RunPluginEntry_cu and Parent-Child Plugins

Bob Zawalich July 31, 2021

You can read about the design of Parent/Child plugins in the document **Parent-Child Plugins**.

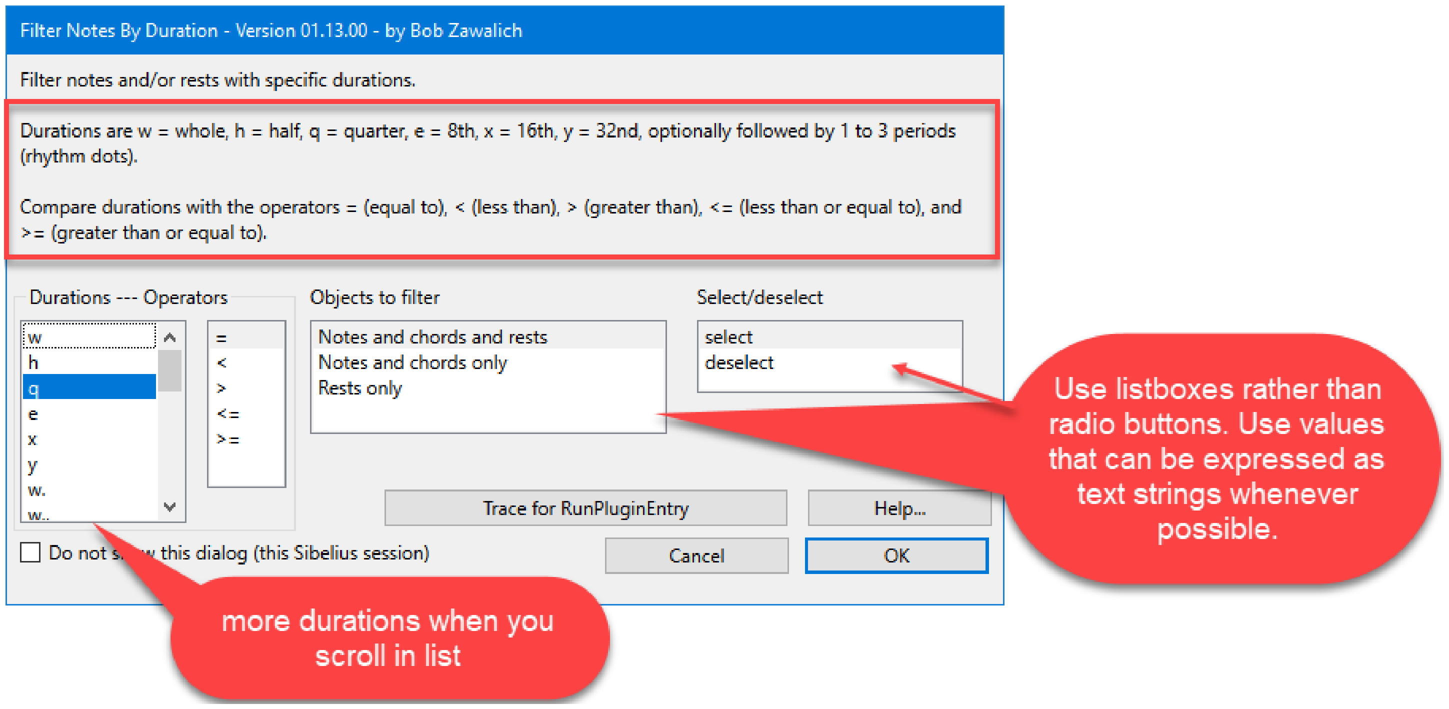
In a nutshell, a **Parent** plugin is one set up so that its dialog settings are collected into a data structure (a Dictionary), and then its main processing routine is called using only the settings in the Dictionary. A **Child** plugin can then create a Dictionary, fill it with appropriate vales, and then call the Parent's main processing routine, knowing that all the data required is contained in the Dictionary, and no dialog is needed.

This document describes the use of the **cmdutils** command **RunPluginEntry_cu** as a universal child plugin for appropriately designed **Parent** plugins.

Using the RunPluginEntry_cu command in Execute Commands/cmdutils.

A **Parent** plugin is an excellent target for commands in cmdutils.plg. The plugin can contain many options in the main dialog, but cmdutils can call a single feature without showing the dialog.

Here is the dialog for **Filter Notes By Durations**, which is set up to be called this way.



Filter Notes By Duration - Version 01.13.00 - by Bob Zawalich

Filter notes and/or rests with specific durations.

Durations are w = whole, h = half, q = quarter, e = 8th, x = 16th, y = 32nd, optionally followed by 1 to 3 periods (rhythm dots).

Compare durations with the operators = (equal to), < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to).

Durations --- Operators
w
h
q
e
x
y
w.
w..
=
<
>
<=
>=
Objects to filter
Notes and chords and rests
Notes and chords only
Rests only
Select/deselect
select
deselect
Trace for RunPluginEntry
Help...
Do not show this dialog (this Sibelius session)
Cancel
OK
Use listboxes rather than radio buttons. Use values that can be expressed as text strings whenever possible.
more durations when you scroll in list

The dictionary passed to the routine **API_ProcessSelection** has text strings for the 4 list boxes, plus entries for score and selection. In the trace below, the score and selection field values do not appear in the trace, but the normal text fields do.

```
TraceDictDialogProperties: tracing property dictionary.
index: 0, name: score, value:
index: 1, name: selection, value:
index: 2, name: str_Duration, value: q
```

```
index: 3, name: str_Operator, value: =
index: 4, name: str_Type, value: Notes, chords, and rests
index: 5, name: str_Action, value: select
There are 6 entries in the dictionary.
```

After I wrote **Filter Notes By Durations**, I added some commands to **cmdutils** to call some of its features. There are a lot of combinations, so I decided to leave the duration as a parameter to the commands (which to date had only allowed a single parameter), and effectively absorbed the other options into the command names.

```
Filter_Duration_Notes_Equal_cu(e)
Filter_Duration_Notes_Greater_cu(q)
Filter_Duration_Notes_GreaterEqual_cu(x)
Filter_Duration_Notes_Less_cu(h.)
Filter_Duration_Notes_LessEqual_cu(w..)
```

These commands only allowed the user to filter notes and chords, but not rests, and had one command for each of the 5 "comparison operands". I could have added 2 more sets to handle the other "objects to filter", but I was aware of up the cost of needing to have a huge number of commands in cmdutils to process a plugin with a lot of options.

I was also aware that users might want to write their own Parent plugins, which could be used as commands in **Execute Commands,** but if they wanted to have children plugins, they would need to create a separate plugin for each of the options. In the model above there would be 720 children plugins plus the parent that would need to be created to cover all the options, because Plugin Commands do not take parameters; only cmdutils commands do, and a user cannot create new cmdutils commands.

So I came up with **RunPluginEntry_cu**.

RunPluginEntry_cu lets you call a plugin at a specified entry point, and give it a set of parameters. It is designed around calling Parent plugins that have routines that are passed parameters in a dictionary. It is currently the only cmdutils command that has more than one parameter.

RunPluginEntry_cu is effectively a universal Child Plugin for any plugin method that accepts its parameters through a Dictionary of named fields.

The format for this command is:

RunPluginEntry (<plugin command id>, <Entry point name>, strName*1*, strVal*1*, strName*2*, strVal*2*, … strName*n*, strVal*n*)

Fields are separated by a comma, followed by zero or more spaces.

IMPORTANT NOTE!!! strVal and strName elements may NOT contain commas (or single or double quotation marks)!!!!! This is a simple parser and commas always separate fields.

- The first parameter is the **plugin command id** (essentially the file name with no path, and a ".plg" extension.
- The second parameter is the **plugin entry point name**, spelled exactly as in the plugin.
- The remaining parameters are pairs of (name, value) strings that will be used as entries into the Dictionary that will be passed as a parameter to the plugin entry point, which must be set up to accept its parameters in a Dictionary.

These parameters must be text only, and any Parent routine called by **RunPluginEntry _cu** needs a score and a selection, so **RunPluginEntry _cu** adds the **score** and **selection** entries for the current active score to the Dictionary before calling the plugin entry point. The score and selection are **Bar Objects,** not text, and so they cannot be specified in the **RunPluginEntry _cu** parameter.

Here is **RunPluginEntry_cu** command line whose parameters produce the same result as the **FilterNotesByDuration** dialog shown above:

```
RunPluginEntry_cu(FilterNotesByDuration.plg, API_ProcessSelection,str_Duration, q, str_Operator, =, str_Type, Notes, chords, and rests, str_Action, select)
```

It is similar to the command built-into cmdutils.plg:

```
Filter_Duration_Notes_Equal_cu(q),
```

except that the built-in function will not include rests in its filter.

This is only 1 of the 720 (3 note/rest choices, 5 comparators, 24 durations, 2 actions) combinations available in **FilterNotesByDuration.** Instead of having to add 15 new commands to cmdutils, I can use **RunPluginEntry_cu** with different parameters, and any user can do the same thing if they know which parameters to use.

This is more “programmery” that most of the cmdutils commands, but it provides a great amount of flexibility and easy future expansion.

A user can now call into any appropriately designed Parent plugin (assuming they know the appropriate parameters), without needing to have anything added to the cmdutils library. They can call a plugin they wrote or a published plugin.

For example, the “built-in” commands that call **FilterNotesByDuration,**

```
Filter_Duration_Notes_Equal_cu(e)
Filter_Duration_Notes_Greater_cu(q)
Filter_Duration_Notes_GreaterEqual_cu(x)
Filter_Duration_Notes_Less_cu(h.)
Filter_Duration_Notes_LessEqual_cu(w..)
```

will only filter notes and chords. If you wanted to have a filter for **rests** whose duration were **greater than or equal to a half note,** you could create this command

```
RunPluginEntry_cu(FilterNotesByDuration.plg, API_ProcessSelection,str_Duration, h, str_Operator, >=, str_Type, Rests only, str_Action, select)
```

and use it in a macro. To create this command I copied the text from the command above, changing the duration to “h” for half note, the operator from “=” to “>=”, and the type to “Rests only”.

Since I would like to have Parent plugins that can be called from **Execute Command**s using **RunPluginEntry_cu,** without needing to update **cmdutils.plg**, the Parent plugins I will publish will use this parameter model.

Parameters available in FilterNotesByDuration that can be used by RunPluginEntry_cu

Here is one set of dialog settings for **FilterNotesByDuration**

> **RunPluginEntry_cu(FilterNotesByDuration.plg, API_ProcessSelection,str_Duration, h, str_Operator, >=, str_Type, Rests only, str_Action, select)**

Here are lists of the possible legal values for each parameter passed by **RunPluginEntry_cu** to **Filter Notes By Duration**

- str_Duration
  - "w"
  - "w."
  - "w.."
  - "w..."
  - "h"
  - "h."
  - "h.."
  - "h..."
  - "q"
  - "q."
  - "q.."
  - "q..."
  - "e"
  - "e."
  - "e.."
  - "e..."
  - "x"
  - "x."
  - "x.."
  - "x..."
  - "y"
  - "y."
  - "y.."
  - "y..."

- str_Operator – any values in this list
  - "="
  - "<"
  - ">"
  - "<="
  - ">="

- str_Type– any values in this list
  - "Notes, chords, and rests"
  - "Notes and chords only"
  - "Rests only"

- str_Action
    - select
    - deselect

Generating RunPluginEntry_cu command strings

You can call any plugin that has an entry point that can be called by a Dictionary of commands, such as a plugin that follows my Parent plugin model. At the time of writing I have 3 published plugins that can be called this way: **Filter Notes By Duration, Filter Notes By Positions**, and **Filter Notes By Beat.** You can call the entry point **API_ProcessSelection** in any of these plugins with an appropriate set of dialog properties.

One of the purposes of having **RunPluginEntry_cu**, though, is to empower a user to write a plugin using that model, and then call into the Parent plugin with a Command Macro or Command Plugin in **Execute Commands**. No changes will need to be made to **cmdutils** or **Execute Commands** or **Run Command Macros** to handle these new commands. To all those plugins, it is just another command.

You need to do 2 things to call an appropriate plugin entry point with **RunPluginEntry_cu**

1. Create the command string with all the parameters
2. Add the command string to a macro or plugin.

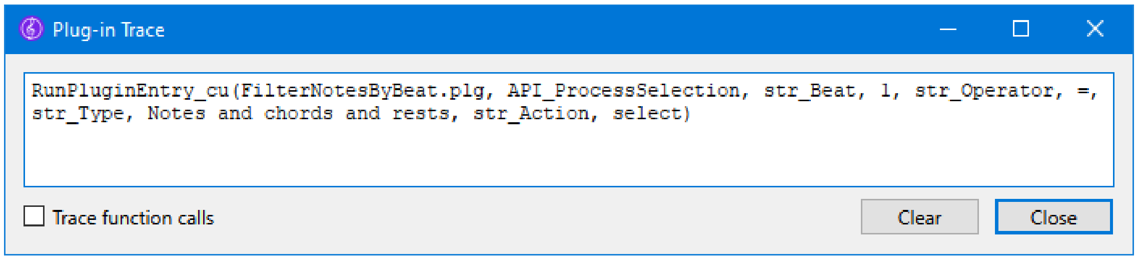
Creating a RunPluginEntry_cu command string

A string like

RunPluginEntry_cu(FilterNotesByDuration.plg, API_ProcessSelection,str_Duration, h, str_Operator, >=, str_Type, Rests only, str_Action, select)

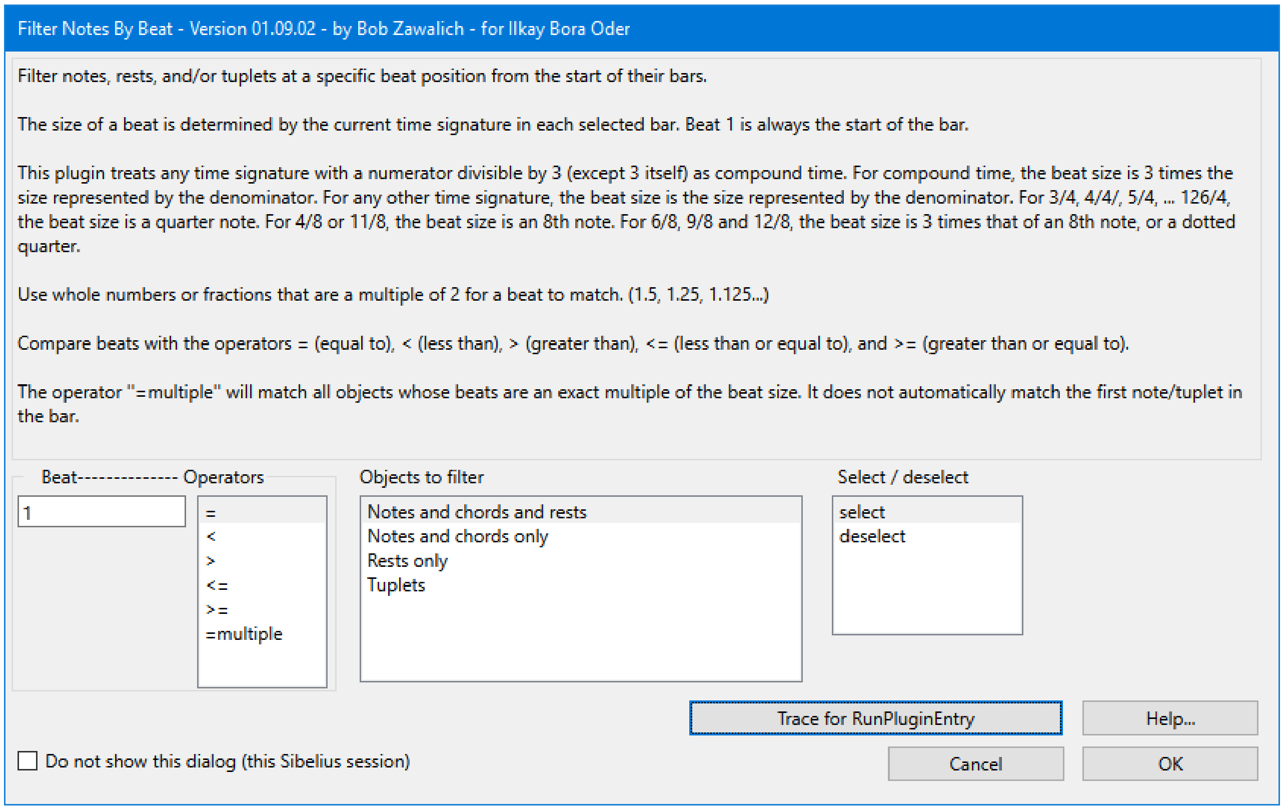
is tedious but not impossible to write. You can open a text editor and type the command and all its parameters, which should be relatively straightforward if you know the commands that your Parent plugin uses.

For some Parent plugins there is a spectacularly easy way to generate the command strings. The 3 **Filter Notes...** plugins described above have a button called **Trace for RunPluginEntry**. If you choose that button, a command line will be written to the Plugin Trace Window, with all the parameters that are appropriate for the current dialog settings in place. This string will be written to the trace window:



```
RunPluginEntry_cu(FilterNotesByBeat.plg, API_ProcessSelection, str_Beat, 1, str_Operator, =,
str_Type, Notes and chords and rests, str_Action, select)
```

for this dialog, with these settings:


Filter Notes By Beat - Version 01.09.02 - by Bob Zawalich - for Ilkay Bora Oder

Filter notes, rests, and/or tuplets at a specific beat position from the start of their bars.

The size of a beat is determined by the current time signature in each selected bar. Beat 1 is always the start of the bar.

This plugin treats any time signature with a numerator divisible by 3 (except 3 itself) as compound time. For compound time, the beat size is 3 times the size represented by the denominator. For any other time signature, the beat size is the size represented by the denominator. For 3/4, 4/4/, 5/4, ... 126/4, the beat size is a quarter note. For 4/8 or 11/8, the beat size is an 8th note. For 6/8, 9/8 and 12/8, the beat size is 3 times that of an 8th note, or a dotted quarter.

Use whole numbers or fractions that are a multiple of 2 for a beat to match. (1.5, 1.25, 1.125...)

Compare beats with the operators = (equal to), < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to).

The operator "=multiple" will match all objects whose beats are an exact multiple of the beat size. It does not automatically match the first note/tuplet in the bar.

You can now select and copy (using ctrl/cmd+c) that text in the Trace Window, and paste it into a text editor, and add it to a .dat file or otherwise make it available to **Execute Commands**.

You can trace a bunch of different settings and get a different command line for each combination.

Adding a RunPluginEntry_cu command string to a macro or plugin

In my opinion, the most efficient way to get a copied command string into a macro or Command Plugin is to edit or create a macro file (which is really just a text file with a .dat extension) in the **Execute_Commands** subfolder of your default **Scores** folder. This macro file can be read into **Execute Commands** using the **Import List** button. **Export List** saves files to that folder, so if the folder does not exist you can use **Export List** to create a file, and create the subfolder at the same time.

You can open an existing dat file and add the command line to that file, or create a new one, and after pasting the command line and saving the dat file, you can immediately bring it into **Execute Commands** with the **Import List** button.

You could also use the **Add new command...** button in **Execute Commands** after copying the traced **RunPluginEntry_cu** command line. That will bring up an dialog with an edit box. Paste the command line into the edit box using ctrl/cmd+v.

At this point, it is like any other command. You can add more commands to the list, or use **Export List** to save it as a macro (dat) file, or use **New Plugin** to generate a plugin from the command.

A couple things to mention:

- Only a very few plugins support the **Trace** option, though I plan to add it to other plugins that can be used in this manner.
- The saved command string will be available for the remainder of the current Sibelius session unless overwritten by another command.
- There is only 1 saved command that can be retrieved by **Get RunPluginEntry text** in **Execute Commands**. If you brought up **Filter Notes By Beat**, and traced statements for 5 different combinations of settings, they would all appear in the plugin trace window, but only the last one you traced would be accessible to **Get RunPluginEntry text.**
- If that is your situation, just copy all the command lines into a text editor, and save a dat file which you can then import into **Execute Commands.**

There you go. Pretty cool, I think.