: Definition of a frame with the title "`Dial`".

Line 6: Definition of a dial with a minimum value of 0, a maximum value of 10 and an initial value of 3. The callback function is called every time you click the dial, rotate it or release the mouse button.

Because the callback procedure is called very often be careful when you add complex procedures. It can be better to use either simple procedures or only use the callback of the panel (ok button).

To read the value of a dial use the procedure `(cx-show-dial id)`.

Creating a panel

In this chapter you learn how to create a panel from the planning to finishing the code. It's a more user-friendly version of the fourth example from solution 2042682 [2]. A compressed version is also available as standalone solution 2042772 [5].

Task description

Create a user-defined number of iso-surfaces for post-processing. The iso-surfaces are parallel to one of the three Cartesian planes and have equal distances. Direction and start and end coordinates can be defined by the user but reasonable default values should simplify the input. Also the name should be user-defined.

The script should be accessible with TUI and with a convenient panel. The main usage scenario is the panel, therefore it's not necessary that the script double checks every single input for sanity.

A help text should be available that describes how the panel works and how to create the iso-surfaces without the panel.

Designing the panel

The panel design is connected with the required features. You can combine the basic design process with developing the feature list. But both steps can be separated depending on who defines the features.

You should follow basic design rules:

- Logical structure of the different buttons and input fields on the panel
- Panel size suited for all screen resolutions
  - Buttons can't update dynamically when a window is resized. Only lists can adapt
- Inputs should go from left to right and top to bottom
- Reasonable default values should be available for the convenience of the user

First of all, the panel needs buttons to create the iso-surfaces, reset the input values and to show the help. Remember, that you can't use the default help button of the panel because it always points to the ANSYS help which can't be edited. These buttons should be at the bottom of the panel.

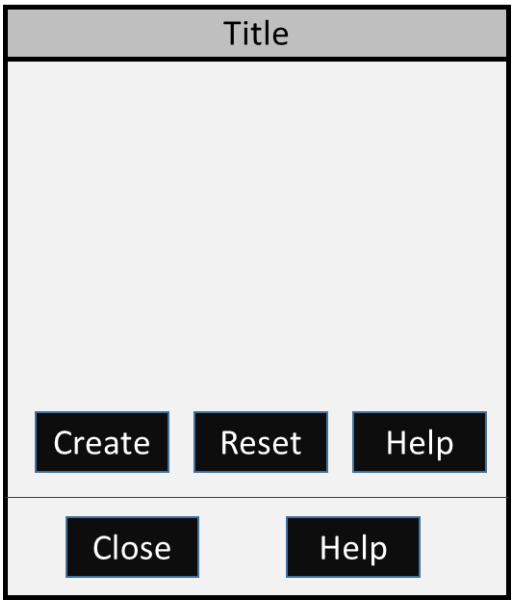


Figure 27: Sketch of the initial panel layout with the create, reset and help buttons

Then you need inputs for direction, coordinates, number of surfaces and name pattern. It's best to keep this order because of the dependencies between these inputs. The coordinates depend on the direction and the name might depend on direction, coordinates and the number of surfaces.

There are two to three options for the direction depending on the dimensions of the case. It's not possible to choose two directions at the same time which means that radio buttons or a dropdown menu is the best choice for a single selection. For this example, you can use radio buttons.

It's possible to update the coordinates automatically with the domain extents whenever you choose another direction. But this behavior might not be desired by all users. You can add a checkbox for this automatic update.

The default value for the radio buttons is the X direction and the automatic update should be disabled by default.

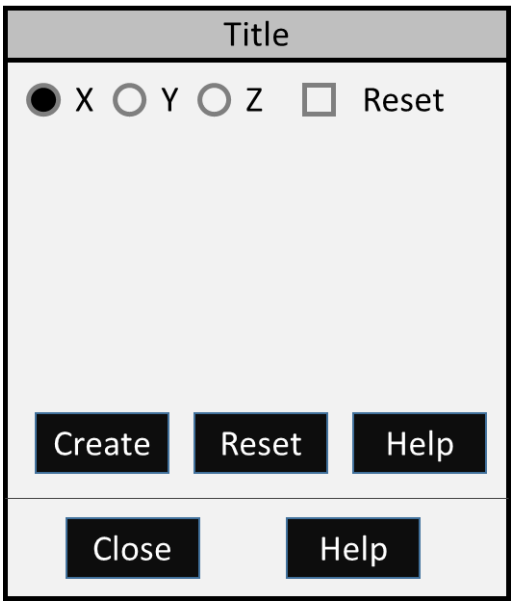


Figure 28: Sketch of panel with direction selection

The coordinates are real inputs. An additional button can reset min and max values to the domain extents in the chosen direction.

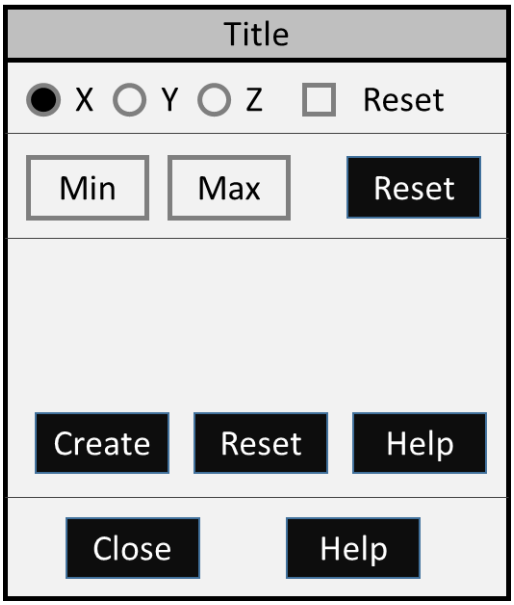


Figure 29: Sketch of panel with bounds definition

There is a number of possible objects to input integer values for the number of planes. For this example, two objects are used for demonstration purposes. You should avoid creating multiple inputs for the same value that you don't confuse the users of the panel.

The first input is a scale. The second input is an integer entry field. Both have to be coupled that they show the same value. However, if this is not possible the integer entry field should take priority over the scale. This has to be considered in the callback procedures.

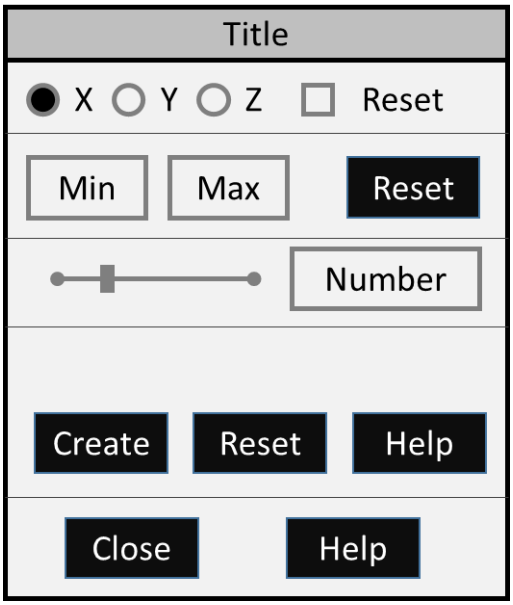


Figure 30: Sketch of panel with selection of number of surfaces

The last group contains several inputs and also an output to see an example for the name of the surfaces. A text entry field is used for the prefix. It's the base of the name which is identical for all surfaces.

Attached to that prefix is a suffix to uniquely identify the surface. The suffix can be a simple number, the location or both. At least one prefix is required which has to be considered in the callback procedures.

Another text entry field can be used to show the result of the chosen pre- and suffixes. Since you can't guarantee that the callback procedure of the first text entry is called, you should consider an additional button to update the output.

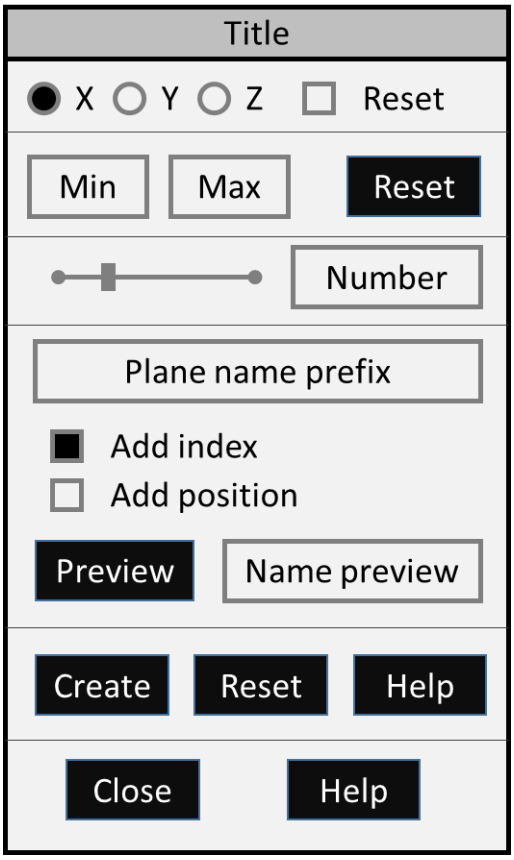


Figure 31: Final sketch of panel with all required GUI objects

Planning the code

The different elements of the panel require callback procedures that are executed automatically when required.

The panel itself has three possible callback procedures but only an automatic update is required when the panel is opened. It should populate all fields with default values. In fact, this callback procedure is identical with the procedure for the reset button at the bottom of the panel. It's best to plan this at the end because several other callback procedures can be re-used inside the panel reset procedure.

The three toggle buttons for X, Y and Z can use a single callback procedure because they depend on each other. If the reset-checkbox next to it is checked, the entry fields below are updated automatically which means the reset button of the second group can be reused. However, that means that the code for that reset button must be defined before the callback procedure for the radio buttons. This is a detail that you can keep in mind but at this stage of planning it's not important.

The procedure needs to determine the direction and the global min and max coordinates that are valid for the second section of the panel.

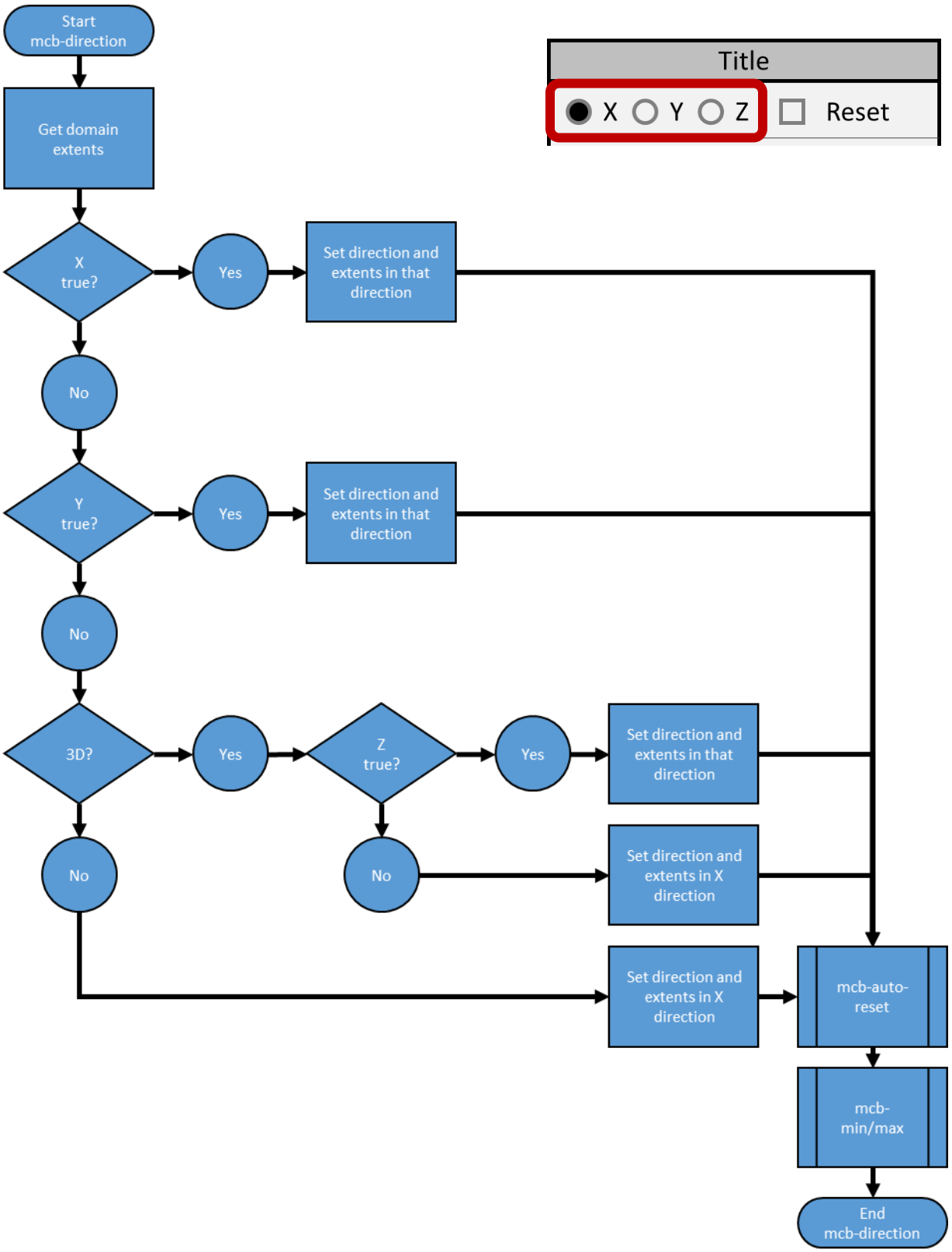


Figure 32: Flow chart of the procedure to evaluate direction selection

The reset checkbox in the first section of the panel can trigger an update of the coordinates in the section below by calling the procedure of the button of the second section.

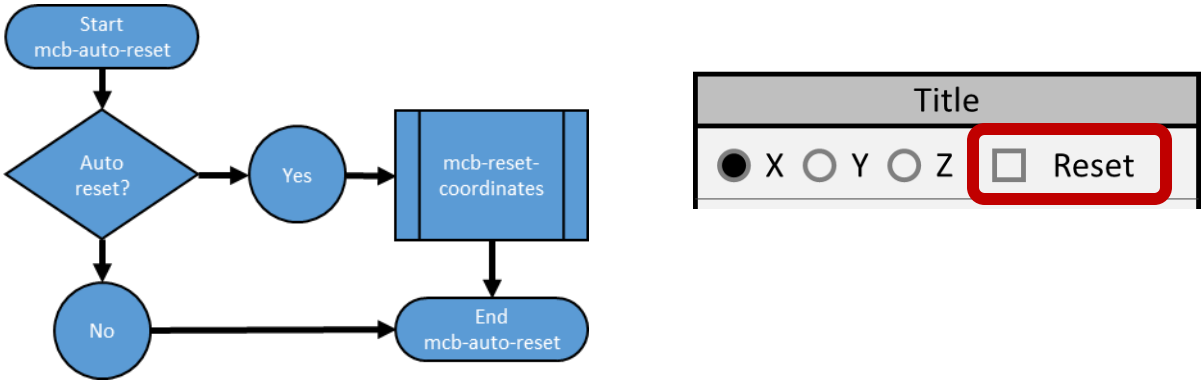


Figure 33: Flow chart of the evaluation of the auto update for the bounds

The min and max real entry fields in the second section require very similar callback procedures but they are not identical. They check if the entry is valid for the chosen direction. It is not possible to limit the input from the beginning because the limits can only be defined when the panel is opened. Afterwards they can't be changed without closing and reopening the panel. This is possible but the effort is usually too high for the small gain of increased user-friendlyness.

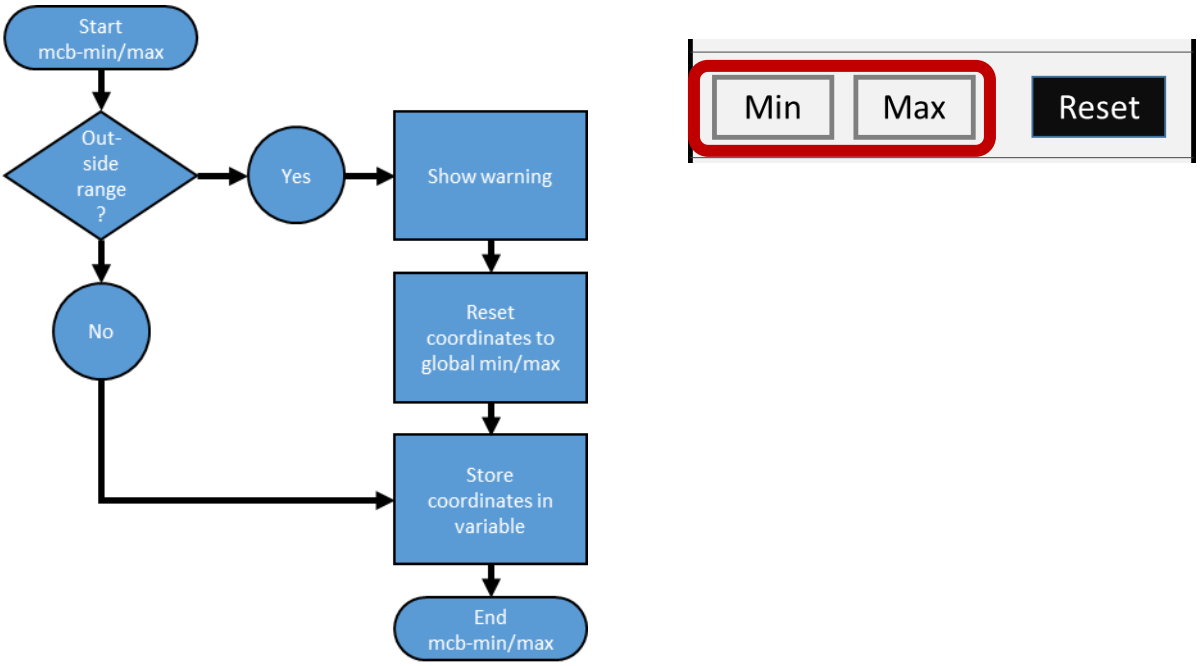


Figure 34: Flow chart for the evaluation of the real entry field for the lower and upper bounds for the surface coordinates

The reset button in the second section of the panel sets min and max values for the surface coordinates to the global domain extents. A fallback method should be included in case there is an invalid value for the direction stored due to a defect in the code. It is also convenient to store the min/max coordinates for the surfaces in their own variables to have better access later, when creating the planes. Strictly speaking this is not required because you can always access the values stored in the input fields.

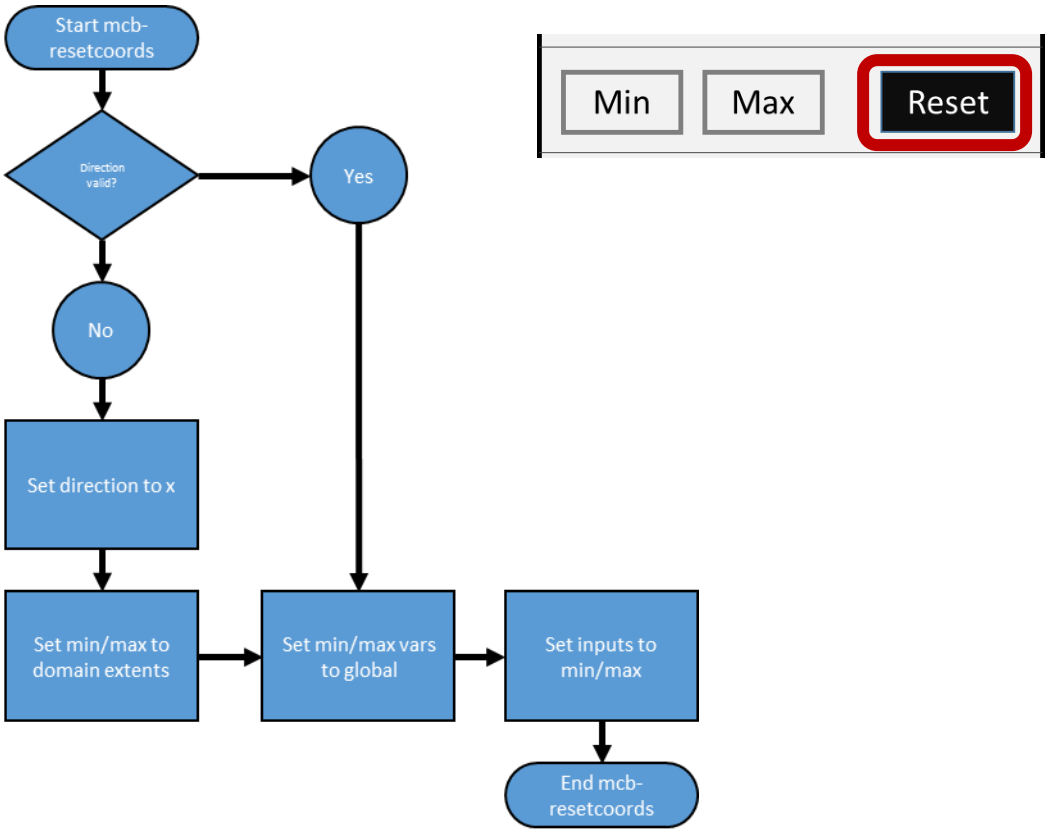


Figure 35: Flow chart for the bounds reset button

For the number of planes the two primitives should update each other. No additional functionality is required.

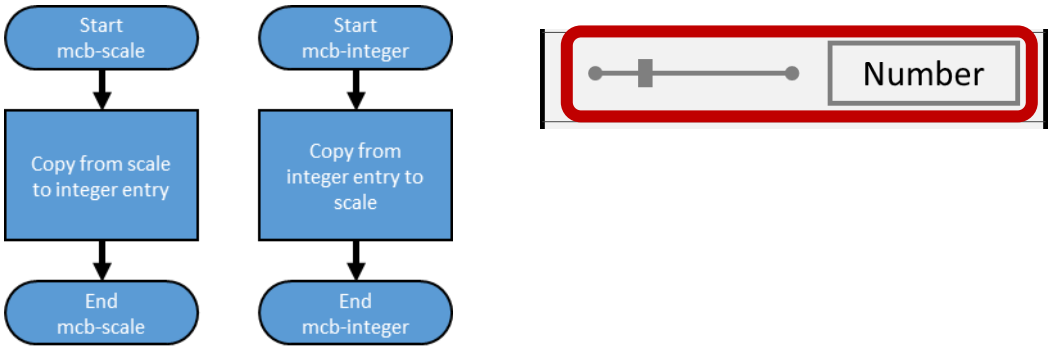


Figure 36: Flow charts for the number selections from a scale and an integer entry object

The text entry field for the surface name prefix should check if the input is empty. If it is, re-populate the field with the default value. Additional checks are not required because the primitive object ensures a valid text input in symbol mode by itself.

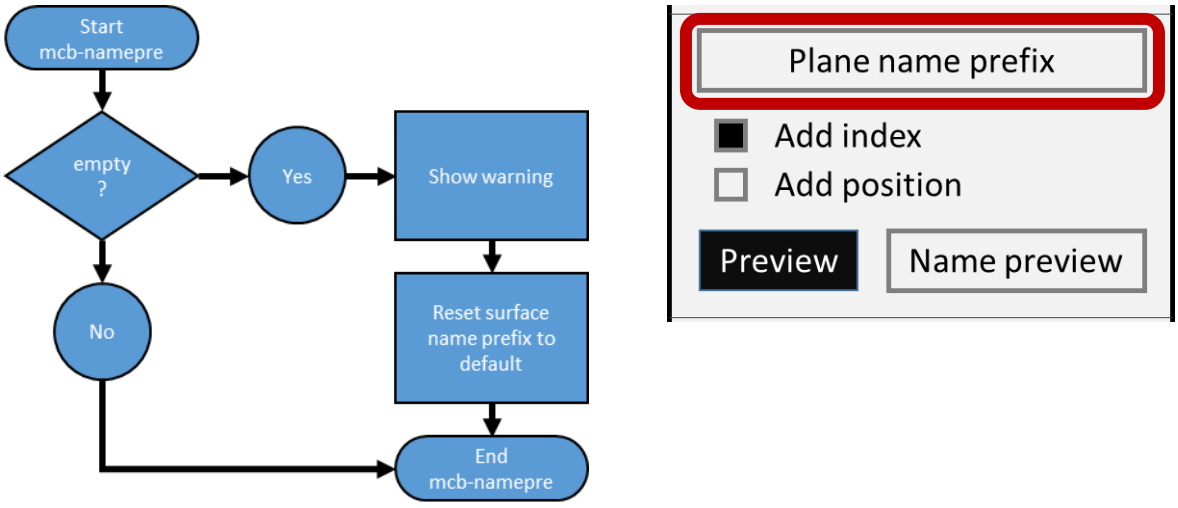


Figure 37: Flow chart to evaluate the surface name prefix

From the two checkboxes, at least one must be active. To implement automatic correction of invalid inputs each checkbox need its own callback procedure. In case both are disabled the other checkbox needs to be enabled. Nothing needs to be done if both or one of them are checked.

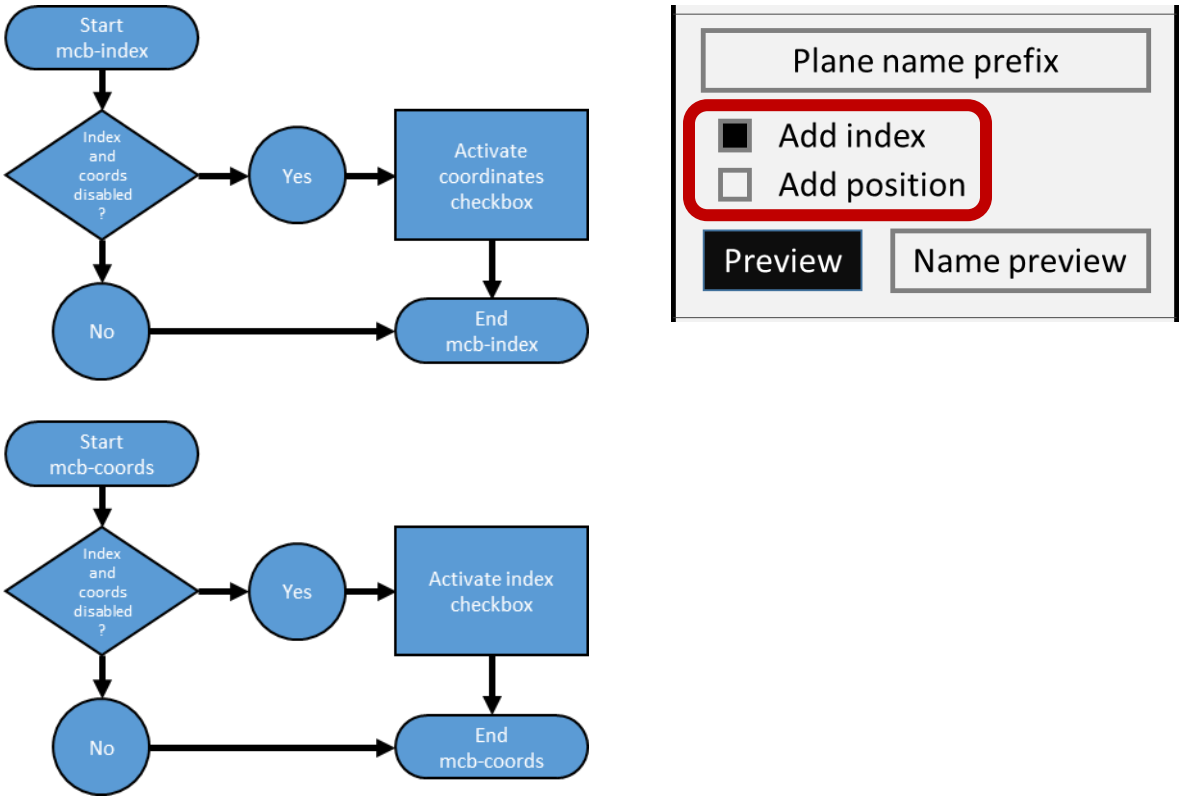


Figure 38: Flow charts for the evaluation of the surface name suffixes

The preview button generates the preview name from the inputs. It has to make sure the text input and the checkboxes are correct in case the callback procedures are not executed automatically.

The surface name itself is created from the name prefix, the optional number of surfaces and the optional center coordinate.

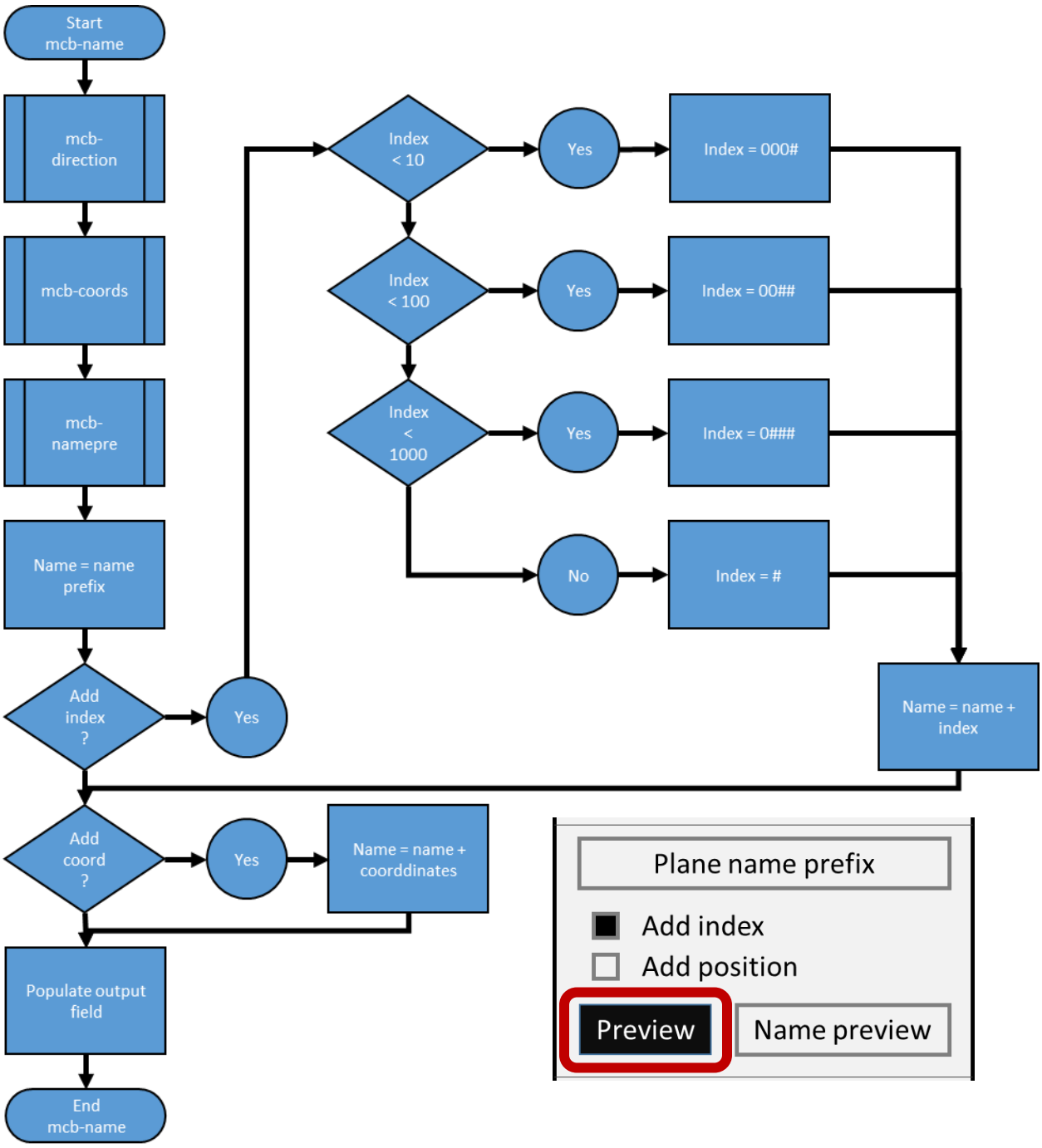


Figure 39: Flow chart to generate a name preview

Now it's possible to plan the reset procedure. Global constants should be defined at the beginning of the whole code, outside of the reset procedure. This is more convenient for adjusting the constants if required.

The first action is to reset the direction, the coordinate-auto-reset checkbox, the number of surfaces the name prefix and the name checkboxes to default values. Then all values can be evaluated by calling the five callback procedures to update the remaining fields.

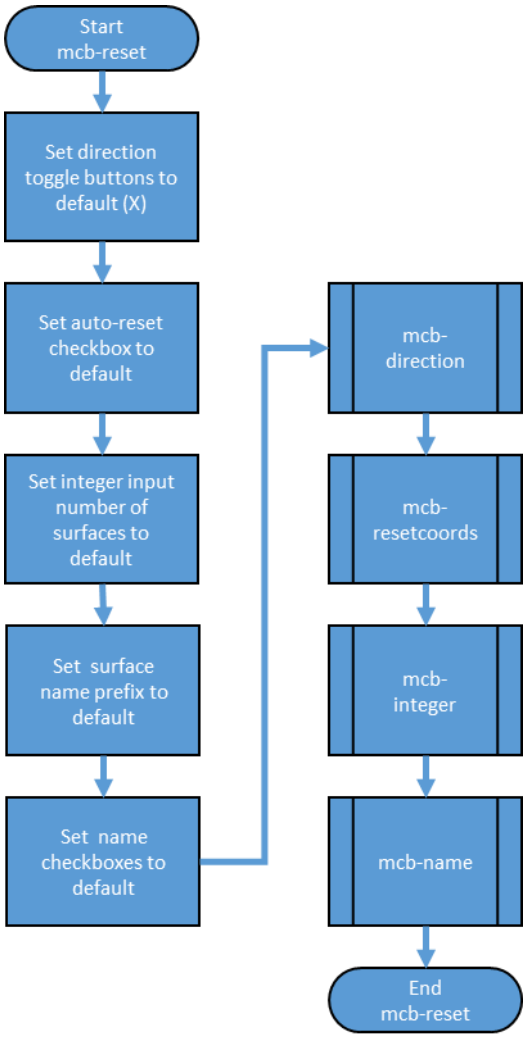


Figure 40: Flow chart to reset all settings to default

The procedure to create the iso-surfaces is only a small part of the complete code. After the declaration of some local variables it's necessary to check all inputs by calling the appropriate callback procedures. Like for the name preview this is required because you can never be sure that the callback procedures of entry fields are executed.

Then it's convenient to store the number of planes and the name prefix in a local variable. Strictly speaking this is not required and a possible source of error. But they are required multiple times in the following code, therefore it's faster to get the values only once.

The direction is evaluated in the next step. Two different strings are derived. One for the optional suffix for the plane name and one for the TUI command that creates an iso-surface.

To calculate the position of each surface the distance between them is calculated from the min and max coordinates and the number of surfaces: $distance = \frac{abs(min)+abs(max)}{number+1}$.

Now it's possible to loop over the number of surfaces, calculate the position, determine the name and create the iso-surfaces. In the end a success message is printed to the console.

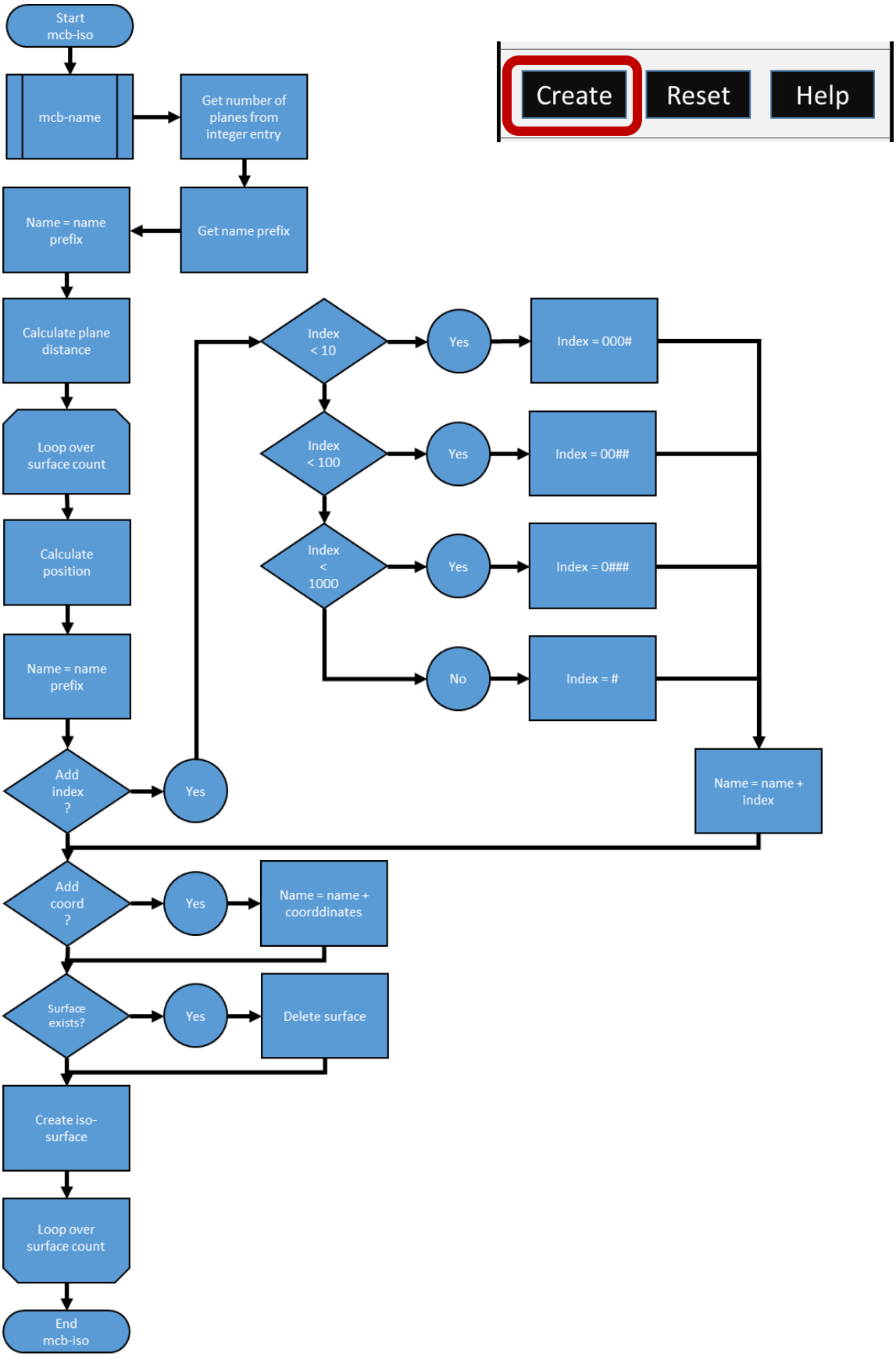


Figure 41: Flow chart to create the user-defined number of iso-surfaces in the selected direction

The last callback procedure is the help text. Essentially, it's just a bunch of display commands, therefore it's not necessary to plan this with a flow chart.

Coding the panel

The basic skeleton of the panel consists of

- the define statement,
- a simple check if data is available,
- a let* statement,
- the declaration of the panel,
- the call to open the panel,
- an error message in case no data is available,
- and the call of the procedure that the panel opens right after loading the Scheme file.

```
001 (define (post-iso-n-panel)
002   (if (data-valid?)
003     (begin
004       (let* (; Create panel
005              (m-pi (cx-create-panel "Create Iso-Surfaces")))
006         ; Definition of panel layout
007         (cx-show-panel m-pi)))
008     (display "\nError! Data not available. Panel not loaded.\nPlease
    initialize the case or read the data. Then type (post-iso-n-panel) in the
    console or read the Scheme file again.\n\n")))
009
010 (post-iso-n-panel)
```

Code fragment 34: Basic code skeleton (01-skeleton.scm)

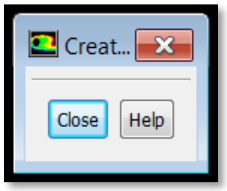


Figure 42: First incarnation of the panel

Since all callback procedures are planned already, you don't need to add unnecessary procedures. However, if you would not have planned it from the beginning, it's good practice to add dummy procedures and fill in the blanks later. That helps that you don't forget to add required callback procedures.

```
001 (define (post-iso-n-panel)
002   (if (data-valid?)
003     (begin
004       (let* (; Callback: Update panel (executed during open)
005              (m-pi-update (lambda ()
006                (display "Panel update procedure m-pi-update not implemented
    yet.\n")))
007              ; Create panel
008              (m-pi (cx-create-panel "Create Iso-Surfaces" 'update-callback
    m-pi-update)))
009         ; Definition of panel layout
010         (cx-show-panel m-pi)))
011     (display "\nError! Data not available. Panel not loaded.\nPlease
    initialize the case or read the data. Then type (post-iso-n-panel) in the
    console or read the Scheme file again.\n\n")))
012
013 (post-iso-n-panel)
```

Code fragment 35: Dummy update callback procedure for initial values (02-update-panel-cb.scm)

Remember that the panel is already planned completely. Now you only must add the required containers and primitives to get as close to the sketch as desired.

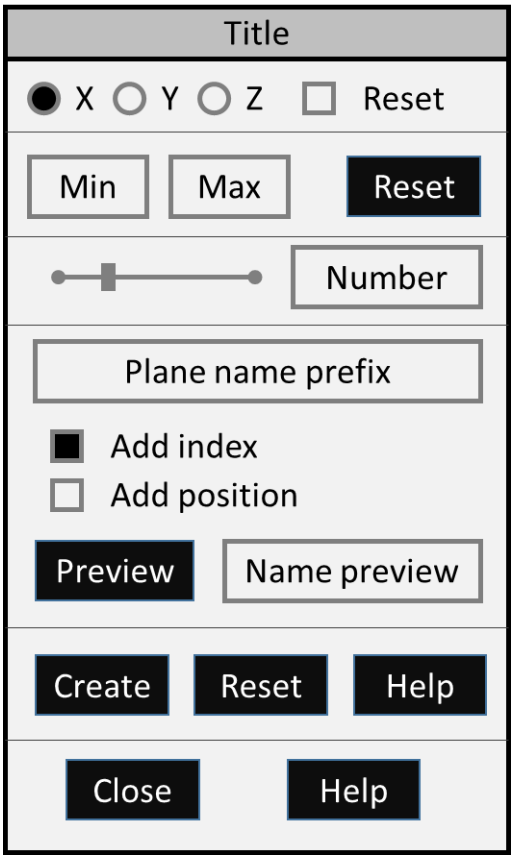


Figure 43: Sketch of the panel design

There are five sections excluding the title and the default buttons. Each of the sections gets its own frame to group the elements together. Remember to add comments to your code to improve readability and understand it even after several months or years.

```
004         (let* (; Conatainer and primitive objects
005                (m-0100-frame-direction)
006                (m-0200-frame-coordinates)
007                (m-0300-frame-number)
008                (m-0400-frame-name)
009                (m-0500-frame-execution)
010                ; Callback: Update panel (executed during open)
...   ...
015           ; Definition of panel layout
016           ; Frame: Direction
017           (set! m-0100-frame-direction (cx-create-frame m-pi "Choose
      direction" 'border #t))
018           ; Frame: Coordinates
019           (set! m-0200-frame-coordinates (cx-create-frame m-pi "Set
      coordinates" 'border #t 'below m-0100-frame-direction))
020           ; Frame: Number
021           (set! m-0300-frame-number (cx-create-frame m-pi "Set number of iso-
      surfaces" 'border #t 'below m-0200-frame-coordinates))
022           ; Frame: Name
023           (set! m-0400-frame-name (cx-create-frame m-pi "Define name pattern"
      'border #t 'below m-0300-frame-number))
024           ; Frame: Execution
025           (set! m-0500-frame-execution (cx-create-frame m-pi "Execute script"
      'below m-0400-frame-name))
026           (cx-show-panel m-pi)))
```

Code fragment 36: Basic frame structure with headlines (03-frames.scm)

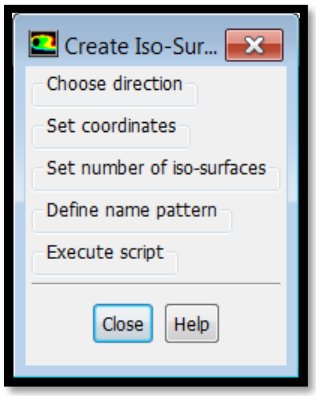


Figure 44: Frame headlines

According to the sketch of the panel the first group contains three radio buttons and a checkbox. The radio buttons must be grouped together in a button box. That's the only possibility to turn checkboxes into radio buttons. Remember that for 2D only two radio buttons are allowed which must be considered in the code.

Figure 45: Sketch of the first section of the panel to choose the direction

There are always three parts of the code that are changed. First comes the declaration of constants, variables, containers and primitives within the declaration part of the let*-statement (lines 6 and 7). Then comes the definition of the callback procedures, also in the declaration parenthesis of the let*-statement but below the primitive declarations (lines 13 to 18). And at the end comes the definition and placement of the objects (lines 27 to 31).

Note: When you are new to coding Fluent panels, do it one line at a time and test your code before continuing. One typo and the panel won't open and the error messages are usually not very descriptive.

```
005              (m-0100-frame-direction)
006              (m-0101-bb-direction) (m-0100-mtb-autoreset)
007              (m-0101-mtb-x) (m-0101-mtb-y) (m-0101-mtb-z)
008              (m-0200-frame-coordinates)
...
011              (m-0500-frame-execution)
012
013              ; Callback: Update direction
014              (mcb-0101-direction (lambda () (display "Callback for
     direction not implemented yet.")))
015
016              ; Callback: Trigger automatic update of coordinates
017              (mcb-0100-autoreset (lambda () (display "Callback for
     automatic update of coordinates not implemented yet.\n")))
018
019              ; Callback: Update panel (executed during open)
...
026          (set! m-0100-frame-direction (cx-create-frame m-pi "Choose
     direction" 'border #t))
027            (set! m-0101-bb-direction (cx-create-button-box m-0100-frame-
     direction "" 'radio-mode #t 'vertical #f 'border #f))
028              (set! m-0101-mtb-x (cx-create-toggle-button m-0101-bb-direction
     "X" 'activate-callback mcb-0101-direction))
029              (set! m-0101-mtb-y (cx-create-toggle-button m-0101-bb-direction
     "Y" 'activate-callback mcb-0101-direction))
030              (if (rp-3d?) (set! m-0101-mtb-z (cx-create-toggle-button m-
     0101-bb-direction "Z" 'activate-callback mcb-0101-direction)))
031            (set! m-0100-mtb-autoreset (cx-create-toggle-button m-0100-frame-
     direction "Auto update domain extents" 'right-of m-0101-bb-direction
     'activate-callback mcb-0100-autoreset))
032          ; Frame: Coordinates
```

Code fragment 37: Cb procedures and buttons for the first section to choose the direction (04-direction.scm)

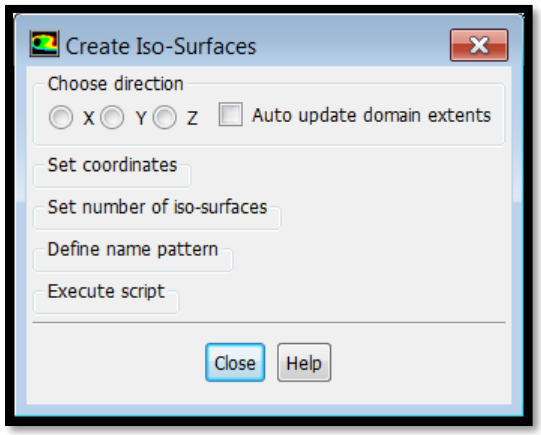


Figure 46: Panel with all primitive objects to select the direction

Notice that none of the radio buttons is active by default. This should be fixed with the panel update callback procedure later just like all other default values.

The second section contains two real entry fields and a button. No additional container is required.

Figure 47: Sketch of the second section to set the bounds for plane creation

```
008             (m-0200-frame-coordinates)
009             (m-0200-re-min) (m-0200-re-max) (m-0200-pb-reset)
010             (m-0300-frame-number)
...
018             (mcb-0100-autoreset (lambda () (display "Callback for
    automatic update of coordinates not implemented yet.\n")))
019
020             ; Callback: Minimum coordinates
021             (mcb-0200-min (lambda () (display "Callback for min coordinate
    validation not implemented yet.\n")))
022
023             ; Callback: Maximum coordinates
024             (mcb-0200-max (lambda () (display "Callback for max coordinate
    validation not implemented yet.\n")))
025
026             ; Callback: Reset coordinates
027             (mcb-0200-reset-coordinates (lambda () (display "Callback to
    reset coordinates to domain extents not implemented yet.\n")))
028
029             ; Callback: Update panel (executed during open)
...
```

```
...
043         (set! m-0200-frame-coordinates (cx-create-frame m-pi "Set
    coordinates" 'border #t 'below m-0100-frame-direction))
044           (set! m-0200-re-min (cx-create-real-entry m-0200-frame-
    coordinates "Minimum" 'activate-callback mcb-0200-min))
045           (set! m-0200-re-max (cx-create-real-entry m-0200-frame-
    coordinates "Maximum" 'activate-callback mcb-0200-max 'right-of m-0200-re-
    min))
046           (set! m-0200-pb-reset (cx-create-button m-0200-frame-coordinates
    "Reset to domain extents" 'activate-callback mcb-0200-reset-coordinates
    'right-of m-0200-re-max))
047         ; Frame: Number
```

Code fragment 38: Global variables, cb procedures and primitive objects to set the surface bounds (05-coordinates.scm)

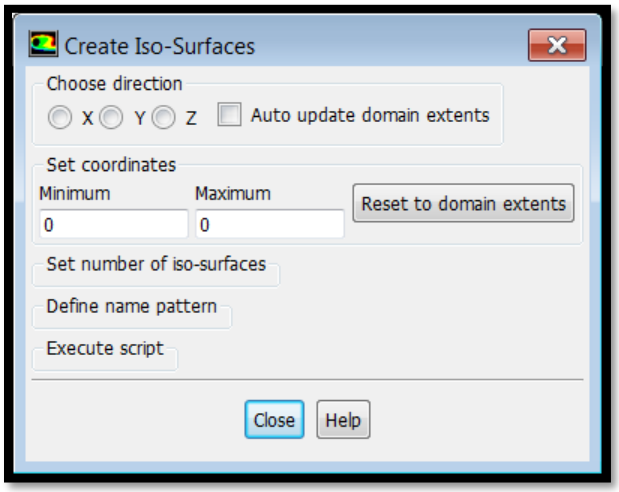


Figure 48: Panel with all primitive objects to set the bounds for surface creation

The third section contains a slider and an integer entry field. Again, no additional container is required for these two primitive objects. However, both inputs need maximum values. It is convenient to define them at the top to change such constants easily if required.

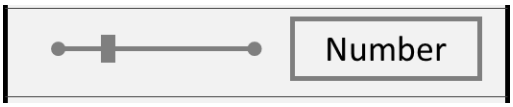


Figure 49: Sketch for the selection of the number of surfaces

The default value of both will also be defined later by the reset and panel update procedures.

```
004       (let* (; Constants and variables
005              (mlc-num-max 100) ; maximum number of surfaces
006              ; Conatainer and primitive objects
...
012              (m-0300-frame-number)
013              (m-0300-s-number) (m-0300-ie-number)
014              (m-0400-frame-name)
...
030              (mcb-0200-reset-coordinates (lambda () (display "Callback to
    reset coordinates to domain extents not implemented yet.\n")))
031
032              ; Callback: Scale update integer
033              (mcb-0300-s-number (lambda () (display "Callback to update
    integer value from scale not implemented yet.\n")))
034
035              ; Callback: Integer update scale
036              (mcb-0300-ie-number (lambda () (display "Callback to update
    scale from integer value not implemented yet.\n")))
037
038              ; Callback: Update panel (executed during open)
...
057         (set! m-0300-frame-number (cx-create-frame m-pi "Set number of iso-
    surfaces" 'border #t 'below m-0200-frame-coordinates))
058           (set! m-0300-s-number (cx-create-scale m-0300-frame-number ""
    'minimum 1 'maximum mlc-num-max 'activate-callback mcb-0300-s-number))
059           (set! m-0300-ie-number (cx-create-integer-entry m-0300-frame-
    number "" 'minimum 1 'maximum mlc-num-max 'activate-callback mcb-0300-ie-
    number 'right-of m-0300-s-number))
060         ; Frame: Name
```

Code fragment 39: Global variables, cb procedures and primitive objects to input the number of iso-surfaces (06-number-surfaces.scm)

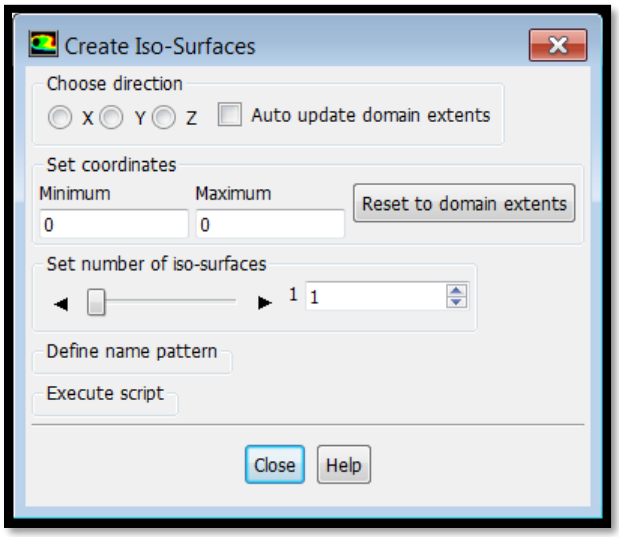


Figure 50: Panel with the inputs to specify the number of surfaces

The fourth section contains all the objects to define the name pattern and to generate a preview. An additional frame is required to group the preview objects together. Otherwise it would not be possible to have two objects below a single checkbox.

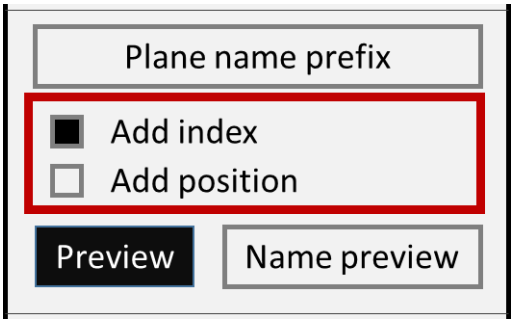


Figure 51: Sketch to choose how the surface name should be derived

```
014                (m-0400-frame-name)
015                (m-0400-te-name) (m-0400-tb-index) (m-0400-tb-coordinates)
016                (m-0401-frame-preview)
017                (m-0401-b-preview-name) (m-0401-te-preview-name)
018                (m-0500-frame-execution)
...
039                (mcb-0300-ie-number (lambda () (display "Callback to update
    scale from integer value not implemented yet.\n")))
040
041                ; Callback: Evaluate surface name prefix
042                (mcb-0400-te-name (lambda () (display "Callback to evaluate
    surface name prefix not implemented yet.\n")))
043
044                ; Callback: Evaluate adding index to surface name
045                (mcb-0400-tb-index (lambda () (display "Callback to evaluate
    index not implemented yet.\n")))
046
047                ; Callback: Evaluate adding coordinates to surface name
048                (mcb-0400-tb-coordinates (lambda () (display "Callback to
    evaluate coordinates not implemented yet.\n")))
049
050                ; Callback: Generate surface name preview
051                (mcb-0401-b-preview-name (lambda () (display "Callback to
    generate surface name preview not implemented yet.\n")))
052
053                ; Callback: Update panel (executed during open)
...
```

```
...
076         (set! m-0400-frame-name (cx-create-frame m-pi "Define name pattern"
    'border #t 'below m-0300-frame-number))
077           (set! m-0400-te-name (cx-create-text-entry m-0400-frame-name
    "Name prefix" 'width 36 'symbol-mode #t 'activate-callback mcb-0400-te-
    name))
078           (set! m-0400-tb-index (cx-create-toggle-button m-0400-frame-name
    "Add index behind name prefix" 'below m-0400-te-name 'activate-callback
    mcb-0400-tb-index))
079           (set! m-0400-tb-coordinates (cx-create-toggle-button m-0400-
    frame-name "Add coordinates behind index or prefix" 'below m-0400-tb-index
    'activate-callback mcb-0400-tb-coordinates))
080           (set! m-0401-frame-preview (cx-create-frame m-0400-frame-name ""
    'border #f))
081             (set! m-0401-b-preview-name (cx-create-button m-0401-frame-
    preview "Generate preview" 'activate-callback mcb-0401-b-preview-name))
082             (set! m-0401-te-preview-name (cx-create-text-entry m-0401-
    frame-preview "" 'editable #f 'width 24 'right-of m-0401-b-preview-name))
083         ; Frame: Execution
```

Code fragment 40: Cb procedures and primitive objects to define how to derive the surface name (07-surface-name.scm)

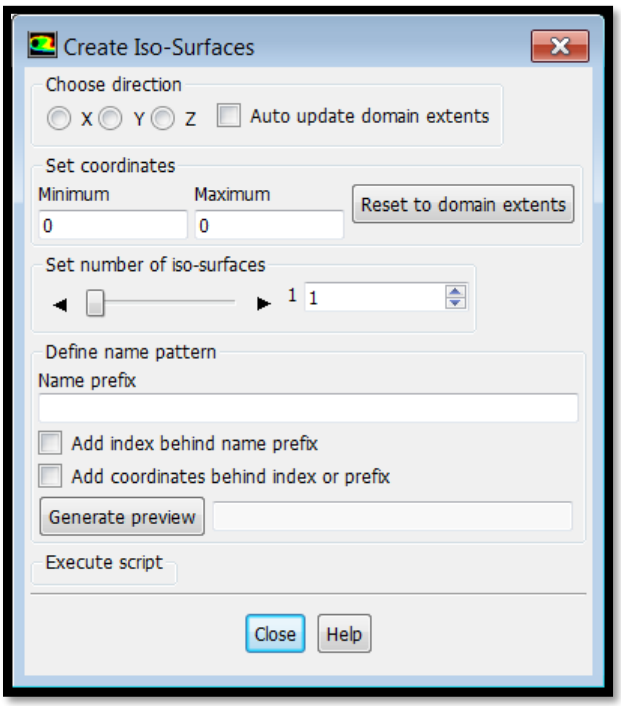


Figure 52: Panel with all primitive objects to define the surface name

The last section of the panel contains three simple buttons.

Figure 53: Sketch for the section to actually create the surfaces or reset to default values

```
018             (m-0500-frame-execution)
019             (m-0500-b-execute) (m-0500-b-reset) (m-0500-b-help)
020
021             ; Callback: Update direction
...
052             (mcb-0401-b-preview-name (lambda () (display "Callback to
    generate surface name preview not implemented yet.\n")))
053
054             ; Callback: Generate surfaces
055             (mcb-0500-b-execute (lambda () (display "Callback to generate
    surfaces not implemented yet.\n")))
056
057             ; Callback: Reset to default values
058             (mcb-0500-b-reset (lambda () (display "Callback to reset
    inputs to default values not implemented yet.\n")))
059
060             ; Callback: Help
061             (mcb-0500-b-help (lambda () (display "Callback to print the
    panel help not implemented yet.\n")))
062
063             ; Callback: Update panel (executed during open)
...
094         (set! m-0500-frame-execution (cx-create-frame m-pi "Execute script"
    'below m-0400-frame-name))
095           (set! m-0500-b-execute (cx-create-button m-0500-frame-execution
    "Create iso-surfaces" 'activate-callback mcb-0500-b-execute))
096           (set! m-0500-b-reset (cx-create-button m-0500-frame-execution
    "Reset to default" 'activate-callback mcb-0500-b-reset 'right-of m-0500-b-
    execute))
097           (set! m-0500-b-help (cx-create-button m-0500-frame-execution
    "Show help" 'activate-callback mcb-0500-b-help 'right-of m-0500-b-reset))
098         (cx-show-panel m-pi)))
```

Code fragment 41: Dummy cb procedures and buttons to create the surfaces, reset the inputs and show a help text (08-execution.scm)

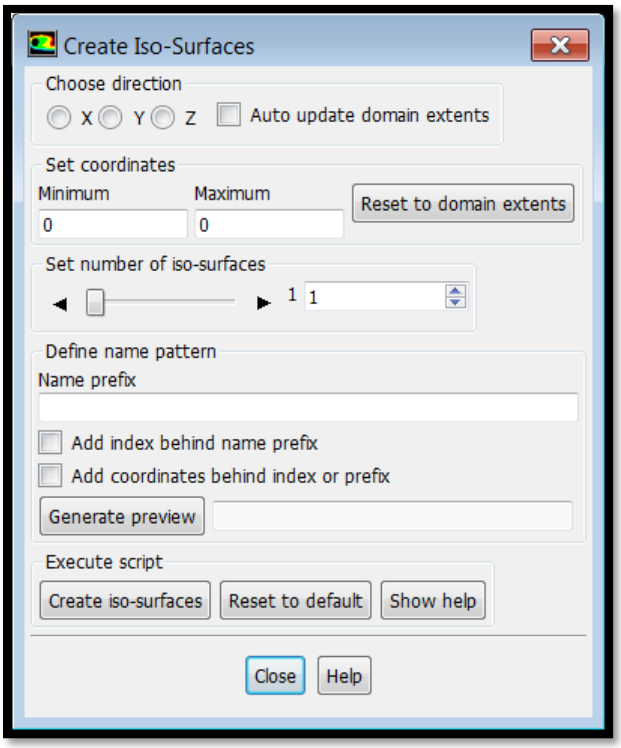


Figure 54: Final panel design

Coding the callback procedures
You should think about the order of the callback procedures before starting to implement them. It's always good to have some structure in your code.

14 callback procedures are planned. Some of them depend on each other which forces a certain order. It can be convenient to visualize the dependencies in tree-diagrams.

From such views, you can easily rearrange the procedures in the code to avoid conflicts. Each procedure must be defined before it is used from another procedure.

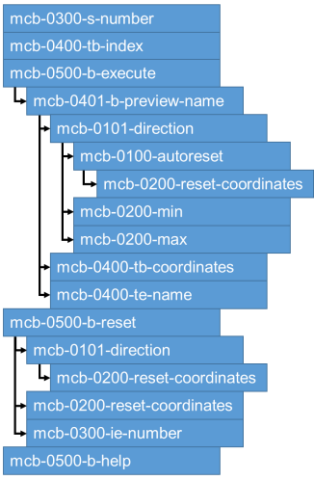


Figure 55: Dependencies of the cb procedures in a tree structure

From this tree structure, you can see that the procedures of the second section are required for the first section. The order must be reversed in the code. The remaining dependencies are consistent with the order in the panel.

```
019             (m-0500-b-execute) (m-0500-b-reset) (m-0500-b-help)
020
021             ; Callback: Minimum coordinates
022             (mcb-0200-min (lambda () (display "Callback for min coordinate
    validation not implemented yet.\n")))
023
024             ; Callback: Maximum coordinates
025             (mcb-0200-max (lambda () (display "Callback for max coordinate
    validation not implemented yet.\n")))
026
027             ; Callback: Reset coordinates
028             (mcb-0200-reset-coordinates (lambda () (display "Callback to
    reset coordinates to domain extents not implemented yet.\n")))
029
030             ; Callback: Trigger automatic update of coordinates
031             (mcb-0100-autoreset (lambda () (display "Callback for
    automatic update of coordinates not implemented yet.\n")))
032
033             ; Callback: Update direction
034             (mcb-0101-direction (lambda () (display "Callback for
    direction not implemented yet.\n")))
035
036             ; Callback: Scale update integer
```

Code fragment 42: Reordering of some dummy cb procedures to align with dependencies (09-reorder-procedures.scm)

Now you can implement the procedures one after another. Since all procedures exist already the order of the implementation is not important. When in doubt, you see the warning messages implemented as feedback that something is still missing. But it can be convenient to use the same order used to line up the procedures.

From the flow chart of `mcb-0200-min` and `mcb-0200-max` (Figure 34; page 50) you see that it starts with an if-condition to check whether the value is out of range. It's convenient to store the required domain extents for this condition in global variables declared at the top of the declaration section of the let*-statement. Remember to add meaningful default values for global variables. In this case you can use the domain extents in x direction (lines 6 and 7 in the code below).

```
005             (mlc-num-max 100) ; maximum number of surfaces
006             (mlv-global-min (list-ref (client-inquire-domain-extents) 0))
007             (mlv-global-max (list-ref (client-inquire-domain-extents) 1))
008             (mlv-min mlv-global-min) (mlv-max mlv-global-max)
009             ; Conatainer and primitive objects
```

Code fragment 43: Global variables to store domain extents in chosen direction and bounds for surface creation

Then you can implement the procedures `mcb-0200-min` and `mcb-0200-max` like sketched in the flow chart:

1. Check if the entered value is smaller than the smallest global coordinate or the largest global coordinate (line 27/38)
   Note that it is necessary to check min and max at the same time for both procedures to avoid invalid input.

2. If outside, display a describing waring message (line 29/40)
3. Set the entry field to the global min (line 31) or the global max (line 42)
4. Outside the if-condition set the global variable to the value in the real entry field (line 33/44)

```
024             ; Callback: Minimum coordinates
025             (mcb-0200-min (lambda ()
026               ; Check if min coordinates are outside of valid range
027               (if (or (< (cx-show-real-entry m-0200-re-min) mlv-global-
    min) (> (cx-show-real-entry m-0200-re-min) mlv-global-max))
028                 (begin
029                   (display (format #f "\n  Warning! Minimum coordinate
    outside of domain. Resetting to domain min: ~a.\n" mlv-global-min))
030                   ; set min value to global min
031                   (cx-set-real-entry m-0200-re-min mlv-global-min)))
032               ; Store real entry value in variable
033               (set! mlv-min (cx-show-real-entry m-0200-re-min))))
034
035             ; Callback: Maximum coordinates
036             (mcb-0200-max (lambda ()
037               ; Check if max coordinates are outside of valid range
038               (if (or (> (cx-show-real-entry m-0200-re-max) mlv-global-
    max) (< (cx-show-real-entry m-0200-re-max) mlv-global-min))
039                 (begin
040                   (display (format #f "\n  Warning! Maximum coordinate
    outside of domain. Resetting to domain max: ~a.\n" mlv-global-max))
041                   ; set max value to global max
042                   (cx-set-real-entry m-0200-re-max mlv-global-max)))
043               ; Store real entry value in variable
044               (set! mlv-max (cx-show-real-entry m-0200-re-max))))
045
046             ; Callback: Reset coordinates
```

Code fragment 44: Cb procedures for real entry fields of surface bounds (10-0200-minmax.scm)

To test these procedures, you can load an arbitrary 2D or 3D case in Fluent, initialize it and load the panel. Then check the domain extents in X direction and enter valid and invalid values. Remember that the callback procedures are only called when you push the return or enter keys.

Note that it doesn't check which of the two inputs are larger. This might be important at a later stage and was not included during the planning phase. If you notice possible error sources like min coordinate > max coordinate either fix it right away or take a note.

Fixing it right away can be inconvenient because a user might enter the values in a different order. Resetting to default values all the time can be disturbing. Therefore, it's better to keep a note of this error source and revisit this note in every other procedure to see if the possible error source can be eradicated there.

Resetting the coordinates is planned with Figure 35; page 51. The direction itself is evaluated in procedure `mcb-0101-direction` and stored in a global variable because it is required by several other procedures. This global variable doesn't exist yet and should be initialized with the x direction.

Usually it is convenient to use numbers instead of strings for such variables. Strings can be difficult to handle. In this case a string reduces the need for additional variables later on because it can be used directly to derive the surface name and inside the TUI command to generate the iso-surfaces.

```
008             (mlv-min mlv-global-min) (mlv-max mlv-global-max)
009             (mlv-direction "x")
010             ; Conatainer and primitive objects
```

Code fragment 45: Global variable to store the direction

The if-condition (line 49) can be strange when you are not used to combining several logical checks. Essentially, you need to check if the contents of the global variable that holds the direction is invalid. This check is more complicated for a string than for a number.

The condition checks if the value is valid and negates the result. That you can read it more easily the statement is broken into several lines. In the attached Scheme files everything is in a single line. Both versions work.

The rest of the procedure is just setting some variables like outlined in the flow chart.

```
047             ; Callback: Reset coordinates
048             (mcb-0200-reset-coordinates (lambda ()
049               (if (not (and (string? mlv-direction)
                                (or (string=? mlv-direction "x")
                                    (string=? mlv-direction "y")
                                    (and (rp-3d?)
                                         (string=? mlv-direction "z")
                                    )
                                )
                           ))
050                  (begin
051                    (display "\n Warning! Direction is invalid. Resetting to
    X.\n")
052                    (set! mlv-direction "x")
053                    (set! mlv-global-min (list-ref (client-inquire-domain-
    extents) 0))
054                    (set! mlv-global-max (list-ref (client-inquire-domain-
    extents) 1))
055                    (cx-set-toggle-button m-0101-mtb-x #t)
056                    (cx-set-toggle-button m-0101-mtb-y #f)
057                    (if (rp-3d?) (cx-set-toggle-button m-0101-mtb-z #f))))
058                (set! mlv-min mlv-global-min)
059                (set! mlv-max mlv-global-max)
060                (cx-set-real-entry m-0200-re-min mlv-min)
061                (cx-set-real-entry m-0200-re-max mlv-max)))
062
063             ; Callback: Trigger automatic update of coordinates
```

Code fragment 46: Resetting surface bounds to domain extents (11-0200-resetcoordinates.scm)

Line 49: If-condition to check if global variable mlv-direction is invalid

Line 50-57: Display warning and reset the direction in case the direction value is invalid. The reset includes a reset of the toggle buttons in the first section of the panel. This code should never be executed if everything works as intended.

Line 58-61: Set the global variables and the real entry fields to the stored domain extents.

To test this code, you can use different values for the variable in line 9. Don't forget setting it back to zero when your tests are complete.

Note that the warning message of line 51 was not planned when developing the flow charts but it is good practice to have some sort of output when fallback measures kick in. Hopefully, this is never executed

once the script is complete. Nevertheless, it's important to check if the values stored in variables are valid to make the code robust. In a perfect world you could just delete lines 49 to 57.

The checkbox "auto update domain extents" (Figure 33; page 50) evaluates itself and, if true, it executes `mcb-0200-reset-coordinates`.

```
063               ; Callback: Trigger automatic update of coordinates
064               (mcb-0100-autoreset (lambda ()
065                 (if (cx-show-toggle-button m-0100-mtb-autoreset)
066                   (mcb-0200-reset-coordinates))))
067
068               ; Callback: Update direction
```

Code fragment 47: Cb procedure to evaluate auto-update checkbox in the direction section (12-0100-autoreset.scm)

When looking at the flow chart for `mcb-0101-direction` (Figure 32; page 49) you can see two boxes called "Set direction and extents in X direction". Something similar also exists in the procedure `mcb-0200-reset-coordinates`. In fact, according to the flow charts the code from line 51 to 57 is required three times. You should avoid duplicate code. Therefore, it makes sense to write a new procedure for this part of the code before continuing. While at it, you can also use a global variable to define the default direction which makes it easier to adjust it if desired. Unfortunately, this makes this code quite long to evaluate this default direction.

Remember that all of that is only a fallback mechanism in case something goes wrong with the panel. This code should never be executed when everything works as expected.

```
005               (mlc-num-max 100) ; maximum number of surfaces
006               (mlc-default-direction "x") ; only x and y are valid for all
    cases
007               (mlv-global-min (list-ref (client-inquire-domain-extents) 0))
...
046                 (set! mlv-max (cx-show-real-entry m-0200-re-max))))
047
048               ; Reset direction to default
049               (mlp-reset-direction (lambda ()
050                 (display (format #f "\n Warning! Direction is invalid.
    Resetting to ~a.\n" mlc-default-direction))
051                 (set! mlv-direction mlc-default-direction)
052                 (if (and (string? mlv-direction) (string=? mlv-direction
    "x"))
053                   (begin ; default: x direction
054                     (set! mlv-global-min (list-ref (client-inquire-domain-
    extents) 0))
055                     (set! mlv-global-max (list-ref (client-inquire-domain-
    extents) 1))
056                     (cx-set-toggle-button m-0101-mtb-x #t)
057                     (cx-set-toggle-button m-0101-mtb-y #f)
058                     (if (rp-3d?) (cx-set-toggle-button m-0101-mtb-z #f)))
```

```
059                   (if (and (string? mlv-direction) (string=? mlv-direction
    "y"))
060                     (begin ; default: y direction
061                       (set! mlv-global-min (list-ref (client-inquire-domain-
    extents) 2))
062                       (set! mlv-global-max (list-ref (client-inquire-domain-
    extents) 3))
063                       (cx-set-toggle-button m-0101-mtb-x #f)
064                       (cx-set-toggle-button m-0101-mtb-y #t)
065                       (if (rp-3d?) (cx-set-toggle-button m-0101-mtb-z #f)))
066                     (if (and (string? mlv-direction) (and (rp-3d?) (string=?
    mlv-direction "z")))
067                       (begin ; default: z direction and 3D
068                         (set! mlv-global-min (list-ref (client-inquire-
    domain-extents) 4))
069                         (set! mlv-global-max (list-ref (client-inquire-
    domain-extents) 5))
070                         (cx-set-toggle-button m-0101-mtb-x #f)
071                         (cx-set-toggle-button m-0101-mtb-y #f)
072                         (cx-set-toggle-button m-0101-mtb-z #t))
073                       (begin ; invalid default direction or z and 2D
074                         (display (format #f "\n ERROR!\n Default direction
    invalid! Resetting to x.\n"))
075                         (set! mlv-global-min (list-ref (client-inquire-
    domain-extents) 0))
076                         (set! mlv-global-max (list-ref (client-inquire-
    domain-extents) 1))
077                         (cx-set-toggle-button m-0101-mtb-x #t)
078                         (cx-set-toggle-button m-0101-mtb-y #f)
079                         (if (rp-3d?) (cx-set-toggle-button m-0101-mtb-z
    #f))))))))
080
081               ; Callback: Reset coordinates
082               (mcb-0200-reset-coordinates (lambda ()
083                 (if (not (and (string? mlv-direction) (or (string=? mlv-
    direction "x") (string=? mlv-direction "y") (and (rp-3d?) (string=? mlv-
    direction "z")))))
084                     (mlp-reset-direction))
085                 (set! mlv-min mlv-global-min)
086                 (set! mlv-max mlv-global-max)
087                 (cx-set-real-entry m-0200-re-min mlv-min)
088                 (cx-set-real-entry m-0200-re-max mlv-max)))
```

Code fragment 48: Correction for mlp-reset-direction to use a procedure as fallback for invalid direction values

The largest part of the direction selection is the evaluation of the toggle buttons with a couple of nested if-statements (lines 52, 59, 66and 73). It would be possible to use a cond-statement instead of nested ifs. But in case the panel doesn't report back a meaningful selection of toggle buttons it could lead to problems with cryptic error messages especially in 2D.

Essentially the content of each if-statement is identical.

1. Set the domain extents in the global variable (lines 54/55, 61/62, 68/69 and 75/76)
2. Set the toggle buttons to the correct values which has to be done for all buttons (lines 56-58, 63-65, 70-72 and 77-79)

Just the else part for an invalid direction definition has an additional error message (line 74).

The cb procedure for the reset coordinates button calls this procedure only if the direction is invalid (lines 83 to 84).

The cb procedure for all three toggle buttons is identical. After checking which of the toggle buttons is active (lines 97, 102, 107/108, 113 and 115) and storing direction (lines 99, 104 and 110) and domain extents in global variables (lines 100-101, 105-106 and 111-112), you can evaluate the checkbox for automatically update the dimensions in the second section of the panel (line 118). Finally, you can check if the values are within the valid range by calling the callback procedures of the real entry fields (lines 120/121).

Note that checking the coordinates here can be inconvenient when a user selects a direction by accident. It's up to you if you want to check it here or in the name preview which is also called by the surface creation procedure.

```
095             ; Callback: Update direction
096             (mcb-0101-direction (lambda ()
097               (if (cx-show-toggle-button m-0101-mtb-x)
098                 (begin ; x-direction
099                   (set! mlv-direction "x")
100                   (set! mlv-global-min (list-ref (client-inquire-domain-
    extents) 0))
101                   (set! mlv-global-max (list-ref (client-inquire-domain-
    extents) 1)))
102                 (if (cx-show-toggle-button m-0101-mtb-y)
103                   (begin ; y-direction
104                     (set! mlv-direction "y")
105                     (set! mlv-global-min (list-ref (client-inquire-domain-
    extents) 2))
106                     (set! mlv-global-max (list-ref (client-inquire-domain-
    extents) 3)))
107                   (if (rp-3d?)
108                     (if (cx-show-toggle-button m-0101-mtb-z)
109                       (begin ; z-direction only in 3D
110                         (set! mlv-direction "z")
111                         (set! mlv-global-min (list-ref (client-inquire-
    domain-extents) 4))
112                         (set! mlv-global-max (list-ref (client-inquire-
    domain-extents) 5)))
113                       (begin ; invalid return values in 3D
114                         (mlp-reset-direction)))
115                     (begin ; invalid return values in 2D
116                       (mlp-reset-direction)))))
117               ; Reset coordinate selections in section 2 automatically?
118               (mcb-0100-autoreset)
119               ; Check if coordinate selection in section 2 is valid for
    the chosen direction
120               (mcb-0200-min)
121               (mcb-0200-max)))
122
123             ; Callback: Scale update integer
```

Code fragment 49: Evaluation of the direction radio buttons (13-0101-direction.scm)

There is a little bit of redundancy in this code. Lines 120 and 121 set the bounds for the coordinates of the iso-surfaces in the global variables. This is also done indirectly from the procedure called in line 118

(compare lines 35, 46 and 85 to 88). However, this is only executed if the auto-reset checkbox is active. This can be avoided by evaluating the checkbox directly in `mcb-0101-direction`. This would also eliminate the dependency of `mcb-0101-direction` and `mcb-0100-autoreset` but not to `mcb-0200-reset-coordinates`.

```
117                 ; Reset coordinate selections in section 2 automatically?
118                 (if (cx-show-toggle-button m-0100-mtb-autoreset)
119                   (mcb-0200-reset-coordinates)
120                   (begin ; no reset of coordinates, check for valid inputs
121                     (mcb-0200-min)
122                     (mcb-0200-max)))))
123
124               ; Callback: Scale update integer
```

Code fragment 50: Alternate version to evaluate auto-reset and valid coordinates in mcb-0101-direction (14-0101-direction-alternative.scm)

Either way, the result is the same. The second approach requires a little bit less CPU time in case the checkbox is active.

The cb procedures for the primitives in the third section are trivial again because they only update each other (Figure 3651; page 51).

```
124               ; Callback: Scale update integer
125               (mcb-0300-s-number (lambda ()
126                 ; Copy value of scale to integer field
127                 (cx-set-integer-entry m-0300-ie-number (cx-show-scale m-
    0300-s-number))))
128
129               ; Callback: Integer update scale
130               (mcb-0300-ie-number (lambda ()
131                 ; Copy value of integer field to scale
132                 (cx-set-scale m-0300-s-number (cx-show-integer-entry m-0300-
    ie-number))))
133
134               ; Callback: Evaluate surface name prefix
```

Code fragment 51: Cb procedures for scale and integer primitives to update each other (15-0300-scale_integer.scm)

The flow chart to evaluate the surface name prefix shows that a default value is required (Figure 37; page 52). This means you need to introduce another global variable. Like before, it's best to define these default values at the top that they can be changed easily if required.

```
006               (mlc-default-direction "x") ; only x and y are valid for all
    cases
007               (mlc-default-name "_iso") ; default surface name prefix
008               (mlv-global-min (list-ref (client-inquire-domain-extents) 0))
```

Code fragment 52: Global variable for the default surface name prefix

The rest of the implementation is trivial again. Most of the checks are done by the text entry field itself because it's set to symbol mode. This makes it impossible to enter invalid characters that can't be used by Fluent.

```
135             ; Callback: Evaluate surface name prefix
136             (mcb-0400-te-name (lambda ()
137               ; Check if plane name is empty, other invalid characters are
    excluded by the text entry field itself
138               (if (string=? (cx-show-text-entry m-0400-te-name) "")
139                 (begin
140                   (display (format #f "Warning. Plane name empty. Reset to
    default '~a'.\n" mlc-default-name))
141                   ; Reset to default name
142                   (cx-set-text-entry m-0400-te-name mlc-default-name)))))
143
144             ; Callback: Evaluate adding index to surface name
```

Code fragment 53: Check for empty surface name prefix (16-0400-name-prefix.scm)

Evaluating the index and the coordinates for the name is also straight forward. You can use a simple if condition to check if both toggle buttons are disabled. If this is true, activate the other one (Figure 38; page 52).

```
144             ; Callback: Evaluate adding index to surface name
145             (mcb-0400-tb-index (lambda ()
146               ; Check if both buttons are disabled and report warning
147               (if (and (not (cx-show-toggle-button m-0400-tb-index)) (not
    (cx-show-toggle-button m-0400-tb-coordinates)))
148                 (cx-set-toggle-button m-0400-tb-coordinates #t))))
149
150             ; Callback: Evaluate adding coordinates to surface name
151             (mcb-0400-tb-coordinates (lambda ()
152               ; Check if both buttons are disabled and report warning
153               (if (and (not (cx-show-toggle-button m-0400-tb-index)) (not
    (cx-show-toggle-button m-0400-tb-coordinates)))
154                 (cx-set-toggle-button m-0400-tb-index #t))))
155
156             ; Callback: Generate surface name preview
```

Code fragment 54: Evaluating the surface name suffix checkboxes (17-0400-name-sufix.scm)

If you compare the flow charts to generate the preview name (Figure 39; page 53) and to create the iso-surfaces (Figure 41; page 55), you see that there is a large portion that is identical again. Instead of writing duplicate code you can move it into its own procedure.

That it works for both instances, the coordinates and the index must be passed to the procedure. For the preview of the surface name any value could be used but the iso-surfaces are created from within a loop where index and coordinates are changing.

It is not possible to test the code of this sub-procedure by itself. Therefore, it's good practice to implement a dummy procedure, first (lines 158/159). This can be called from the preview name button (line 166) and allows you to run tests while you code.

```
155                 (cx-set-toggle-button m-0400-tb-index #t))))
156
157             ; Procedure to generate the name from provided index and
    coordinates
158             (mlp-name (lambda (mlpv-index mlpv-coords)
159             ))
160
161             ; Callback: Generate surface name preview
162             (mcb-0401-b-preview-name (lambda ()
163               (mcb-0101-direction)
164               (mcb-0400-tb-coordinates)
165               (mcb-0400-te-name)
166               (cx-set-text-entry m-0401-te-preview-name (mlp-name (cx-
    show-integer-entry m-0300-ie-number) (/ (+ mlv-min mlv-max) 2)))
167             ))
168
169             ; Callback: Generate surfaces
```

Code fragment 55: Initial implementation of the cb procedure to generate the surface name preview

Line 166 might need some explanations. It assigns the return value of the sub-procedure `mlp-name` to the text entry field `m-0401-te-preview-name`. This field is used only for output. `mlp-name` is called with two arguments (see line 158, two are required). The first argument is the number of iso-surfaces. They are grabbed from the integer entry field `m-0300-ie-number`. The second argument is the average of the min and max coordinates.

If this line is too confusing for you, you can split it up. Use a let-statement in line 161 to define a number of local variables that hold name, index and coordinates. This might improve the readability of line 165. This step is not used for this example, here.

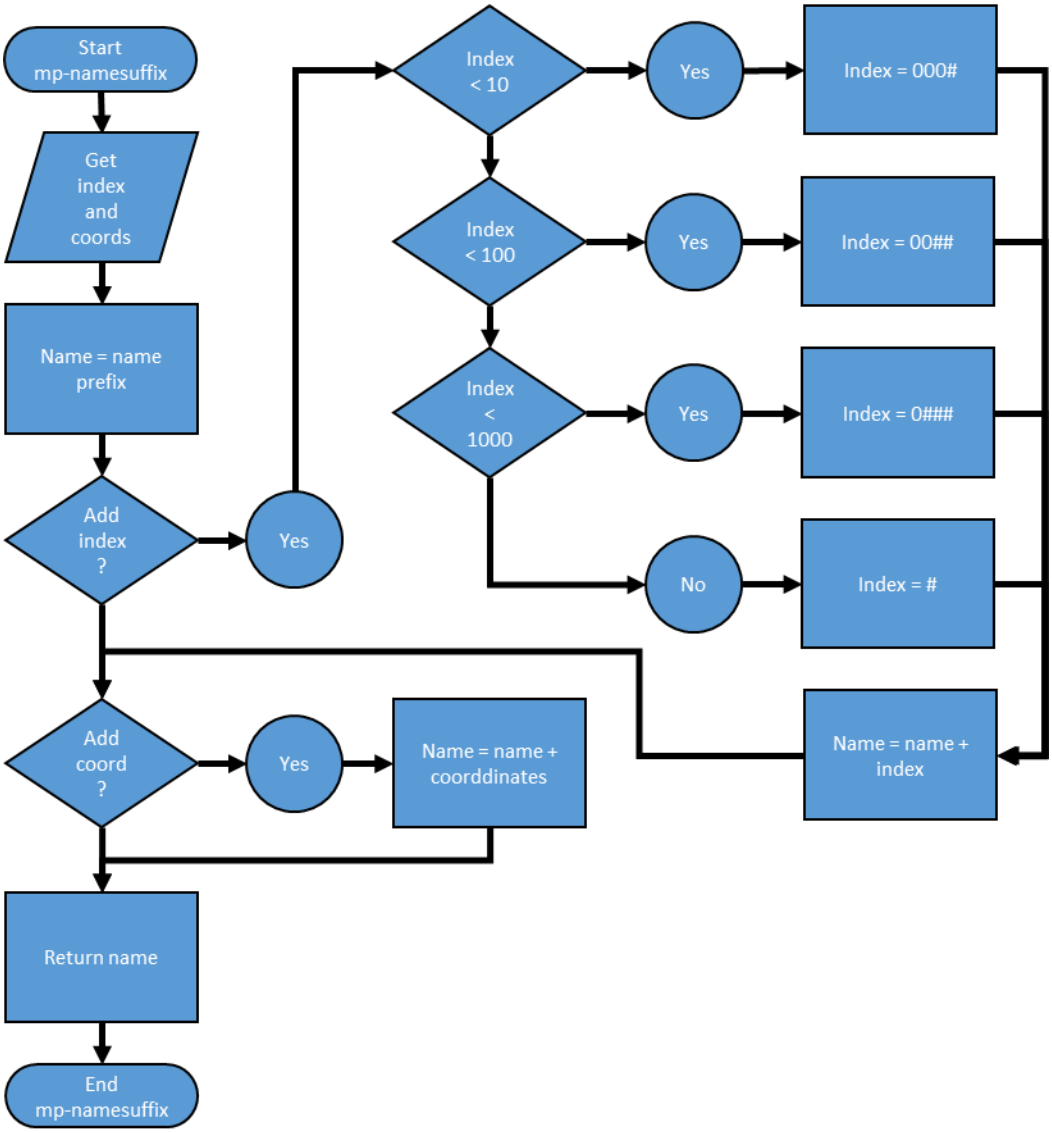


Figure 56: Flow chart of a sub-procedure to generate the name of iso-surfaces

Now you can implement the missing part of evaluating the inputs and create the name for an iso-surface. You can use a cond-statement to add the leading zeros for the index. This makes it more convenient for sorting in Fluent.

```
157             ; Procedure to generate the name from provided index and
    coordinates
158             (mlp-name (lambda (mlpv-index mlpv-coords)
159               (let ((mlpv-name (cx-show-text-entry m-0400-te-name)))
160                 (if (cx-show-toggle-button m-0400-tb-index)
161                   (cond
162                     ((< mlpv-index 10) (set! mlpv-name (format #f
    "~a~a000~a" mlpv-name mlc-separation-char mlpv-index)))
163                     ((< mlpv-index 100) (set! mlpv-name (format #f
    "~a~a00~a" mlpv-name mlc-separation-char mlpv-index)))
164                     ((< mlpv-index 1000) (set! mlpv-name (format #f
    "~a~a0~a" mlpv-name mlc-separation-char mlpv-index)))
165                     (else (set! mlpv-name (format #f "~a~a~a" mlpv-name
    mlc-separation-char mlpv-index)))))
166                 (if (cx-show-toggle-button m-0400-tb-coordinates)
167                   (set! mlpv-name (format #f "~a~a~a=~a" mlpv-name mlc-
    separation-char mlv-direction mlpv-coords)))
168                 mlpv-name)))
169
170             ; Callback: Generate surface name preview
```

Code fragment 56: Implementation of the name generation sub procedure

When you test this code you might notice that there is something missing. Remember the note you took in the beginning that the values of the coordinates are not checked?

Actually, for the preview this is no problem but this is a good place to fix the issue. Min and max coordinates should satisfy three conditions:

1. Both values should be within the computational domain.
2. The min value should be smaller than the max value.
3. Both values should not be identical.

The first condition is ensured by the procedure `mcb-0101-direction` already. But the other two are not satisfied yet.

You can add this in a new procedure that you call from `mcb-0401-b-preview-name` or add it after line 174. Since it is only required once, you can add it to the existing procedure but this might make code adjustments more difficult in the future.

Always remember to update the flow charts if required. This can make it easier for you or another person to check the code for errors.

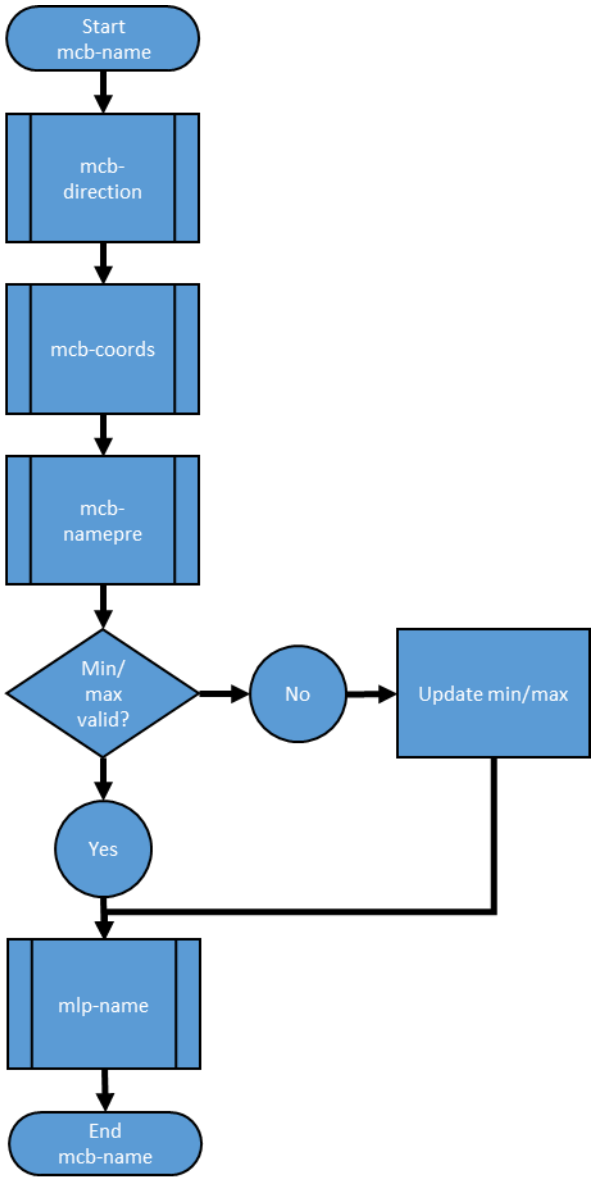


Figure 57: Updated flow chart for mcb-0401-b-preview-name

```
170             ; Callback: Generate surface name preview
171             (mcb-0401-b-preview-name (lambda ()
172               (let ((mlpv-tmp))
173                 (mcb-0101-direction)
174                 (mcb-0400-tb-coordinates)
175                 (mcb-0400-te-name)
176                 ; Check if min/max are valid
177                 (if (> mlv-min mlv-max)
178                   (begin ; min > max -> reverse
179                     (set! mlpv-tmp mlv-min)
180                     (set! mlv-min mlv-max)
181                     (set! mlv-max mlpv-tmp)
182                     (cx-set-real-entry m-0200-re-min mlv-min)
183                     (cx-set-real-entry m-0200-re-max mlv-max)))
184                 (if (eqv? mlv-min mlv-max)
185                   (begin ; min = max -> reset
                      (display (format #f "\n Warning. Iso-surface bounds
186 are identical.\n Resetting to domain extents.\n"))
187                     (mcb-0200-reset-coordinates)))
188                 (cx-set-text-entry m-0401-te-preview-name (mlp-name (cx-
    show-integer-entry m-0300-ie-number) (/ (+ mlv-min mlv-max) 2))))))
```

Code fragment 57: Final implementation of the preview name cb procedure (18-0401-name-preview.scm)

Only three procedures are left. It's better to implement the reset procedure before the actual generation of the iso-surfaces. Let's check the flow chart (Figure 40; page 54) to see which procedures you can reuse.

Setting the direction toggle buttons to default exists already in the sub-procedure `mlp-reset-direction`. Since the default is now defined by a variable rather than a fixed value it would be best to split this procedure into two. The first one simply resets to default. The second one shows an error and calls the first one. `mcb-0500-reset` can then reuse the first one.

The auto-reset checkbox in the first section doesn't have a default value yet. This should be defined at the top that it can be changed easily. The same applies to the integer input and the surface name suffixes checkboxes.

When all these default values exist, you can assign them to the primitive objects and call the callback procedures that are already in the flow chart to update the rest.

Start with the global constants.

```
004       (let* (; Constants
005              (mlc-num-max 100) ; maximum number of surfaces
006              (mlc-default-direction "x") ; only x and y are valid for all
    cases
007              (mlc-default-name "_iso") ; default surface name prefix
008              (mlc-separation-char "_") ; character to separate surface name
    prefix from suffixes
009              (mlc-default-auto-reset #f) ; Automatically reset the
    coordinates when clicking on a different direction?
010              (mlc-default-number-surfaces 10) ; Default number of iso-
    surfaces to be created
011                ; At least one of the following two should be #t
012              (mlc-default-index #t) ; Attach the number to the name prefix?
013              (mlc-default-coordinates #f) ; Attach the coordinate to the
    name prefix?
014              ; Variables
015              (mlv-global-min (list-ref (client-inquire-domain-extents) 0))
```

Code fragment 58: Definition of default values as constants

Then split `mlp-reset-direction` into two procedures.

```
056              ; Reset direction to default
057              (mlp-reset-direction-without-message (lambda ()
058                (set! mlv-direction mlc-default-direction)
059                (if (and (string? mlv-direction) (string=? mlv-direction
    "x"))
060                  (begin ; default: x direction
061                    (set! mlv-global-min (list-ref (client-inquire-domain-
    extents) 0))
062                    (set! mlv-global-max (list-ref (client-inquire-domain-
    extents) 1))
063                    (cx-set-toggle-button m-0101-mtb-x #t)
064                    (cx-set-toggle-button m-0101-mtb-y #f)
065                    (if (rp-3d?) (cx-set-toggle-button m-0101-mtb-z #f)))
066                  (if (and (string? mlv-direction) (string=? mlv-direction
    "y"))
067                    (begin ; default: y direction
068                      (set! mlv-global-min (list-ref (client-inquire-domain-
    extents) 2))
069                      (set! mlv-global-max (list-ref (client-inquire-domain-
    extents) 3))
070                      (cx-set-toggle-button m-0101-mtb-x #f)
071                      (cx-set-toggle-button m-0101-mtb-y #t)
072                      (if (rp-3d?) (cx-set-toggle-button m-0101-mtb-z #f)))
073                    (if (and (string? mlv-direction) (and (rp-3d?) (string=?
    mlv-direction "z")))
074                      (begin ; default: z direction and 3D
075                        (set! mlv-global-min (list-ref (client-inquire-
    domain-extents) 4))
076                        (set! mlv-global-max (list-ref (client-inquire-
    domain-extents) 5))
077                        (cx-set-toggle-button m-0101-mtb-x #f)
078                        (cx-set-toggle-button m-0101-mtb-y #f)
079                        (cx-set-toggle-button m-0101-mtb-z #t))
```

```
080                       (begin ; invalid default direction or z and 2D
081                         (display (format #f "\n ERROR!\n Default direction
    invalid! Resetting to x.\n"))
082                         (set! mlv-global-min (list-ref (client-inquire-
    domain-extents) 0))
083                         (set! mlv-global-max (list-ref (client-inquire-
    domain-extents) 1))
084                         (cx-set-toggle-button m-0101-mtb-x #t)
085                         (cx-set-toggle-button m-0101-mtb-y #f)
086                         (if (rp-3d?) (cx-set-toggle-button m-0101-mtb-z
    #f))))))))
087
088               (mlp-reset-direction (lambda ()
089                 (display (format #f "\n Warning! Direction is invalid.
    Resetting to ~a.\n" mlc-default-direction))
090                 (mlp-reset-direction-without-message)))
091
092               ; Callback: Reset coordinates
```

Code fragment 59: Splitting direction reset procedure to reuse it in the global reset procedure without error message

Then implement the global reset procedure more or less according to the flow chart.

```
202               ; Callback: Reset to default values
201               (mcb-0500-b-reset (lambda ()
202                 (mlp-reset-direction-without-message)
203                 (cx-set-toggle-button m-0100-mtb-autoreset mlc-default-auto-
    reset)
204                 (cx-set-integer-entry m-0300-ie-number mlc-default-number-
    surfaces)
205                 (cx-set-text-entry m-0400-te-name mlc-default-name)
206                 (cx-set-toggle-button m-0400-tb-index mlc-default-index)
207                 (cx-set-toggle-button m-0400-tb-coordinates mlc-default-
    coordinates)
208                 (mcb-0101-direction)
209                 (mcb-0200-reset-coordinates)
210                 (mcb-0300-ie-number)
211                 (mcb-0401-b-preview-name)))
212
213               ; Callback: Help
```

Code fragment 60: Implementation of the global reset procedure according to flow chart

When you test this code you might notice that under some circumstances warning messages appear. They are shown by `mcb-0101-direction` which calls `mcb-0200-min` and `mcb-0200-max`. Fortunately, when looking at the code, this procedure is no longer required to reset the direction. `mlp-reset-direction-without-message` is doing that already which means that line 208 can be deleted.

Don’t forget to update the flow chart. Although only two boxes are affected it is an important modification of the code.

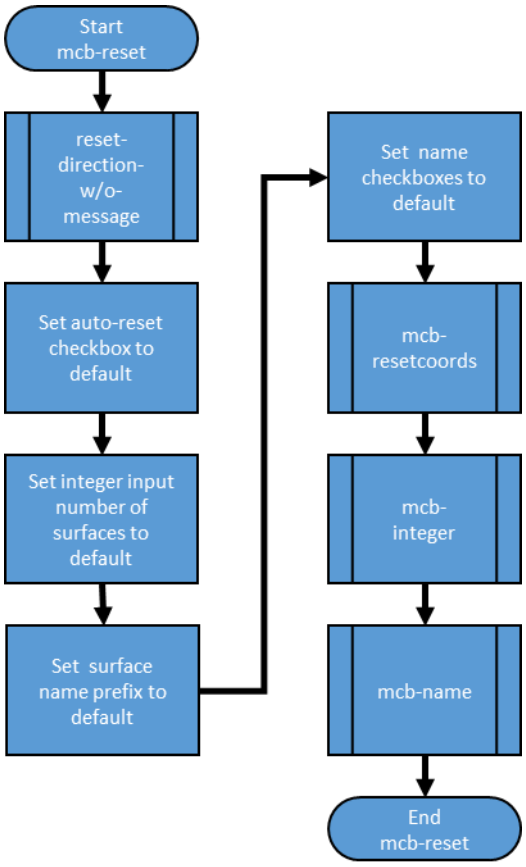


Figure 58: Updated flow chart for mcb-0500-reset

Finally, change the update cb procedure for the panel to call the reset procedure. The dummy implementation in lines 218 to 220 can be deleted.

```
202                ; Callback: Reset to default values and update panel (executed
    during open)
201                (mcb-0500-b-reset (lambda ()
202                  (mlp-reset-direction-without-message)
203                  (cx-set-toggle-button m-0100-mtb-autoreset mlc-default-auto-
    reset)
204                  (cx-set-integer-entry m-0300-ie-number mlc-default-number-
    surfaces)
205                  (cx-set-text-entry m-0400-te-name mlc-default-name)
206                  (cx-set-toggle-button m-0400-tb-index mlc-default-index)
207                  (cx-set-toggle-button m-0400-tb-coordinates mlc-default-
    coordinates)
208                  (mcb-0200-reset-coordinates)
209                  (mcb-0300-ie-number)
210                  (mcb-0401-b-preview-name)))
211
```

```
212             ; Callback: Help
213             (mcb-0500-b-help (lambda () (display "Callback to print the
    panel help not implemented yet.\n")))
214
215             ; Create panel
216             (m-pi (cx-create-panel "Create Iso-Surfaces" 'update-callback
    mcb-0500-b-reset)))
217         ; Definition of panel layout
```

Code fragment 61: Final implementation of the reset procedure also as panel update (19-0500-reset-scm)

Finally, you can implement the main procedure. The changes implemented over the last pages require an update of the initial flow chart.

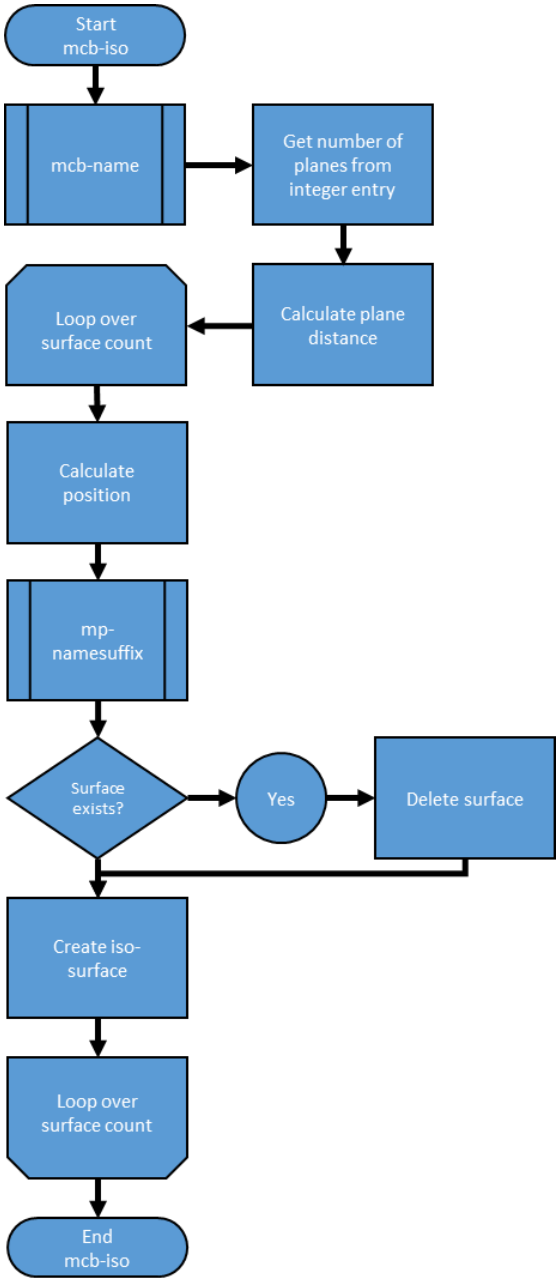


Figure 59: Updated flow chart to create the iso-surfaces

```
001 ; Execute text commands without console output
002 (define (m-quiet command)
003   (let ((psst))
004     (set! psst (with-output-to-string
005       (lambda ()
006         (ti-menu-load-string command))))))
007
008 (define (post-iso-n-panel)
...
206               ; Callback: Generate surfaces
207               (mcb-0500-b-execute (lambda ()
208                 (let ((mlpv-name) (mlpv-pos) (mlpv-distance) (mlpv-number-
    planes) (i))
209                   ; Check inputs
210                   (mcb-0401-b-preview-name)
211                   ; Get number of planes and name prefix
212                   (set! mlpv-number-planes (cx-show-integer-entry m-0300-ie-
    number))
213                   ; Calculate distance between planes
214                   (set! mlpv-distance (/ (+ (abs mlv-min) (abs mlv-max)) (+
    mlpv-number-planes 1)))
215                   ; Loop to create planes
216                   (do ((i 1 (+ i 1)))
217                     ((> i mlpv-number-planes) (display (format #f "\nCreated
    ~a iso-surfaces along ~a-direction.\n" mlpv-number-planes mlv-direction)))
218                     (begin
219                       (set! mlpv-pos (+ mlv-min (* i mlpv-distance)))
220                       (set! mlpv-name (mlp-name i mlpv-pos))
221                       (if (member (string->symbol mlpv-name) (inquire-
    surface-names))
222                         (delete-surfaces (list (surface-name->id (string-
    >symbol mlpv-name)))))
223                       (m-quiet (format #f "/surface/iso-surface ~a-
    coordinate ~a () () ~a ()\n" mlv-direction mlpv-name mlpv-pos)))))))
224
225               ; Callback: Maximum coordinates
```

Code fragment 62: Evaluate all inputs and create iso-surfaces (20-0500- execute.scm)

The procedure in lines 1-6 is used to suppress the output of the TUI command in line 223. This is discussed in part 2 of this document series [2].

The first thing in `mcb-0500-b-execute` is the definition of some local variables for the name of the surface, the position, the distance between two iso-surfaces, the total number of planes and a counter (line 208).

The input values are all checked in `mcb-0401-b-preview-name`, so this procedure can be reused (line 210). This will also update the surface name preview in the panel to correspond to the current settings.

Then you can store the total number of planes in the local variable defined earlier (line 212). Strictly speaking this is not required because you could also access the integer entry directly in the three occurrences in lines 214 and 217. Using a variable might improve readability of the code for such a case.

The distance between the iso-surfaces can be calculated from the min and max coordinates and from the total number of planes (line 214).

There are a couple of recursion methods available in Scheme. In this case you can use a do loop that runs from 1 to the maximum number of planes. If the counter variable i is larger than that maximum number, you can print a success message. Note that this particular implementation just prints a message but it doesn't check if the planes were really created successfully.

Inside the loop you calculate the current position (line 219), derive the name of the current plane (line 220) with the help of the sub-procedure `mlp-name` and check if the surface exists already (lines 221/222). If it exists, that surface is deleted before creating the iso-surface with the calculated values (line 223).

That's all that is required to generate a number of iso-surfaces in a robust way. Most possible invalid inputs are either captured by the panel objects themselves or the individual callback procedures.

Now that everything important is implemented you should test this code thoroughly. Make sure all the buttons work as expected and all invalid inputs are caught. If that's the case, you can implement the final cb procedure that just prints out a help text to the Fluent console. This explains how to use the panel and which inputs are valid.

```
237             ; Callback: Help
238             (mcb-0500-b-help (lambda ()
239               (newline)
240               (display "Help text for panel \"Create Iso-Surfaces\"\n\n")
241               (display "With this panel you can distribute a number of
    post-processing surfaces across the domain. The\nsurfaces are parallel to
    one of the Cartesian planes.\n\n")
242               (display (format #f "The maximum number of planes that you
    can create with the panel is set to ~a. You can adjust this in\nthe code.
    Search for \"(mlc-num-max\" and adjust the integer value behind it. You can
    also adjust\nthe default number \"(mlc-default-number-surfaces\" which is
    set to ~a and the default name\n\"(mlc-default-name\" which is currently
    \"~a\".\n\n" mlc-num-max mlc-default-number-surfaces mlc-default-name))
243               (display "The panel has five sections:\nChoose
    direction\nSet coordinates\nSet number of iso-surfaces\nDefine name
    pattern\nExecute script\n\n")
244               (display "Choose direction\n")
245               (display "Select one of the available directions. The panel
    detects if the case is 2D or 3D and adjusts the\noptions accordingly.\nIf
    the coordinates in the next section become invalid they update
    automatically to the domain extents.\n\n")
246               (display "Set coordinates\n")
247               (display "Select the coordinates between which the iso-
    surfaces should be distributed. They will be created\nbetween the specified
    boundaries. No planes are created exactly on these coordinates. You can
    use\nthe button \"Reset to domain extents\" to populate both fields with
    the extents of your simulation\ndomain. If you enter values outside of your
    domain, the invalid value will be reset to the domain\nextents during the
    creation of the iso-surfaces.\n")
248               (display "Set number of iso-surfaces\n")
249               (display "Select the number of planes either with the slider
    or by typing in a number in the field below the\nslider. Although both
    depend on each other, it is possible to provoke a mismatch by entering
    a\nnumber in the field and then clicking outside instead of pressing return
    on your keyboard. In the case\nof a mismatch, the value in the integer box
    is dominant.\n\n")
```

```
250                 (display "Define name pattern\n")
251                 (display "Define the base name of the planes. If you leave
    this field empty it will be populated again with the\ndefault name during
    the creation of the planes. The text entry field prevents you from
    specifying an\ninvalid name (e.g. starting with a number or using
    space).\n")
252                 (display "The two options below the base name allow you to
    add an index and/or the coordinates to the\nname. At least one of both has
    to be active. You can't deactivate both.\n")
253                 (display "You can also use the button \"Generate preview\"
    to check all fields of the panel and print the name to\nthe field right of
    the button.\n\n")
254                 (display "Execute script\n")
255                 (display "Use the button \"Create iso-surfaces\" to create
    the iso-surfaces with the specified options.\n")
256                 (display "\"Reset to default\" returns all inputs to the
    default values.\n")
257                 (display "\"Show help\" prints this help text to the Fluent
    console.\n")
258                 (display "\"Close\" closes the window without any additional
    action. The settings are not preserved.\n")
259                 (display "\"Help\" opens the ANSYS Help system with an error
    message. The help for this panel can only be\nprinted with the button
    \"Show help\".\n\n")))
256
257             ; Create panel
```

Code fragment 63: Help text to complete the last cb procedure for the iso-surfaces panel (21-0500-help.scm)

Putting everything together

To get easy access to all your scripts you can add everything into the .fluent file. But this can be difficult to maintain.

Instead, you can create a new folder in your home directory where you put all your scripts. Then you can add references to these file in the .fluent file as menu structure. .fluent is always located in the home directory. For this example, you can place the two scripts for the surface integral and the iso-surfaces panel in the folder fluent_scripts.

To access this folder, you can store the path in a variable:

```
(define my_script_dir (string-append (getenv "home") "/fluent_scripts/"))
```

Then you can load a script with:

```
(load (string-append my_script_dir "iso-surfaces-panel.scm"))
```

Remember that for Fluent it doesn't matter if you use a forward or a backward slash between folder names. For everything you add in your scripts you should use the Linux symbol: /

For Windows systems the resulting string might look a bit strange with a mixture of forward and backward slashes but Fluent can access the correct files and folders nonetheless.

Now that you know how to load Scheme scripts from a certain folder you can write the .fluent file and place it in your home directory.

Start with checking the scripts. They should define the procedures but they don't need to execute them. The file `07-si-temp-secondary-phase.scm` can be used without modifications for now. `21-0500-help.scm` opens the panel in the last line (line 296). Simply remove this line and save both scripts in the folder

fluent_scripts in your home folder. You can use describing names like `si-temp-secondary-phase.scm` and `iso-surfaces-panel.scm`.

Then add the first lines to your .fluent file.

```
001 (define my_script_dir (string-append (getenv "home") "/fluent_scripts/"))
002
003 (load (string-append my_script_dir "si-temp-secondary-phase.scm"))
004 (load (string-append my_script_dir "iso-surfaces-panel.scm"))
```

Code fragment 64: First segment of the .fluent file which loads scripts from other files

When you start a new Fluent session you can call both procedures from the TUI in the usual manner. But you can make your life easier by defining keyboard shortcuts and menu items.

Keyboard shortcuts can be defined easily once you have the scripts loaded.

```
006 (set! *cx-key-map*
007   (append *cx-key-map*
008     '(
009       ("control shift i" . "(post-iso-n-panel)")
010       ("control shift m" . "(my-surface-integral-temperature-secondary-
    phase)")))))
```

Code fragment 65: Second segment of the .fluent file to define keyboard shortcuts for the loaded procedures

Again, in a new Fluent session you can call the procedures easily as long as you know the keyboard shortcuts and a graphics window is active.

For the menu items you can increase the robustness by checking if the procedure symbols are known before adding their menu items.

```
012 (define menu-userscripts (cx-add-menu "User scripts" #f))
013 (define menu-userscripts-post (cx-add-hitem menu-userscripts "Post-
    Processing" #f))
014 (if (symbol-bound? 'my-surface-integral-temperature-secondary-phase (the-
    environment))
015   (define menu-userscripts-post-temperature (cx-add-item menu-userscripts-
    post "Temperature Second Phase (CTRL+SHIFT+M)" #f #f #t my-surface-
    integral-temperature-secondary-phase)))
016 (if (symbol-bound? 'post-iso-n-panel (the-environment))
017   (define menu-userscripts-post-surfaces (cx-add-item menu-userscripts-post
    "Create Iso-Surfaces (CTRL+SHIFT+I)" #f #f #t post-iso-n-panel)))
```

Code fragment 66: Last segment of the .fluent file to define menu items to access the loaded procedures

These are all required steps to make scripts available for all Fluent sessions starting with a certain home directory. If you're using multiple systems like a local Windows workstation and remote Linux visualization nodes you have to add folders and the .fluent file to each home directory to get the same access everywhere.

Summary

Scheme offers you a versatile toolset to automate tasks in ANSYS Fluent. You can define procedures and add them to your .fluent file to have easy access to features you need all the time. You can also add menu items, create your own panels and even create user-define keyboard shortcuts to execute your scripts quickly.

After working through this and the previous two documents you should have all the basics to develop your own scripts with user-friendly panels. Furthermore, you should be able to understand most of the examples that are available on the ANSYS Customer Portal. Search the solutions for the keyword “scm” and filter the results by Fluent to get relevant search results.

The huge downside of Scheme scripting is the lack of documentation and support. **If you have trouble with your Fluent simulations, rename the .fluent file and try to reproduce the issues without using scripts. Contact the ANSYS technical support only when you are sure that the problem exists without using Scheme at all.**

If you need a reliable customization or scripting solution for your projects for a specific Fluent version, you can contact the ANSYS consulting team to discuss your needs and get an individual offer.

This document is provided for your convenience as self-help content. No support is provided for its content. The examples were tested with Fluent 18.0. But ANSYS cannot guarantee that everything works with this or other releases.

References

[1] ANSYS, Inc., „Solution 2042681: Introduction to ANSYS Fluent scripting - Part 1 - Introduction to Scheme,“ ANSYS, Inc., 2016.

[2] ANSYS, Inc., „Solution 2042682: Introduction to ANSYS Fluent scripting - Part 2 - Accessing solver data,“ ANSYS, Inc., 2016.

[3] ANSYS, Inc., "ANSYS Fluent Customization Manual, Release 17.1," ANSYS, Inc., 2016.

[4] ANSYS, Inc., „ANSYS Fluent User's Guide, Release 17.1,“ ANSYS, Inc., Canonsburg, PA, 2016.

[5] ANSYS, Inc., „Solution 2042772: How to create many equally spaced (iso-) surfaces for ANSYS Fluent post-processing?,“ ANSYS, Inc., 2016.

Attachments

All scripts discussed in this document are available for download in a single zip file. They are organized in folders:

01_recording:	Example of a recorded journal
02_keyboard_shortcuts:	Example for keyboard shortcut definitions
03_menu:	Example to create menu items
04_panel_basics:	Small examples for the different panel primitive objects
05_iso-surfaces_flow_charts:	Flow charts for the planning of coding the creation of a user-defined number of iso-surfaces
06_iso-surfaces:	Scheme scripts for the panel and the creation of a user-defined number of iso-surfaces
07_summary:	Combining all previous examples in a .fluent file for easy access

Keywords: Best-Practice; best practices; guidelines; guideline; Scheme; Script; Scripting; ANSYS Fluent; Journal; Journaling; TUI; Text User Interface; Automation; keyboard shortcut; hotkey; hotkeys; panel; panels; window; windows; menu; menu item; menu items; submenu; submenus

