Action Groups Facility (Agroups)

Table of Contents

Action Groups, or Agroups as we abbreviate, are groups of action entries that an Emacs user can create, save, name and access quickly. The Introduction chapter will explain this in more detail. It will get you started with your own Agroups save file and some basic operations. You are then in a position to experiment with the rest of the manual which serves as a reference manual. The Introduction also serves as a short tutorial and its sections must be read in sequence since each builds on the previous. The notations defined for Meta Keys in the Introduction section, See Section 1.2 [Must Read First], page 2, are used throughout the reference manual and tutorial, so you will be lost if you don't read that first.

1 Introduction to Action Groups

1.1 Overview

Action Groups are groups of action entries that an Emacs user can create, save, name and access quickly. In a general sense these actions are any automation that a user can imagine to help with his activities. The user instantiates these automations as action entries which are a specific instances of some action from the current collection of actions. Actions can be as simple as finding a commonly used file in a buffer, to more complex like executing a previously defined keyboard macro, to very complex like an unimaginable whopper defined by a user created Action Template. Action Templates are an extensibility feature of Action Groups and allows the user to easily create new actions, Agroups supplies a useful predefined collection of actions created with Action Templates.

A collection of action entries is called an action group. Typically the user associates each action group with a concentrated activity such as a project. Each entry of an action group is associated with a specific instance of an action. A group itself is actually an entry associated with a special action called `group` so that a group entry can be yet another group itself. This implies that groups can have subgroups and subgroups can have subgroups and so on. This gives the user structured organizational capabilities.

The Action Groups facility was designed to allow easy and fast creation of groups and entries to help automate an Emacs user's work. At the same time it was designed to allow fast execution of the entries. To this end a user can use the Emacs completion facility to execute entries or key bindings or a combination of both.

In essence Action Groups are just this simple. But this description is pretty abstract, so let's do some concrete things with Action Groups to see what they are all about. It is important to go through these sections in the order presented since each depends on the previous.

1.2 Must Read First

This document assumes that the user has bound the `agroups` command to a keystroke. For example a common thing to do is bind `agroups` to something like the keystroke `C-zC-a`. This can be done by placing the following in your ~/.emacs file

```
(global-unset-key "\C-z")
(global-set-key "\C-z\C-a" 'agroups)
```

Note that this example removes the binding of `C-z` and makes `C-z` a prefix key. This is not too serious since the Emacs default key-binding of `C-z` is the same as `C-xC-z`. The user is free to bind the `agroups` command to what ever he likes, but for the purpose of illustration this document uses `C-zC-a`.

Now for the *must read first* stuff. Since the `agroups` command and all bindings within a specific user's Agroups are completely customizable there is a need in this document to use some abbreviations for some of the functions bound. In particular Agroups provides a few meta keys that allows a user to escape from the usual Agroups entry selection. There will be more of a description of meta keys later but for now we just list these abbreviations and their meaning. In this table `=` means `"the binding of"`

```
KA = the agroups command key (eg. C-zC-a)
KO = the Agroups meta operation key (default RET)
KG = the Agroups meta group key (default TAB)
KC = the Agroups meta completion key (default SPC)
```

This document and Agroups itself use the standard abbreviations of RET for the Return or Enter key on a keyboard, TAB for the Tab key and SPC for the Space Bar key.

We gave as illustration a default binding of `C-zC-a` for `agroups`. We abbreviate this binding as KA in this document regardless if it is bound to some other key combination. Note from the above that automatic default bindings in Agroups for KO, KG and KC are respectively RET, TAB and SPC. You will see in another section that the user can change these bindings to suit his preferences. So if we had bound `C-zC-a` to the `agroups` command and settled for the default bindings then "KA KO" would be interpreted as typing `C-z C-a RET`.

But having settled this we will no longer refer to these specific meta key bindings, we will simply just say KA KO KG and KC. So for example if we say "KA a" it means have `agroups` execute action "a" in current group. And "KA KG g a" means execute action "a" in group "g". We will say more on meta keys later but it is important to read this section first so that there is no confusion where these abbreviations are used.

1.3 Trying Action Groups from Scratch

To use Action Groups you must first load the '`agroups.elc`' file. See the Emacs Users Guide for how to compile the '`agroups.el`' and then load the compiled .elc file.

The Emacs user can execute Action Groups by giving the command `agroups`. Typically `agroups` is bound to a keystroke since it combines nicely with other keystrokes for user actions. We will use the abbreviations KA KO KG and KC mentioned above, See Section 1.2 [Must Read First], page 2.

This section assumes that you do *not* have an existing Agroups save file, by default '`~/.agroups`'. So if you happen to have a save file at this point for some reason move it temporarily out of the way.

The first thing that we will look at is what Agroups gives you as a default set of groups. First let's just try entering simply KA. After doing this you should see in the minibuffer

```
Select entry:
```

This is the default action of Agroups. It is asking you to select an entry in the current group for execution. But since we haven't created any entries yet let's now try an Agroups operation by entering KO. Recall that from the previous section that by default KO is bound to RET. You should now see in the minibuffer

```
Select operation:
```

If you screwed up somewhere you can always abort any Agroups processing by typing C-g. Try aborting and then get back to the "Select operation:" prompt by entering again "KA KO". At this point it is waiting for an Agroups operation entry. If you enter KO again you should see a list of all Agroups operations come up in the Agroups buffer called "*Agroups*". And if you keep entering KO the Agroups buffer will scroll until it gets to the end and then wrap and scroll and so on. Try this and note that one of the operations in the Agroups buffer is the line

```
Display current group entries (keys: d)
```

In all Agroups entry displays including operations, group entries and groups this entry format is the same. It is the entry id followed by keys, if any, in parentheses. If the keys are there it means that you can select this entry with these keys. So let's abort again with C-g and then enter

```
KA KO d
```

and note that this a lowercase "d". You should then see in the Agroups buffer

```
Group: global (keys: .)
```

This is telling you that the current group id is "global" and that it has no entries. When you were looking at the list of operations you might have noticed the line

```
Display top level groups (keys: D)
```

So let's try that with

```
KA KO D
```

and note that this an uppercase "D". Now you should see in the Agroups buffer

```
Top level groups (C marks current persistent, c marks current)
C  global (keys: .)
   zzzap (keys: !)
```

This displays all existing groups. When you first try to execute an Agroups command and you don't have an Agroups save file it creates one for you and automatically creates two groups in that file: one with an id of "global" and one with an id of "zzzap". Note that in this display the "global" group is marked with a "C" on the left. This indicates that "global" is the current group and that is consistent with what we saw in the previous group entries display which was supposed to display the current group.

The "global" group is a complementary group that Agroups starts you off with and is also used for examples in this document. After you know how to use Agroups and create your own groups you are free to remove this group. However you may want to keep it as a place to put entries that you don't think belong in any other group. However the group with the id "zzzap", which we refer to as the zzzap group, can never be removed. Oddly enough you can change the id and keys of the zzzap group but never remove the group itself. The reason for this is that the zzzap group has a special significance in Agroups. There is no way to delete an entry in Agroups, you can only move entries around. Instead of deleting an entry you simply move it to the zzzap group. There is an operation to purge entries from the zzzap group which is explained in the Zzzap Group section, See Section 3.6 [Zzzap Group], page 16.

1.4 Creating our first action Entry

Now that we know some simple navigation in Agroups let's create an entry. Every entry in an Agroups group represents some action. For example an action could be a previously defined keyboard macro. Perhaps the simplest action might be simply finding a commonly used file, so let's start with that. First bring some file into a buffer with the usual Emacs command `find-file` (by default C-xC-f). Now, while in that buffer enter

```
KA KO a f
```

You should then see in the minibuffer

```
Enter entry id:
```

In all cases where Agroups prompts for an id the user has the option of just hitting RET in which case Agroups generates a reasonable id for that entry or to type some more descriptive id. For the sake of this example let's do the latter and type the id

```
My commonly used file
```

and then type RET. Now you should see

```
Enter Accelerator Keys () :
```

Agroups is now asking for the keys of the id/keys pair mentioned above. These are called Accelerator Keys since a user always has the option of choosing none in which case to select the entry later they type in the id manually or use the Emacs completion facility or they can choose a sequence of keystrokes here which accelerates making the selection later by just entering those keystrokes. Even if the user chooses some accelerator keys he can later still also use completion to select this entry so it never hurts to have accelerator keys. For the sake of this example we will type in the keys "f1" and then RET to complete the entry. This should pop-up the Agroups buffer with the following

```
Added entry (marked by m) to group: global
m  My commonly used file (keys: f 1)
```

In the previous section we displayed the current group and saw no entries, so now we should see one. So enter "KA KO d" and the Agroups buffer should display

```
Group: global (keys: .)
  My commonly used file (keys: f 1)
```

Now let's try to execute this new action entry. First kill the buffer we brought up to create this entry. You can do this with the `kill-buffer` command (by default C-x k). Then try "KA f 1". It should bring the file back into a buffer.

1.5 Creating our first Group

Now that we've created an entry in the "global" group let's create one of our own groups. We do this by entering "KA KO G" (notice, that is a capital "G") and should see in the minibuffer

```
Enter a group id:
```

For the sake of this example let's give it the id "test" and then accelerator key "t". After this you should see two things. The minibuffer displays

```
Current group: test
```

and the Agroups pop-up buffer displays

```
Top level groups (C marks current persistent, c marks current)
C  global (keys: .)
c  test (keys: t)
   zzzap (keys: !)
```

You could see the same thing by entering "KA KO D" as before. This display shows two things. First that the test group has been added and second that the lowercase c on the left shows that this newly created group becomes the current group which is consistent with what the minibuffer says.

Now let's create an entry in this new group. This time let's create a slightly more complex action that will do a meaningless thing just for illustration. We will create a keyboard macro that goes to a junk buffer and puts trash in it. To do this type the following

```
C-x ( C-x b junk trash C-x )
```

This will create the keyboard macro. If you want, you can test the keyboard macro by killing the junk buffer and then typing "C-x e". This should bring up a junk buffer with trash in it.

Now let's create an Agroups entry in our test group with this keyboard macro. To do this enter

```
KA KO a k
```

You should then see the prompt

```
How keyboard macro will execute
 0 = Will just execute
 1 = Will execute if user says yes
 2 = Will load into last keyboard macro
```

We simply want a keyboard macro that will just execute when we select this action, so just hit RET here to choose the default 0 choice. When it asks for the id give "put trash in a junk buffer" and for keys "j". After this you should see in the Agroups buffer

```
Added entry (marked by m) to group: test
m  put trash in a junk buffer (keys: j)
```

You now have a persistent copy of this keyboard macro so that anytime the test group is the current group you can enter "KA j" and you should have a junk buffer with trash. Kill the junk buffer again and try it.

1.6 Performing cross group operations

By now it should be apparent that there is always a Current Group. Actually there is always a Current Persistent Group and a Current Group. This allows you to have a current group that you work on for some time and then easily switch to another group and when done, revert back. However if all that you want to do is a single operation or action entry in some other than current group that is easy to do as well.

If you are following the subsections in this introduction sequentially as was recommended you should now have a test group with one entry

```
Group: test (keys: t)
  put trash in a junk buffer (keys: j)
```

and a global group with one entry

```
Group: global (keys: .)
  My commonly used file (keys: f 1)
```

The current group is test but let's say that we want to just execute the action "My commonly used file" in the group global without making global the current group. You can execute any operation or action in any other group using the group meta binding that we described above as KG, which at this point should be bound to TAB. In essence the group you choose after a KG temporarily becomes the current group.

So first let's display the contents of the global group without making it current. You can do this by entering

```
KA KG . KO d
```

And now let's execute the "My commonly used file" action in the global group with

```
KA KG . f 1
```

Executing operations in other groups is not just convenient but functional as well. For example let's move the "My commonly used file" entry in the global group into our current test group. Do this by entering

```
KA KG . KO m f 1 t
```

Upon doing this successfully you should see the following in the Agroups buffer

```
Moved entry (marked by m) to group: test
m  My commonly used file (keys: f 1)
   put trash in a junk buffer (keys: j)
```

This "cross group" capability of KG is Agroups way of making a single pattern for any operation between groups and subgroups.

1.7 Using completion on Agroups ids

There is only one more meta binding left to try. That is the meta binding called completion that we described as KC, which at this point should be bound to SPC. SPC was chosen as the default binding for KC since that is the standard keystroke used for the Emacs completion facility. Also, KC introduces the Emacs completion facility in the context for any operation, group or entry the user is considering executing.

As a simple example, above we executed the "put trash in a junk buffer" with keys

```
KA j
```

We could have executed it with completion

```
KA KC SPC put SPC RET
```

Actually, Agroups tries to be intelligent here and it disambiguates keys typing versus id completion typing. So in this case you could do the same completion without the KC by entering

```
KA put SPC RET
```

or even

```
KA p RET
```

In fact you could choose to never use keys at all and do literally everything using completion. Although, it is pretty difficult to beat accelerator keys in terms of making executing automations optimal,

1.8 Making a group the Current or Current Persistent

As we mentioned in a previous section there are two notions of the Current Group. When a group is the Current Persistent group then the next time you enter Emacs that group will be the Current Group. But you can use this to conveniently make another group the Current Group, do some work in that group and then revert back to the Persistent Current Group.

At this point if you have been following the instructions then when you do "KA KO D" you should see

```
Top level groups (C marks current persistent, c marks current)
   global (keys: .)
C  test (keys: t)
   zzzap (keys: !)
```

as we have seen previously. The C is marking the Current Persistent group. That is, if you exit Emacs and restart you should see the same since the test group being Current Persistent is saved in your Agroups save file. The fact that there is just one C means that it is not only the Current Persistent Group but also the Current Group. To understand this better let's make the global group the Current Group by entering

```
KA KG . KO c
```

Now do a "KA KO D" you should see

```
Top level groups (C marks current persistent, c marks current)
c  global (keys: .)
C  test (keys: t)
   zzzap (keys: !)
```

The lowercase c is showing that the global group is now the Current Group and the uppercase C is showing that the test group is now the Current Persistent Group. What this means is that all operations and selections you do now will be done in the global group but if you were to restart Emacs, the test group will become the Current Group again along with being the Current Persistent Group. A this point you could have entered

```
KA KO C-c
```

which means revert back to the Current Persistent Group as the Current Group. In other words the C marked test group would revert back to becoming the Current Group. But instead let's enter

```
KA KO C
```

which makes the global group the Current Persistent and Current Group. So now if you do a "KA KO D" you should see just one C again but now it is on the global group. But for the next section let's go back to having the test group being the Current Persistent and Current Group. Do this by entering

```
KA KG t KO C
```

The C operation says make the group you specify the Current Persistent and the Current Group.

1.9 Editing entries once created

Once you create entries and groups it is easy to edit the components of them. As an example let's edit the entry "My commonly used file" that we moved to our test group in the previous section. Make sure that our test group is the current group with "KA KG t KO c". Suppose that you want to change the id of our entry to "My one commonly used file" and change the keys to be just "1". First change the keys by entering

```
KA KO e f 1
```

Lets make a temporary (probably wrong) assumption that the file you picked for the id "My commonly used file" in the test group was `/home/me/MyFile`. Then you should then see the following in the Agroups buffer

```
This entry is in group: test
  id: My commonly used file
  keys: f 1
  action: file
  object: This is a template based entry with slot values:
    File path specification:
      /home/me/MyFile
```

This displays all of the components of the entry: id, keys, action, and object. All entries (including groups) have this same form. You can easily remember what these components mean with the saying that ties them all together in the Entry Components section, See Section 3.4 [Entry Components], page 15.

At the same time that this entry is displayed in the Agroups buffer you are prompted in the minibuffer to select which of these components to edit

```
Enter to edit: id: i, keys: k, object: RET, exit: SPC:
```

For now we want to edit the keys component so enter "k". And now you should see the familiar prompt

```
Enter Accelerator Keys () :
```

So just enter "1" for the accelerator keys followed by a RET and now you should see what we just saw re-displayed in the Agroups buffer

```
This entry is in group: test
  id: My commonly used file
  keys: 1
  action: file
  object: This is a template based entry with slot values:
    File path specification:
      /home/me/MyFile
```

and note that the keys now just has "1". Now lets edit the id by entering

```
KA KO e 1
```

Note that now we are saying edit the entry with keys "1". After entering the above[1] at the component prompt enter "i" for id which should pop up the following in the Agroups buffer

```
      ---- Edit or enter below and type C-cC-c when done ----
My commonly used file
```

This is the standard way to edit ids and objects. You are given a buffer with the contents of the id or object slot value and may edit the buffer until you have what you want. It is permissible to leave this buffer and do any other Emacs commands and then return to this buffer. When you are though editing in this buffer type C-cC-c as it says at the top of the buffer. At that point the edited entry will replace the old entry and be made persistent.

So edit this buffer to read instead

```
      ---- Edit or enter below and type C-cC-c when done ----
My one commonly used file
```

and type C-cC-c. If you do a "KA KO d" you should now see

[1] There is actually as easier way to re-edit the last entry edited, See Section 9.1 [Editing Operations], page 39.

```
Group: test (keys: t)
  My one commonly used file (keys: 1)
  put trash in a junk buffer (keys: j)
```

You can edit the components of the current group in a similar way by entering (with an uppercase "E")

```
KA KO E
```

For the next section edit the id of the current group test so that its id is "Test" with an uppercase "T". Now do "KA KO d" again and you should see as a result

```
Group: Test (keys: t)
  My one commonly used file (keys: 1)
  put trash in a junk buffer (keys: j)
```

1.10 Creating subgroups for organization

Once you start creating lots of actions you may want to organize them into subgroups. Subgroups can have subgroups and so on. This organizational structure is similar to file directories. Subgroups are identical to top level groups. In fact you can promote a subgroup to a top level group and vice versa. The only difference is that a subgroup is a group that is an entry in another group. And a group is identical to an entry. It has the same form of entry components that we saw in the previous section, See Section 1.9 [Editing entries once created], page 8. Its action is `group` and its object is the entries it contains. The action `group` means select one of my entries and then execute that.

To get a clearer picture of this let's create a subgroup in our newly renamed group Test. Imagine that our newly renamed entry "My one commonly used file" is now only one of many files that we want to bring up easily and we don't want to mix files with other kind of actions. So let's create a subgroup called "Files" and put "My one commonly used file" into it for starters. Make sure that the current group is Test. We first create the subgroup by entering

```
KA KO g
```

Again, like any other new entry Agroups will prompt you for an id and keys. Make the id "Files" and keys "f". Now if you enter "KA KO d" you should see the following

```
Group: test (keys: t)
  Files (keys: f)
  My one commonly used file (keys: 1)
  put trash in a junk buffer (keys: j)
```

Now lets move our file action into the new subgroup. Do this by entering

```
KA KO m 1
```

At this point Agroups prompts you for the target group. Since we want to move this into a subgroup of our current group we enter "t". Normally this would mean move our entry into the Test group itself, but since we now have a subgroup in our Test group we get the following prompt that gives us a choice of the Test group itself or a subgroup of this group

```
There are subgroups in this group: Test
 0 = Select this group as target group
 1 = Select a subgroup of this group as target group
```

and in the minibuffer

```
Choose target (default 0):
```

For the target choose "1". And then to

```
Select group:
```

enter "f" which should pop-up

```
Moved entry (marked by m) to group: Files
m  My one commonly used file (keys: 1)
```

Note that we could have done this move a lot easier using a current group local move See Section 8.4 [Local moves], page 37.

Now if you enter "KA KO d" you will see

```
Group: Test (keys: t)
  Files (keys: f)
  put trash in a junk buffer (keys: j)
```

Since a subgroup is just another entry whose action is on its object, being its group of entries, you can also perform any operation on that group object. Consequently you can display the entries of the subgroup with

```
KA KO f KO d
```

Which should pop-up

```
Group: Files (keys: f)
  My one commonly used file (keys: 1)
```

1.11 Philosophy of Action Groups

The concept of *automation* is important to the philosophy of Action Groups. What we mean by automation is that when you hit a button or key it automatically does something for you that would ordinarily require a lot of work and recall on your part. It seems obvious that automations save time and make life more pleasant. What is not obvious is how to make automations easy to create, manipulate and trigger.

Agroups tries to achieve this non-obvious task. Keyboard macros in Emacs are a good example of how to make automations easy to create and execute. Action Groups is a more encompassing type of automation tool where specific actions have a purpose similar to keyboard macros. In fact, an action can be a keyboard macro. The action is the unit of automation. An action group provides an organizational unit for collections of action instances called entries. In developing Action Groups the following criteria are important

1. Entries and groups must be easy to create, edit, move around and manipulate.
2. Entries must be easy to execute.
3. Entries and groups must automatically be made persistent.
4. A group should be as much like an entry as possible.
5. Some mechanism must be provided to make it easy for a user to define new action types.

The current Agroups addresses these criteria point by point as follows:

1. Agroups operations were designed to be few, powerful, combinatorial and executed exactly the same way that action entries are executed. This design make some operations see rather odd, for example the simple move operation serves to move both groups and entries and also serves to delete, replace and copy or clone groups and entries depending on the move targets. But this makes the design simpler and also when you get used to it makes it easier in that you only have to remember how to move entries.
2. Action entries can be bound to any keystroke easily and interactively and entries can also always be selected by using Emacs completion.
3. Agroups has a save file and changes are automatically saved there. Usually a user does not have to think about what is happening here.
4. Most operations that can be applied to entries, can in turn be applied to groups. A group can be an entry in a group and hence, one can organize subgroup trees.
5. Agroups tries to achieve this with what it calls *action templates* that allow everything about any given automation pattern to be easily specified and the variable parts of the pattern are automatically asked for from the user when a template instance in requested by the user.

Action Groups started off as a sort of experiment in activity flow automation that eventually evolved into something called Action Stacks that allowed you to stack up your automations like a stacks of notes on your desk. Although you could access any entry in a stack quickly this paradigm turns out to be too simple since activity flow tends to be more randomly structured while at the same time hierarchically structured. In another dimension it needed to be easy to quickly switch between different groups of activities and easy to evolve groups of actions during activity flow. This implied a complete rethink and rewrite of Action Stacks into what is called here Action Groups. Action Groups however is still considered an experiment and suggestions of improvements or philosophy are welcome.

2 Action Groups Save File

The Agroups program makes group and group entries persistent by copying any changes in the memory resident Agroups structure to a disk save file when the changes happen. When the save file is updated in this way a backup is created using the standard Emacs backup mechanism. The default save file is ~/.agroups, but the user can change this by setting the variable

```
agroups-file
```

to a different save file in the user's ~/.emacs file. For example

```
(setq agroups-file "~/mydir/agroups")
```

The user can have as many Agroups save files as desired and can switch between them by either setting this variable or can switch between them temporarily using the operations

```
Set agroups Save file (keys: s s)
Set agroups Save file back to persistent (keys: s S)
```

So lets say the save file is the default ~/.agroups and we want to temporarily change it to ~/mydir/agroups. We would type

```
KA KO s s ~/mydir/agroups
```

When you move between different Agroups save files in this way Agroups keeps track of the persistent save file that was set in the variable `agroups-file` when you first run Emacs. When we want to move back to the persistent save file we type

```
KA KO s S
```

You can view the current actual save file with the operation

```
View agroups Save file (experts only) (keys: v s)
```

This operation warns "experts only" since you need to know more or less what the meaning of this file is and unless you are a real expert you should not try to edit this file since you can really shoot yourself in the foot. This operation brings the save file up in view mode and as long as you are only viewing the file you can peruse the file without doing any damage.

3 Group Structures

The Action Groups structure is quite simple and mirrors a hierarchical file system directory structure where the group is like a directory and a group entry is like an ordinary file.

3.1 Groups and Entries

An Agroups top level group structure is a hierarchical structure composed of groups of entries. A group itself is by definition an entry and therefore groups can have entries that are groups. When a group has an entry that is itself a group then we refer to that entry as a subgroup, See Section 3.3 [Subgroups], page 15. You can have subgroups inside of subgroups recursively with no limit. Also, any subgroup is referred to as a group and is equivalent to any other group in form.

The whole Agroups group structure is contained in a single top level group. This single top level group has no identity itself so we just refer to all of the entries in this single top level group as top level entries. All of these top level entries are in fact forced to be groups and we also refer to them as top level groups. Each Agroups save file, See Chapter 2 [Save File], page 13, maps a single group of top level groups.

Group entries that are not subgroups will be instances of actions and we call these "action entries". All Agroups entries, including groups, have a uniform set of components, See Section 3.4 [Entry Components], page 15. This keeps the model simple which is important for example for editing individual groups or entries, See Chapter 9 [Editing Entries and Groups], page 39.

A major advantage of the equivalence of groups and entries is that any operation that can be applied to an entry can be applied to a group.

3.2 Group Information

You can view all top level groups with the operation

```
Display top level groups (keys: D)
```

This operation is not only important for viewing all top level groups but also marks the current group and current persistent group. The current group is marked with a small “c” in the left margin and the current persistent group with a capital “C” in the left margin. If the current and current persistent group are the same group then that group will be marked with only a capital “C”.

If the current or current persistent group is not a top level group it will still be marked but indented under the top level group that it is a subgroup of. No matter how deeply nested inside the top level group the marked group is it will be indented the same amount. The indenting only serves to indicate that the current or current persistent group is somewhere underneath that top level group. For example in the following display of top level groups

```
Top level groups (C marks current persistent, c marks current)
   Group green (keys: g)
C  Group blue (keys: b)
   Group red (keys: r)
c    sub-sub-group of red (keys: r 2)
   global (keys: .)
   zzzap (keys: !)
```

The current persistent group is the top level group "Group blue" but the current group is a subgroup of a subgroup of "Group red". But nothing other than the current group's id text here indicates that it is in fact a subgroup of a subgroup of "Group red".

You can view the entries in the current group or any other group or subgroup with the operation

```
Display current group entries (keys: d)
```

For how to view groups or subgroup other than current groups with this operation, See Section 4.3 [Current Group One Time], page 21. You can also edit groups to see or change detailed information about a group, See Chapter 9 [Editing Entries and Groups], page 39.

Whenever you display a group with this operation if the group is not the actual current group then it gets recorded and you can easily refer to this displayed group later, See Section 4.7 [Last Non-Current Displayed Group], page 23.

3.3 Subgroups

The top level group entries must all be group entries. So for example a top level entry can not be a keyboard macro. But all entries below the top level can be any kind of entry including another group entry. When an entry is a group entry we call it a subgroup.

Each entry is has an action component like `file` or `dired`. If an entry has the action component `group` then it is a group entry. But the action `group` can be executed just like any other action and when executed means simply

```
For any subgroup that can be executed in a sequence, pretend that
subgroup is the current group for the rest of that sequence.
```

Actually this is a specific instance of the "Current Group One Time" principle and the section, See Section 4.3 [Current Group One Time], page 21, has examples of the above principle for subgroups.

3.4 Entry Components

Every entry has four components

```
id              Identification of entry
keys            Accelerator keys
action          Action that will happen when entry is selected
object          Object that action will be taken on
```

The meaning of these components can be explained with a saying that ties them all together:

The *action* on *object* can be achieved by typing
keys of the entry identified by *id*.

Actually this explanation would be really complete if we just allow the part "typing keys" to allow substitution with "typing completion of id". That is, at any time a user can use completion instead of accelerator keys.

A simple example is

```
id              My file 1
keys            f 1
action          file
object          ~/somedir/myfile1
```

So when typing "KA f 1" the file "~/somedir/myfile1" is found in a buffer. Here is an example of a group entry components

```
id              My group
keys            g
action          group
object          (entry1 entry2 ... entryN)
```

So when typing "KA g" Agroups selects the object and a subgroup and waits for for your selection of entry1 entry2 ... entryN.

Keys are intended to be strings of any character that can be typed. There is no limit to the number of characters for the keys component and they can be any characters including control, meta and function characters except C-g and RET. The C-g character is used to abort a keys entry and the RET character is used to complete a keys definition entry. Agroups makes a distinction between upper and lower case characters. For instance "A" can be used to select something differently than "a".

3.5 First Given Groups

When the user first declares a new empty save file, See Chapter 2 [Save File], page 13, Agroups will automatically create two top level groups. These are the zzzap group and the global group. The global group can be removed by the user but the zzzap group can never be removed since it is used in the mechanism for removing entries and groups, See Section 3.6 [Zzzap Group], page 16.

The global group is given initially as a courtesy and as a starting top level group other than the zzzap group. It is also used in examples in this document. Even though you may remove the global group you may want to keep it around as a group of actions that are intended to be common to all other groups. For example, we show how access to a global group can be bound to another Emacs key, See Section 13.3 [Creating New Agroups Commands], page 62.

3.6 Zzzap Group

The zzzap group is given to you initially and can never be moved to another group or deleted; however it can be cloned. You can change the id and accelerator keys but not the zzzap group itself. In this document we refer to this group as "the zzzap group" which means the group with initial id "zzzap". So if you were to change the id of this group there is no confusion with the terms zzzap or Zzzap used in this document.

The zzzap group is kind of like the kill stack in Emacs. Even though you can effectively removed unwanted groups and entries from a group by moving them to the zzzap group

they still remain there persistently until purged or "zapped" from the zzzap group. When comfortable you can purge the whole zzzap group with the operation

```
Purge Zzzap group (keys: z)
```

So, typing

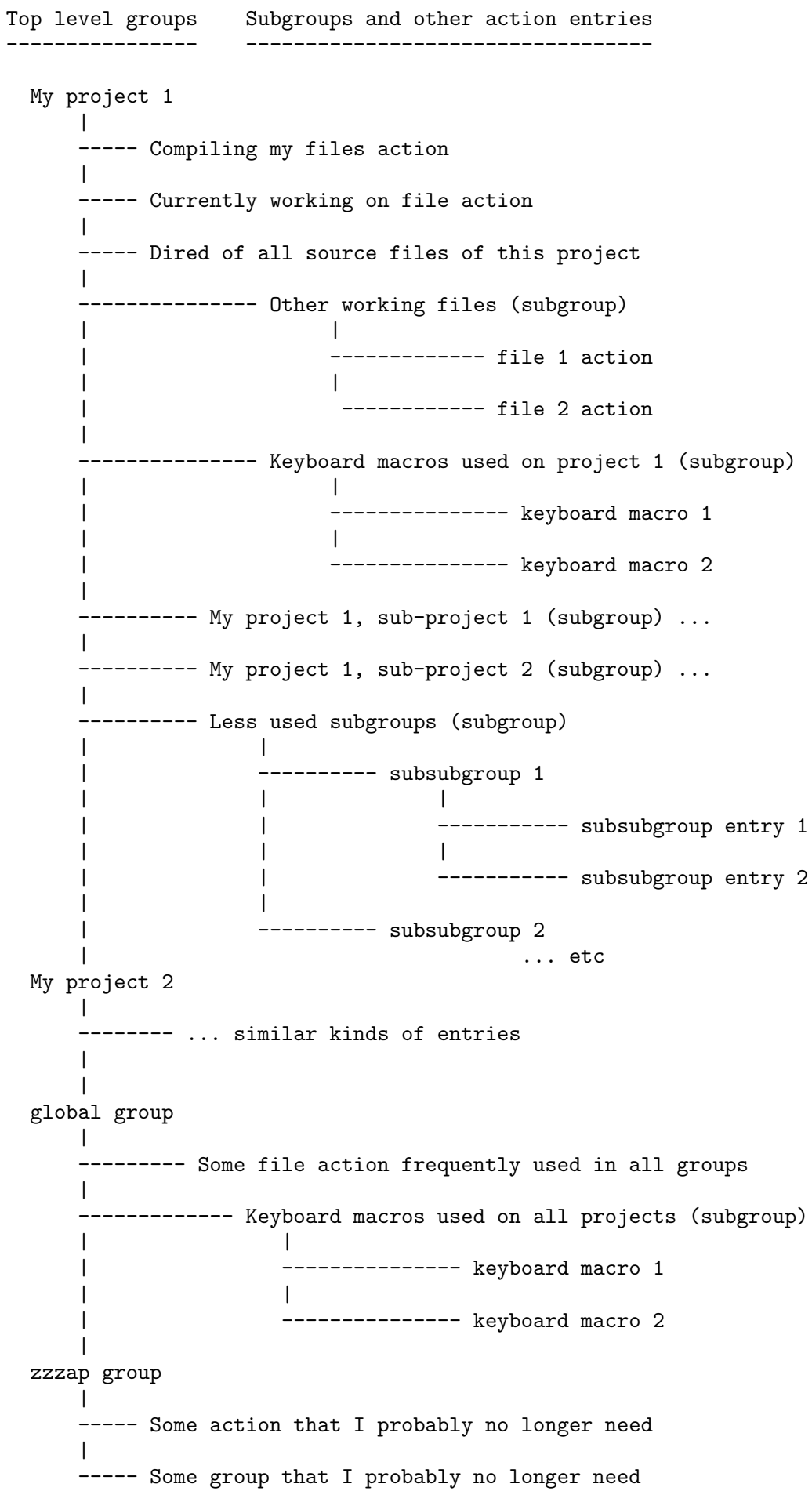
```
KA KO z
```

would remove everything in the zzzap group forever. However, even that is not completely true since before the purge operation the Agroups save file is saved to a backup file as does every operation that changes the save file.

In the event that a subgroup of the zzzap group is the current group when it is purged the current persistent group becomes the current group. If such a subgroup is the current persistent group the current group becomes the current persistent group. If both the current group and current persistent group are contained by the zzzap group when purged the zzzap group becomes the current and current persistent group.

3.7 Group Organization Example

This section gives no useful information about Agroups itself, it just gives some ideas about how Agroups can be used. Typically a user will associate higher level groups with the actions of some project or class of actions. Here is a typical group organization example


Top level groups
Subgroups and other action entries
My project 1
Compiling my files action
Currently working on file action
Dired of all source files of this project
Other working files (subgroup)
file 1 action
file 2 action
Keyboard macros used on project 1 (subgroup)
keyboard macro 1
keyboard macro 2
My project 1, sub-project 1 (subgroup) ...
My project 1, sub-project 2 (subgroup) ...
Less used subgroups (subgroup)
subsubgroup 1
subsubgroup entry 1
subsubgroup entry 2
subsubgroup 2
... etc
My project 2
... similar kinds of entries
global group
Some file action frequently used in all groups
Keyboard macros used on all projects (subgroup)
keyboard macro 1
keyboard macro 2
zzzap group
Some action that I probably no longer need
Some group that I probably no longer need

Of course much like file directory structures, your organization is completely your choice. It is also your choice as how to use subgroups to organize your actions or not use them at all. However subgroups can be quit useful and here are a few examples:

1. Manipulating project groups. As an example, when you want to put on hold and stash away a project that you have represented as group you can move to another group called, let's say, "Old Projects".

2. Using a subgroup as a prefix. For example, lets say that you have a project group where you have all sorts of text strings that you need to recall and insert that are specific to that project. You can create a subgroup called "Insert Text" with accelerator keys "i". The "i" then in effect becomes a prefix for inserting text. So if you create in the subgroup "i" actions to insert text then you can type

```
KA i <one of the text inserts>
```

In effect it is as if the "i" is a prefix key to all of your insertion automations. If KA is bound to a control key then it may make sense to us "C-i" here as an accelerator keys instead of "i".

3. Category systems. For example, extending the example in (2.) instead of just a subgroup "i" we might have subsubgroups in "i" for different text categories. And we could enter

```
KA i <select text category> <one of the text inserts>
```

to get to a specific instance of a category.

4. Sub-projects. You could have a project that has sub-projects and sub-sub-projects etc. Since a subgroup can be made the current and/or current persistent group it makes it easy to switch contexts while associated the sub-projects with the major project.

4 The Current Group

The "current group" is an important concept in Agroups operations. Most Agroups operations focus on the current group.

4.1 What is the Current Group

Unless otherwise specified all operations and the selection of a group entry apply to the current group. After typing KA, Agroups is waiting for the selection of a group entry to execute from the current group. After typing KA KO, Agroups is waiting for an operation to be selected that will operate on the current group unless otherwise specified.

Any top level group or subgroup nested to any level from any top level group can be made the current group. At any time there is only one current group. You can see in the minibuffer what the current group is with the operation

```
What is current group (keys: w)
```

4.2 What is the Current Persistent Group

The current persistent group is the group that will be the current group when you first run Emacs. In other words, it is the group that is saved to a save file persistently as the current group.

The current group and current persistent group can be two different groups during an Agroups session. And in fact, this turns out to be a simple but quite useful device. For one example, the current persistent group serves as a sort of home base group while switching between current groups. As a specific example of that: You may be working on a project for days in which case you might want to make that project the current persistent group. But then someone interrupts you with a problem on another project. So you make that other project the current group for awhile then go back to the persistent current group project as the current group.

Any top level group or subgroup nested to any level from any top level group can be made the current persistent group. At any time there is only one current persistent group. You can see in the minibuffer what the current persistent group is with the operation

```
What is current persistent group (keys: W)
```

You can also see both the current group and current persistent group in relation to other groups with the operation for displaying the top level groups See Section 3.2 [Group Information], page 14.

As motioned above, when you first run Emacs the current group is also the current persistent group, but it common to set any group or subgroup to be the current persistent group See Section 4.4 [Setting Current Group], page 22. The group that is the current persistent group when you exit Emacs will be the current persistent group and current group when you reenter Emacs at a later time.

4.3 Current Group One Time

At any time the current group is the last group selected as the current group by the user. However the current group status can for one time, that is for one operation or one group action, belong to any group or subgroup. The consequence of this can be stated simply in the following principle

```
The "current group one time" means that for any group that can be
selected by a user input sequence, pretend that that group is the
current group for the rest of that sequence.
```

Sequences that select groups fall into two categories, those that select groups with operations and those that select subgroups by executing them. Those that select by operations fall into two more categories, those that select using the meta operation KG and those that select with ordinary operations.

Any group can temporarily become the current group by typing KA KG and then selecting any top level group. Or selecting any top level group and then selecting any subgroup. For example, selecting an entry in another group, the sequence

```
KA KG G a
```

selects a group named “G” and then an entry in that group named “a” regardless of what the current group is. And selecting an operation in that group, for example

```
KA KG G KO d
```

would display the entries of the group named “G” regardless of what the current group is.

Similarly if the current group has a subgroup entry named “g” then

```
KA g a
```

would execute an entry named “a” in that subgroup named “g”. And

```
KA g KO d
```

would display the entries in the subgroup “g”.

Similarly if that subgroup “g” has a subsubgroup “g” within it and in turn that subsubgroup has an action “a” then

```
KA g g a
```

would execute that action “a” and

```
KA g g KO m a
```

would move that action “a”. One example of an ordinary operation that selects a group as current one time is that of visiting the last displayed group See Section 4.7 [Last Non-Current Displayed Group], page 23.

The reason that Agroups has this "Current Group One Time" concept is by doing this there is only one paradigm for operating on a group and its entries, which is the paradigm for operating on the current group. Consequently, anywhere in the document that refers to an operation on or in "the current group" also applies to a Current Group One Time sequence.

4.4 Setting Current Group

At all times some group is the current group. To make some other group the current group use the operation

```
Make current group the Current group (keys: c)
```

This sounds a little confusing at first until we realize that when we say current group in an operation like this we mean more generally the current group or current group one time See Section 4.3 [Current Group One Time], page 21. In general to make some other group the current group select the other group though any sequence of operations and then apply this "c" operation. For example

```
KA KG ! KO c
```

makes the zzzap group the current group one time and then the "KO c" makes it the current group. Another example is

```
KA a KO c
```

which makes a subgroup within the current group named "a" the current group. The operation

```
Make a group the Current persistent group (keys: C)
```

does the same thing as the "c" operation but the current group also becomes the current persistent group.

Conversely, the current persistent group can be made the current group with the operation

```
Make current persistent group the Current group (keys: C-c)
```

This operation is also useful when you want to make some group other than the current persistent group the current group and then revert back.

4.5 Auto Setting Current Group

There are conditions where the current group is automatically set by Agroups as follows

- When a new top level group is created it automatically becomes the current group See Chapter 5 [Creating Groups], page 24.
- When a group is promoted to the top level it automatically becomes the current group See Section 8.3 [Promoting groups to top level], page 37.
- When a top level group is cloned it automatically becomes the current group See Section 8.5 [Cloning Entries and Groups], page 38.

The rationale here is that the appearance of a group at the top level frequently means that the group will be operated on subsequently.

4.6 Visiting Current Group Parent

The parent of the current group can be visited for current group one time operations. The following operation

```
Visit Parent of current group (keys: p)
```

is exactly equivalent to selecting the parent of the current group through the KA meta operation. This operation will report an error if the current group is a top level group, which has no parent.

Here is an example of using this operation

```
KA KO p KO d
```

It would display the current group's parent group entries. Because of the current group one time principle the following

```
KA KO p KO p d
```

would display the parent of the parent of the current group.

4.7 Last Non-Current Displayed Group

When you display any group that is not the actual current group with the "d" operation it gets recorded as the "Last Non-Current Displayed" group and keeps this status until another such non-current group is displayed. When such a group is recorded it can be more easily revisited with the operation

```
Revisit last non-current Displayed group (keys: C-d)
```

So if some non-current group g was displayed then the subsequent sequence of

```
KA <sequence that selects non-current group g> <anything>
```

is exactly equivalent to

```
KA KO C-d <anything>
```

So for example if we display a subgroup s of some non-current group g

```
KA KG g s KO d
```

then later we can execute an entry e in s with

```
KA KO C-d e
```

or do any operation like edit that entry e with

```
KA KO C-d KO e e
```

The basic idea is that if you need to display the group first to see the entries then later you shouldn't have to reenter all the keys that it took to display that group to operate on it. But this has other uses such as conveniently working on two groups at the same time.

5 Creating Groups

You can create a new top level group with the operation

```
Create a top level Group (keys: G)
```

This group becomes the current group. If you want it to be the current persistent group you can use the "C" operation, See Section 4.4 [Setting Current Group], page 22.

You can create a new subgroup in the current group with the operation

```
Create a subgroup in current Group (keys: g)
```

6 Actions and Operations

This chapter formally describes the semantics of executing actions and operations and escaping to operations and groups. It assumes that you have read the Must Read First section about Meta Keys, See Section 1.2 [Must Read First], page 2. Recall the notations for Meta Keys: KO KG and KC. We use those notations here and describe their semantics. Normally when a user types KA, Agroups prompts for an entry selection from the current group. The Meta Keys allow an escape from this.

6.1 Executing Actions and Operations

The Agroups user executes actions and operations by entering

```
KA <something>
```

where <something> can be any combination of meta keys, group and entry selections but must always terminate with an entry action selection or operation. This is not completely true since at any time during the entry of <something> the user can abort with a C-g.

You can also supply a prefix argument that will be interpreted by various actions in different ways when an action specializes on an operation such as a action specific editor or when an action entry is executed as follows

```
<prefix argument> KA <something>
```

And each action described in this manual will state whether it can take a prefix argument and what it means in that context. For example if we had a group action that inserts some text, See Section 7.3.8 [Insert Text Action], page 33, and lets say, "i" was the accelerator keys for that action. Then the following

```
ESC 10 KA i
```

would insert that text into the current buffer 10 times by virtue of the prefix argument of 10. An example of a prefix argument where an action specializes on an operation is the keyboard macro action that specializes on the edit operation, See Section 7.3.3 [Keyboard Macro Action], page 30, where it passes its prefix argument to the standard Emacs keyboard macro editor.

The <something> after the KA above by default is to select and execute an action entry in the current group. The philosophy of Agroups was to make this the least effort since the majority of time spent in an activity flow is that of executing actions in the current group. All others like operations and cross group actions are done by escaping which are described in the following sections, See Section 6.2 [Operation Escaping], page 25, See Section 6.3 [Group Escaping], page 26.

6.2 Operation Escaping

KO allows a user to escape from selecting an entry to selecting an Agroups operation. As an example you can display the entries of the current group with

```
KA KO d
```

After entering KO another entry of KO will also show the list of all operations above but will still keep prompting for an operation. So

```
KA KO KO d
```

will do the same as "KA KO d" but will first display the list of operations in the Agroups buffer as a reminder. The third and subsequent repeat of KO will scroll the Agroups buffer to make it easy to find the operation you are looking for.

You can also see a list of all operations with the operation

```
View agroups Version and operations (keys: v v)
```

This operation also shows the version of Agroups that is being used.

6.3 Group Escaping

KG allows a user to escape from selecting an entry in the current group to one operation or entry selection of another group other than the current group. For example if the current group is g1 then

```
KA KG g2 e1
```

will select and execute entry e1 in group g2. This is useful when you don't want to switch contexts but need to execute an operation or entry in some other group just once. Similarly, if you wanted to display a list of the entries in group g2 you could do that with

```
KA KG g2 KO d
```

Similar to KO a second repeated entry of KG will display all groups as a reminder before the user types the desired operation or selection. So instead of the above if you type

```
KA KG KG g2 KO d
```

You will get a list of all top level groups displayed before you select the d operation. You can also see a list of all top level groups with the operation

```
Display top level groups (keys: D)
```

Also similar to KO, the third and subsequent repeated entries of KG will scroll the display of all top level groups.

6.4 Completion versus Keys

At all times it is possible to use completion instead of accelerator keys. This applies to selecting entries, groups, and operations. Completion is used to select one of these based on the id component of an entry or operation. That is, when you use completion to select then it overrides using accelerator keys. In fact if the user does not want to use accelerator keys at all he can set the options

```
dont-ask-for-new-entry-keys
dont-ask-for-new-group-keys
```

and only use completion, See Section 13.1 [Setting Options], page 60. This is not recommended since it requires more typing, but some people may prefer to do this. A nice feature of Agroups is that even when you use accelerator keys you can *still* use completion when it's needed. This is done in one of two ways.

First, if the user starts typing and there are no accelerator keys that match what the user has typed so far then completion mode will start automatically with the keys typed so far as a beginning of an id to use completion on. For example if we type

```
KA c a t
```

and there are only entries with keys "ca1" and "ca2" then by virtue of the "t", completion mode will initiate. So for example if there is an entry with id "cats and dogs" it could easily be selected with completion at this point.

Second, a user can always force completion by entering KC. For example

```
KA KC cats SPC
```

might be used to select the entry with the "cats and dogs" id. As mentioned in other parts of this document you can change even the meta key bindings, but it is strongly advised that you do not change the binding of KC which is by default SPC. This is because SPC is used for the standard Emacs word by word completion.

6.5 Ambiguous Keys Resolution

Agroups makes no attempts to automatically force accelerator keys to be unambiguous for entries of a group or top level groups. If the user intentionally defines two entries with the same accelerator keys then he might need to use completion to get himself out of trouble. However Agroups does detect ambiguous accelerator keys when defined or moved and gives the user a chance to resolve the ambiguous keys.

As an example of accelerator keys resolution suppose that we have an entry in a group called "junk test" and with accelerator keys "jt" and then we create a new entry their called "junk text" but we make the mistake of assigning to the new entry the accelerator keys "jt" as well. Agroups would recognize the ambiguous situation and give us a chance to resolve the situation with the following prompt

```
This entry's accelerator keys (keys: j t) are ambiguous with other entry:
    junk test (keys: j t)
The following resolutions are possible:
 0 = Allow both entries to have ambiguous keys
 1 = Try entering keys for this entry again
 2 = Change other entry's keys
 3 = Replace other entry with this entry
```

This gives us the opportunity to either allow the ambiguous situation, reenter this entry's keys, change the other entry's keys, or replace the other entry with the new entry. In the case of the replace, resolution 3, the other entry is automatically moved into the zzzap group.

Lets say for the sake of illustration that at this point we choose resolution 0 so that both entries will have accelerator keys of "jt". Then we can still access the entries for some operation like editing using completion. But we can also access one of the entries by its accelerator keys. The rule is that when there are ambiguous keys the first entry in a displayed entry list will be the one selected.

One should notice at this point that since there is no terminator required for accelerator key typing that two accelerator keys don't have to be the same length to be ambiguous. In the above example two entry's accelerator keys of "jt" and "jtx" would still be ambiguous.

By default, accelerator keys resolution does not happen when the target group is the zzzap group. If don't want this you can set the `resolve-keys-to-zzzap-target` option, See Section 13.1 [Setting Options], page 60.

7 Creating Action Entries

This chapter describes the Agroups actions that can be selected as an operation which creates an entry of a specific instance of that action in the current group. Actions are accumulated into two collections: Predefined Actions and User Defined Actions. Predefined Actions are Agroups given actions. We discuss below how to access Predefined Actions and then describe the Predefined Action collection.

The user can define his own collection of actions using action templates, See Chapter 12 [Action Templates], page 46. But obviously, since some user needs to define these specific user action templates they can not be described in this document. This chapter just shows how to access user defined actions when and if they are created.

Note that some action entry instances when executed may take optional prefix arguments for doing the action multiple times or for some special effect of the action. Each action description below will discuss the effects of its prefix arguments if any.

7.1 Entering Action Data

All actions will allow a user to optionally just hit RET when asking for an entry id and will automatically generate a reasonable id.

All Agroups actions are created using action templates. In the process of selecting an operation that creates an action entry the operation expands the associated action template and it frequently asks the user to enter some data for filling a particular template slot. For example, if you were to try the predefined actions operation

```
Insert Text (keys: t)
```

it would prompt you to enter text in the Agroups buffer with the following

```
Text to insert
    ---- Edit or enter below and type C-cC-c when done ----
```

After entering the data the action template slot called "Text to insert" will be satisfied. In general after all such slots for an action are satisfied the action entry is added to the current group.

This is the common paradigm to enter and edit data both for action template expansion and entry ids. See the section Entering and Editing Data for a complete description of this paradigm, See Section 9.2 [Entering and Editing Data], page 40.

Any other kind of specialized data entry will be described in each action description in this chapter. For example some action template expansion will request data that doesn't have to be entered just selected from a list of options. And some actions use special editors for entering data. The Keyboard Macro action is an example of this.

7.2 Accessing Predefined Actions

Agroups provides a predefined collection of actions that the user can select to create an action entry with the operation

```
Add a selected Action to current group (keys: a)
```

You can see a list of these predefined actions with the operation

```
View predefined Actions (keys: v a)
```

This "a" operation is in effect a prefix key for the action operations in this list. For example, one of the predefined actions is

```
Shell command (keys: s)
```

and so if you type

```
KA KO a s
```

it will prompt for shell commands that will be executed when the resulting action entry is executed. The following section describes all of the predefined actions in detail.

7.3 Predefined Action Collection

The following sections describe the Agroups predefined action collection. The "a" operation asks the user to select a predefined action which when selected creates an action entry of that associated action.

7.3.1 File Action

The File Action operation

```
File or dired (keys: f)
```

will create an entry that given the current buffer's file or dired will return to that file or dired when the action is executed. For example if the current buffer is visiting some file then

```
KA KO a f
```

will create an entry in the current group that will return to that file when executed later. The same applies if the current buffer was a dired, it would later return to that dired.

Note that unlike the Point actions, See Section 7.3.2 [Point Actions], page 29, it will not tamper with the buffer's point when executed. This means that you can use a File Action entry to first bring up a commonly used file in a buffer and to subsequently return to that buffer as you work on that file instead of flipping between buffers.

7.3.2 Point Actions

There are two given actions that return to a point in a file

```
File Point (keys: p)
File region Point (keys: P)
```

The p operation will create an entry that given the current buffer's file and current point in that buffer will return to that point when executed.

The P operation will create an entry that given the current buffer's file and current point and mark in that buffer will return to that point when executed based on the text between point and mark. In other words, when later executed the region (defined by mark and point) of text will be searched for and the point will be placed after the region if point is after the region or before the region if point is before the region. Note that this is easy to do since C-xC-x swaps point and mark. If you make a mistake with before/after by not positioning point where you desired it then you can always edit the entry and change the before/after slot.

Both point actions have the characteristic that if a specific point action of either type (p or P) already exits in the current group with the same file then it will ask if you want to update that action entry with the new point or create a new point action entry. The former is common usage when one wants to keep moving a specific point around as a tracker.

7.3.3 Keyboard Macro Action

This operation will take the last keyboard macro, ie. the last keyboard you defined starting with start-kbd-macro, and creates an entry in the current group with that keyboard macro

```
Keyboard macro (keys: k)
```

When executed this operation will prompt for some options:

```
How keyboard macro will execute
 0 = Will just execute
 1 = Will execute if user says yes
 2 = Will load into last keyboard macro
```

Usually the default 0 choice is what you want, ie. just execute the keyboard macro when the action entry is executed. However the 1 choice is useful when the keyboard macro does a difficult to undo sequence. This choice will prompt first before actually executing the keyboard macro. The 2 choice is seldom needed, it will simply load the last keyboard macro so that you can execute it with call-last-kbd-macro.

When you execute a keyboard macro action that was created with the 0 and 1 choice, a prefix argument will repeat the macro exactly the same way a prefix argument will repeat call-last-kbd-macro.

When editing the "Keyboard macro" slot this action's template will bring up the standard Emacs keyboard macro editor and will work the same way including any prefix arguments given to the edit operation. See the Emacs Editor document for how to edit keyboard macros.

7.3.4 Keys Macro Action

The Keys macro action, which is short for Agroups Keys only macro action

```
Keys macro (keys: K)
```

is something like the Keyboard Macro action where you can create a keyboard macro that executes Agroups itself followed by some legitimate Agrouos keys. But the Keys Macro has some important differences

- All of the characters in a Keys Macro are only interpreted by Agroups and not by Emacs in general as the Keyboard Macro is.
- The keys do not have to be pre-executed as in the Keyboard Macro.
- It does not consume any Agroups command prefix arguments and thus any Agroups command prefix argument gets passed to a final referenced action or actions.
- If a Keyboard Macro action executes an Agroups command it must contains the Agroups command itself, but with the Keys Macro it is just like using Agroups so that you just specify the keys relative to the current group.

- The Keys Macro is transformed into a structure instead of just a string. A future release of Agroups will capitalize on this structure to give Keys Macros greater parameterization of how referenced entries are combined.

All of these differences are designed with the idea of referencing groups and entries, See Chapter 10 [Referencing Macros], page 42. This referencing is similar to that of symbolic links in Unix. Key Macros are not intended to supplant Keyboard Macros in any way but only to augment in this regard relative to Agroups.

A couple of simple examples will clarify this idea. Suppose that in some group g you want to reference an entry e in a subgroup of g called s. You can then create a Keys Macro entry in g that has contains simply

```
s e
```

Note that what we display here is the standard Emacs representation for displaying a sequence of keys and that whitespace is significant in a Keys Macro just like a Keyboard Macro. So the above would actually be entered as "se" with no space in between

For a little more complex example, suppose that in the global group "." you want to create a subgroup t of text insertions that are used by many groups. Then in each group where that is needed you could create a Keys Macro that contains

```
KG . t
```

where KG is the actual key-binding of KG. Then in each of these groups executing that entry would act exactly equivalent to having that `t` subgroup in each group except that modifying the subgroup `t` or any of its entries would in effect happen for all such references to `t`. The interesting thing about his example is that this would be impossible with Keyboard Macros without some sort of contrived escaping mechanism since "KG . t" is an incomplete operation. Try to create a Keyboard Macro action that tries to do this if it is not clear!

Note that if you want to create a Keys Macro action that selects an entry that does not have any accelerator keys you can do it with completion. You would do this by including a SPC to initiate completion and then an unambiguous string for the target group entry id followed by a RET. To include a RET just type `C-q RET`. For example, suppose that you wanted to create a Keys Macro action that executes an entry in the current group whose id is "My file of stuff" and suppose that "My fi" would be unambiguous. Then to create the macro we would enter in the Agroups edit buffer " My fi^M", without the double quotes, where the "^M" was the result of typing `C-q RET`. As a result this Keys Macro entry contents would be displayed as

```
SPC M y SPC f i RET
```

Much like the Keyboard Macro action the Keys Macro action gives you the option to execute or prompt but it does not give you the option to load into last keyboard macro since the Keys Macro is never a legitimate Keyboard Macro. Also the Keyboard Macro action loads its initial slot from the last keyboard macro while the Keys Macro action just pops up an Agroups edit buffer for you to enter the keys. Note that similar to the Keyboard Macro you can use a `C-q` to enter a control character in the Agroups edit buffer for the Keys Macro.

7.3.5 Elisp Actions

The following three operations

```
Evaluate ELisp (keys: l)
Evaluate ELisp from file (keys: C-l)
Elisp file to Load (keys: L)
```

add an action entry that evaluates Elisp. If you don't know what Elisp is you probably should not use the "l" and "C-l" operation actions since you may create something that will execute in error.

However if you do know Elisp then these actions give you the ultimate covering case, since anything that can be possible be done in Emacs can be done with these actions. Having said this, the intent of Agroups with its template feature is to *not* use Elisp if at all possible. If the automation that you are creating with Elisp fits a pattern that might be repeated with different parameters, you might first consider defining a user defined template so that a new action can be applied to an infinite number of specific instances and it makes it easier for other people and even easier yourself to use later, See Chapter 12 [Action Templates], page 46.

The "l" operation action evaluates an Elisp s-expression that the user specifies. For example if in the user specifies

```
Elisp s-expression to be evaluated
    ---- Edit or enter below and type C-cC-c when done ----
(message "Hello World!")
```

then on executing the resulting action entry

```
Hello World!
```

should pop up in the minibuffer. Note that the "l" operation action is asking for a *single* s-expression, so if multiple s-expressions are needed then a progn could be used.

The "L" operation action finds and opens a file of Elisp code that the user specifies, evaluates all the forms in it, and closes the file. This can be Elisp source code or byte compiled code.

Any Elisp code invoked in these actions can use the Agroups prefix arguments, See Section 12.11 [Accessing Agroups Prefix Arguments], page 58. For example if using with the "l" operation the user specified the Elisp

```
(next-line agroups-numeric-arg)
```

then when the resulting action entry was executed with a C-u prefix argument then point would be advanced 4 lines in the current buffer.

The "C-l" operation action is exactly the same as the "l" operation but the user specifies a file with the s-expression. It might seem like the "C-l" operation action is the same as the "L" operation action but it is different in two ways: First, it only evaluates one s-expression in the file just like the "l" operation action. Second, it does not load a file as the "L" operation action does. This is important to allow the "C-l" operation action to be completely symmetric with the "l" operation action since loading a file causes certain changes of context. For example some Elisp that operates on the current buffer using the "l" operation action would *not* work using the "L" operation action with that same Elisp in a file, but would work using the "C-l" operation with that same Elisp in a file. A case in point is the `next-line` Elisp example given above.

7.3.6 Compile Command Action

Emacs has a Compilation Mode paradigm for making and or compiling programs. This operation allows the user to create an action entry that uses this paradigm

```
Compile command (keys: c)
```

When this operation is selected the following choice menu is given

```
Type of compile command
 0 = A full stand alone command
 1 = A command that will be inserted in a prompt
 2 = A command that will be applied to current buffer's file
     Put a %s in command where buffer file will be substituted
```

The "0" choice will give you an action such that when executed will automatically do a command and bring up the Compilation Mode. The "1" choice will give you the same thing except that the compile command that you give will be inserted into the minibuffer when this action entry is executed so that it can be edited before the command is executed.

The "2" choice is the same as "0" but will act as a command on the file you are visiting in the current buffer when the action is executed. When you enter the command you need to put a %s in the command somewhere. Then when the action is executed the visited file will be substituted for the %s, or in other words your command will be executed on the buffer's visited file.

Once the type of file command is chosen then the action asks for the command itself. You can specify any number of command lines and unlike the Shell Command the default is to run the commands asynchronously so you do not need to put an & on the end of the last command.

Since this action uses the Emacs compile paradigm you can do the usual things, like repeated "C-x`" (next-error) to jump to the next compile error in the file where the error occurs.

7.3.7 Info File Action

This operation is used to create an action entry to bring up an Emacs Info node

```
Info File (keys: i)
```

When selected it asks for

```
Pathname of Info file to invoke
    ---- Edit or enter below and type C-cC-c when done ----
```

An absolute pathname of an Emacs Info file must be entered here. When executed then the resulting action entry will pop up the Emacs Info mode with the chosen Info file as the top node.

7.3.8 Insert Text Action

This operation is used to create an action entry that inserts some text in the current buffer at point when executed

```
Insert text (keys: t)
```

First it asks for the source of the text, either direct text insertion or to insert text from a file

```
Source of text to insert
 0 = Insert text directly
 1 = Insert text from file
```

Then it asks for the text

```
Text to insert or file to insert text from
    ---- Edit or enter below and type C-cC-c when done ----
```

If you selected 0 in the first prompt then you enter the actual text to insert here. Otherwise you enter a file pathname that will contain the text to be inserted. If the text that is going to be inserted is going to be quite long it is recommended to use the 1 option above and put the text in a file.

When executed the resulting action entry will insert the text specified above at point in the current buffer. If the executed action entry is given a numeric prefix argument N the text is inserted N times.

7.3.9 Insert Text and Position Point Action

This operation does exactly the same thing as the previous Insert Text Action operation but in addition it allow the positioning of point into the text after it is inserted into the current buffer

```
Insert text and position point (keys: T)
```

So all of the prompts are the same as the Insert Text Action except for the last one which asks where to position the point after text insertion

```
Move point back this number of characters
or the symbol 'begin' to go back to beginning of inserted text
or any Elisp expression to be evaluated after text insertion
    ---- Edit or enter below and type C-cC-c when done ----
```

To this prompt you can either enter a number N, in which case point will move N characters backward into the text after the text is inserted. Or you can enter the symbol `begin` which will cause point to be positioned at the beginning of the text. Or you can enter any arbitrary Elisp expression that will be evaluated after the text is inserted.

7.3.10 Shell Command Action

This operation is used to create an entry that runs a shell command when executed

```
Shell command (keys: s)
```

it prompts for a command with

```
Shell commands to execute
    ---- Edit or enter below and type C-cC-c when done ----
```

Note that is says "Shell commands", so any number of command lines can be entered. The shell commands follow the same rules of the Emacs command `shell-command`. By default the commands will be run synchronously. If you want them to run asynchronously then put an "&" at the end of the last command.

Normally the results of the shell commands are placed in a buffer called "*Shell Command Output*", but an Agroups prefix argument will place the results in the current buffer. This can only be done synchronously.

7.3.11 Run a program in a process Action

This operation is used to create an entry that runs a program in a process

```
Run a program in a process (keys: S)
```

The specified program process runs independently of Emacs. That is, if the Emacs process is killed the specified program process continues to run. When you first execute this operation you get the prompt

```
Program call to execute
    ---- Edit or enter below and type C-cC-c when done ----
```

where you enter a program name to execute which in most cases is an operating system command so the program name can be relative to execution search paths or an absolute file specification. You then get the prompt

```
Mode of process operation
 0 = Asynchronous: Run process in parallel with Emacs
 1 = Synchronous: Emacs waits for process to complete
```

You can select if the process is to run synchronously where Emacs waits for the process to complete or asynchronous where Emacs runs the process and continues with what it was doing. Then you get a prompt

```
The name of the buffer where the output of program should go
Leave blank if output should be ignored
    ---- Edit or enter below and type C-cC-c when done ----
```

where if you type in nothing and just type `C-cC-c` then the output from the process is ignored. If you type in a name the output from the process will be inserted in a buffer by that name.

7.4 Accessing User Defined Actions

Besides the predefined actions the user can create a collection of his own actions using action templates, See Chapter 12 [Action Templates], page 46. Given that, the user can select one of these new actions to create an action entry with the operation

```
Add a selected user Action to current group (keys: A)
```

You can see a list of user defined actions with the operation

```
View user defined Actions (keys: v A)
```

This "A" operation is in effect a prefix key for the user defined action operations in this list. Except for the fact that the user creates these new actions in this user collection of actions, the "A" operation is the same as the "a" operation.

8 Moving Entries and Groups

There are only two necessary move operations: the "m" operation for moving entries and the "M" operation for moving the current group. all the possible kinds of moving, deleting and cloning of entries and groups be can done with just these two. The local move operations are not necessary but make moving within the current group a little easier. Cloning is done, in effect, by moving an entry or group into itself. Deleting, in effect, is done by moving an entry or group to the zzzap group.

8.1 Moving entries

You can move an entry from any group to any other group or subgroup. Once again, a group itself is just another entry and so moving a group around is equivalent to moving a non-group entry. Note that moving a group entry with either the "m" or "M" operation it can moved to any other group or be promoted to a top level group, See Section 8.2 [Moving current group], page 36.

If you move an entry to the same group that it is in then instead of physically moving the entry into itself it clones the entry, See Section 8.5 [Cloning Entries and Groups], page 38. Agroups prevents the user from moving any group entry inside itself. When you move an entry to the zzzap group it implies that you intend to delete it later, See Section 3.6 [Zzzap Group], page 16.

You move an entry with the operation

```
Move an entry in current group to another group (keys: m)
```

For example

```
KA KO m x
```

would move an entry x to some group that Agroups will prompt for at this point. When an entry is moved it is always moved to a target group. The target group can be a top level group or any subgroup.

When you move an entry to a group that has a subgroup Agroups notices this and asks whether you want to move it to the group proper or a subgroup. Lets say that we are moving an entry from some group to a group called "Test" and the Test group has subgroups. We would then see

```
There are subgroups in this group: Test
 0 = Select this group as target group
 1 = Select a subgroup of this group as target group
```

If we accept 0, the default here, then the entry being moved would be moved to the target group Test. If we chose 1 here then we would asked to choose the subgroup entry as the target group. This can go on and on recursively to as many depths of subgroups that we may have.

8.2 Moving current group

You can move the current group with the operation

```
Move current group to another group (keys: M)
```

Moving a top level group to the zzzap group is the first step in removing that top level group. When a top level group is moved to itself it is cloned, See Section 8.5 [Cloning Entries and Groups], page 38. Note that this operation like all others that apply to groups, can be applied to a group other than the current group for one time, See Section 4.3 [Current Group One Time], page 21. For example typing

```
KA KG g s KO M
```

would move a subgroup s of top level group g instead of the current group.

8.3 Promoting groups to top level

Whether you move any group or the current group using the "m" or "M" operation respectively, if that group is a subgroup then you will have the option of moving that subgroup to another group or promoting that subgroup as a top level group. Agroups automatically detects that you are moving a subgroup and gives you the prompt

```
You have elected to move a nested group entry.  There are two options:
 0 = Move this group entry to another group
 1 = Promote this group entry as a top level group
```

If you choose option 1 and promote the group as a top level group Agroups assumes that you want to do something with it and so makes it the current group.

8.4 Local moves

The two local move operations are

```
Move Locally an entry in current group (keys: l)
Move Locally an entry in current group to its parent (keys: L)
```

As mentioned above these local move operations are not necessary and can be done with regular move operations, but they make it easier to do local moves within the current group. The "l" operation moves an entry in the current group into the current group. So it makes it easy to move an entry in the current group into a subgroup of the current group or clone an entry in the current group.

The "L" operation will move an entry in the current group up to the parent group of the current group. Note that as with the regular move operations only a group type entry can be promoted to a top level group. Also note for the "L" operation the Current Group One Time concept applies here, See Section 4.3 [Current Group One Time], page 21. So for example, you can use the "L" operation to move an entry in a current group subgroup up to the current group, specifically

```
KA s KO L e
```

would move an entry "e" from inside the subgroup "s" in the current group up to the current group.

8.5 Cloning Entries and Groups

In the previous sections on moving entries and groups we mentioned that when an entry is moved to the same group that it is in or a top level group moved to itself, then it is cloned. When an entry is cloned in this way the clone becomes an entry of the same group as the original. It follows that when a top level group is cloned, the clone group becomes another top level group. When a top level group is cloned the cloned group becomes the current group.

Cloning entries is essentially the process of making an exact copy of an entry or group. Sometimes this is a very useful thing to do since frequently the user wants exactly the same action with just some minor differences. It is also useful to save an entry before experimenting with its original. Cloning a group can be useful for various things also. For example when breaking up a group into two groups it may be easier to clone the group and edit out pieces of each. As usual, in other Agroups capabilities, it is up to the user as how to get creative with cloning.

When an entry or group is cloned, the clone will prompt for an new id with the original id plus the word CLONE at the end as a default. You can just accept the default or edit the default id in the minibuffer. Agroups will also ask you for a new accelerator keys component, since it assumes that you want to have distinct keys between the clone and the original. All other components in the clone entry are exactly copies of the original components. And when cloning groups all sub-entries are copied recursively.

9 Editing Entries and Groups

Once you create entries and groups you may not be happy with one of the components. Agroups has an Editor and edit operations that you can use to both look at the details of entries or groups and edit them. When creating new entries action template expansion may also use the Agroups editor to fill slots that need data from the user. Special editors for certain slot data can also be invoked.

9.1 Editing Operations

There are three operations for editing

```
Edit an entry (keys: e)
Edit current group (keys: E)
Edit last edited or added entry (keys: C-e)
```

The "e" operation will ask the user to select an entry from the current group to edit. The "E" operation will edit the current group. The "C-e" operation will re-edit the last entry you edited or the last entry, subgroup or clone that was added. Re-editing is useful when you want to continuously edit components of the same entry or prototyping an entry.

When you edit an entry or group the Agroups editor asks you to choose one of the components of the entry to edit, See Section 3.4 [Entry Components], page 15. Thus, when you edit an entry Agroups displays the group that the entry is in followed by the components of that entry and prompts you with

```
Enter to edit: id: i, keys: k, object: RET, exit: SPC:
```

where each of these choices represents one of the components. SPC just exits the edit mode so enter SPC if you just wanted to view the contents of the entry or skip editing it. Selecting RET will edit the object of the entry which we say more about later.

When you select to edit the id with "i" the Agroups editor brings up an Agroups edit buffer with the current id and waits for you to edit and save it. See the section Entering and Editing Data for how to do this, See Section 9.2 [Entering and Editing Data], page 40.

When you select to edit the keys with "k" Agroups prompts you in the minibuffer to enter one or more keys terminated by a RET. Note that there is no way to edit this minibuffer since, except for C-g and RET anything is possible to type for keys. For example a C-b normally bound to backward-char would just be accepted as a legitimate key. So if you make an error in typing the keys you just have to abort and retype the keys or accept and re-edit the keys.

When you edit a non-group entry the Agroups editor displays the object component of that entry as

```
object: This is a template based entry with slot values:
```

followed by a list of the description of slots with their values. When the user selects to edit the object with RET the Agroups template editor will list and ask for a slot to edit. When you select the slot Agroups then pops up an Agroups edit buffer with the value of the slot in that buffer for editing. See the section Entering and Editing Data for how this is done, See Section 9.2 [Entering and Editing Data], page 40.

Editing the object of a group entry is equivalent to the process of selecting and editing an entry in that group. This is evident when the object of the group entry is displayed

```
object: This is a group entry who's object is more entries
        Editing this object will select and edit one of those entries
```

9.2 Entering and Editing Data

When you edit entry component data the Agroups editor uses a simple and consistent paradigm. This section describes this data entry paradigm. Templates also use the Agroups editor for entering data and thus the paradigm outlined in this section applies to editing and entering the template data as well.

As an example, when you edit the id component of an entry an Agroups edit buffer pops up with the following

```
Entry id (one line only)
    ---- Edit or enter below and type C-cC-c when done ----
```

This is the common buffer layout for both editing action components and template expansion that needs similar kinds of data. The first lines in this buffer ending and including the line with the `C-cC-c` prompt are called the header. The first line in the header will start the description of the kind of data it is expecting you to enter or edit and varies from one action component to another. In this particular component it suggests you enter one line only for the id since ids of one line look better and are easier to deal with and if you enter more than one line Agroups will truncate the id to the first line. The object component slot data though could have any number of header lines to describe the data to be entered and allow any number of user entered lines after the `C-cC-c` prompt line.

The completion directive line with the "type C-cC-c" is always the same and directs you to type `C-cC-c` when you are done entering or editing your data. You should not tamper with these header lines. You simply start entering and editing your data after the completion directive line and finish with a `C-cC-c`.

In the case of editing an existing object component slot, the lines following the directive line will reflect the existing slot value. The mode of the Agroups edit buffer will also reflect the type of data there. For example if it is text it will be in text-mode, if it is Elisp it will be in lisp-interaction-mode.

In the event that you do not want what you have done so far in an entry component edit buffer then you can just ignore or kill this buffer. This is always safe since entering or editing entry component data never takes any action until it has all of its data.

It is a general rule of thumb that putting a RET at the end of data entered this way is usually not a concern unless that return will be used in some way by the user actually executing the entry action. For example in the Insert Text action if the user wants a final RET in his data when inserted into a buffer he will put it there otherwise he would not. And in the above case of "Entry id (one line only)" the editor constrains the input to one line so any RET or second or more lines will be ignored. We say "usually not a concern" since in the space of user defined action templates anything is possible, however a well designed template will make this transparent.

9.3 Special Slot Editors

Every slot in a template does not have to be edited by the standard Agroups editor paradigm discussed in the previous section. Action templates allow the designer to specify

a special editor for chosen template slots. When the user chooses to edit a particular action object slot that had such a special slot editor specified Agroups will automatically send that slot value to the special editor and then when the special editor paradigm completion takes place the resulting value will be placed in the slot just like in the standard editor paradigm.

The given actions in this document will specify when and where a special editor is used. The Keyboard Macro action is a typical example of this. Editing the Keyboard Macro action entry slot for the macro's keys string will bring up the Emacs keyboard macro editor on that slot, See Section 7.3.3 [Keyboard Macro Action], page 30.

9.4 Setting Entry Actions

All components of an entry can be edited except the action component. This is deliberate. Allowing the user to edit entry actions would cause chaos since there is little chance that any two given actions are congruent, ie. that all of their slots match in type and number. However, in very special circumstances Agroups allows a controlled form of doing this to some extent.

Agroups provides an operation for setting all entry actions of an existing action in a user's Agroups save file to another action

```
Set save file Actions (experts only) (keys: s A)
```

It will prompt for an existing action and then a new action to replace it with. The existing action can be any possible action name except the "group" action.

This operation should only be done by experts or on the advice of experts since the unwary user can create bogus entries in his Agroups save file by using this operation. It should only be done where the new action is congruent to the existing action. For example, if the new action is just a renaming of an existing one they would be congruent. Most use of this operation is by user template designers who have to change the name of an action in their template for some reason.

This operation will replace all occurrences of an existing action name with of a new action name in the users Agroups save file. If it finds any occurrences of the old action it will first save the user's save file in a file called

```
<save file name>-with-<old action name>
```

and then do the replacements; possibly needed to revert back to if a mistake was made in performing this operation.

10 Referencing Macros

There is an interesting synergy between Agroups and Emacs keyboard macros. Keyboard macros extend Agroups in a way that we discuss in this chapter and Agroups extends keyboard macros by giving keyboard macros a convenient execution repository. This is no accident and it was part of the design approach of Agroups to capitalize on this synergy along with keys macros to complete the Agroups concept, See Section 7.3.3 [Keyboard Macro Action], page 30, See Section 7.3.4 [Keys Macro Action], page 30.

The way that Keyboard Macro actions and Keys Macro actions extend Agroups in this regard is to allow the user to create entries that reference other Agroups entries . We refer to these actions when used in this way collectively as *referencing macros*. There are a few ways that referencing can be used

1. Referencing macro action entries can reference other groups and entries.
2. Referencing macro action entries can combine other action entries of different action types.
3. Referencing macro action entries can reference other referencing macro action entries and hence can be infinitely nested.

The second and third are actually corollaries of the first but are listed since they imply useful concepts in their own right. Notice that the first is true of groups in a general sense which means referencing any group or subgroup. Here is an examples to help illustrate this. Suppose that we create a keyboard macro action whose object keyboard macro slot is composed of

```
KA s a
```

Where KA is replaced by its key binding. What does it do? It executes an entry `a` in a subgroup `s` of the current group. Let say that the current group in this case was group `g` and we wanted other groups to be able to execute this action when those other groups are the current group. This can be done by making this entry absolute as follows

```
KA KG g s a
```

Now, no matter what group this entry is executed from it will first go absolutely to top level group `g` then to its subgroup `s` and then execute action `a` in that subgroup. Using the keys macro action instead we could more simply do the same thing with the macro

```
s a
```

then whether this entry is executed from its containing group or any other groups it would reference *relatively* within its containing group, in this case being the entry `a` in subgroup `s` of top level group `g`. Note that this would not work as a keyboard macro action since `s` is only a key binding recognized by Agroups in this particular case and intent.

We could also use the keys macro action to reference another group absolutely. Lets says that in another group other than group `g` we wanted to reference this action `a`. We could create the keys macro action in that group as

```
KG g s a
```

Now lets get back to illustrating the ways that referencing can be used. This keys macro action illustrates all three

```
s a s b
```

This first is that it references action entries in other groups namely subgroup `s`. The second is that it combines the actions `a` and `b`. And the third is that since actions `a` or `b` could be yet other keys macro actions it illustrates a potential infinite nesting of referencing.

The biggest problem with such referencing macros is that if one wants to modify the key bindings of an entry then those entries that reference such a modified entry would not work until edited to fix the key references. In a future release of Agroups there will be automatic detection of such points in referencing macros wherever any Agroups key binding is changed. It is known that this can be done for keys macro action entries but it is not clear yet if this can be done for keyboard macro action entries in a general way. In the mean time it may be wise to use keys macro actions for most referencing.

11 Regular expression operations

Agroups has the capability of applying a regular expression to the entry ids of the current group to describe a subset. The regular expression can then be applied to an Agroups operation that operates on one or more entries, but in the case of the regular expression operates only on the subset described by the regular expression. For example the user could specify a regular expression to display all entries in a group that match that regular expression or use a regular expression to move all matched entries in a group to someplace else.

We use the Emacs abbreviation of "regexp" in this chapter where a regular expression needs to be formed by the user. If you don't know how to form regular expressions or what they mean you should read the chapter in the Emacs manual on Regular Expressions.

11.1 Apply regexp to operation

The following operation allows you to create a regexp and apply it to some operation

```
Apply regular eXpression to operation (keys: x)
```

This works as sort of an operation prefix on some other operation

```
KA KO x <regexp> <some operation>
```

where the user enters the regexp followed by a RET and then some operation. That operation will then be applied to the subset of entries described by the regexp applied to the ids of the entries of the current group. For example while

```
KA KO d
```

would display all of the entries in the current group, the following

```
KA KO x  [Ww]ork RET d
```

would display the subset of all those entries with the words "Work" or "work" in their ids. In a similar way the keys

```
KA KO x  [Ww]ork RET m
```

would move that subset of entries to a selected group.

11.2 Apply last regexp to operation

The previous section describes how to apply a regexp to the entry ids of the current group for some operation. When this is done that regexp is recorded as the "last regular expression". The last regular expression can then be used with the operation

```
Apply last regular eXpression to operation (keys: C-x)
```

It works exactly the same way as the x operation in the previous section but the regexp does not have to be reentered. For example if, as in the previous section, the user typed

```
KA KO x  [Ww]ork RET d
```

to display the subset of entries that match the regexp "[Ww]ork" then subsequently the user could type simply

```
KA KO C-x m
```

to move that subset to another group,

The last regexp expression keeps its status until another regexp is entered with the x operation described in the previous section.

11.3 Specifics of regexp operations

Although applying regexps to operations is pretty straight forward some things need to be mentioned about some of the specifics for various operations.

On move operations when a subset of entries is moved they can never be moved to the top level even if there are groups in that subset. Also when a regexp subset is moved no ambiguous keys resolution is performed on the target group, See Section 6.5 [Ambiguous Keys Resolution], page 27.

On the edit entry operation the regexp subset of entries will be asked to be edited successively. You can abort out of this editing loop with a C-g as usual.

12 Action Templates

Action Templates are declarative data constructs that reduce the amount of work to create new Agroups actions. They are considered the extensibility feature of Agroups. All Agroups predefined and user defined actions are implemented with Action Templates. Hopefully the real power of action templates is realized in user defined templates. This chapter describes how a user can create new action templates and hence add to the user collection of general actions.

To create new action templates you should know something about Elisp. It is possible to know a little about Elisp and learn as you go. It will be helpful if you actually try the examples in this chapter to see the effect and get some facility with both the form of templates and some Elisp.

In some of the sections that follow, for precision we use dot notation to describe various forms. So as not to cause confusion for the new Lisp user we just remind that the dotted list (x . (a b c)) is equivalent to the list (x a b c).

12.1 Simple Action Template Example

In this section we show an extremely simple and probably useless action template just to spark interest and get us started. This will be an action template that has one slot: some text to display as a message in the minibuffer. Here is this simple template that creates a new action called simple-message

```
(agroups-add-action-template
 '((action simple-message)
   (id "Simple Message")
   (keys "sm")
   (afun (message text))
   (slots (text "Text of message"))))
```

The function `agroups-add-action-template` is used to create new actions and is given a single argument being a template. The template is a declarative data structure represented as a list of properties. In this simple example the properties are: `action, id, key, afun` and `slots`. We will go through these one by one to see what they mean in the next section. But first to get a more overall picture of what this does, evaluate the above `agroups-add-action-template`. After evaluating this template see if it has been added by entering

```
KA KO v A
```

and note that is a capital A. You should be able to see this registered in the user action collection in the Agroups buffer

```
User defined actions
  Simple Message (keys: s m)
```

At this point for all practical purposes this now looks like any of the given actions in the Predefined Action collection. Furthermore we should now be able to create an entry of this action in the current group. So lets do this with

```
KA KO A s m
```

and again note that this is a capital A. When Agroups prompts for the "Text of message" slot enter

```
Text of message
    ---- Edit or enter below and type C-cC-c when done ----
Hello World
```

but don't put a RET after the text "Hello World". Then type `C-cC-c` to enter the slot value and give it whatever id and keys you want. Now execute the action and you should see

```
Hello World
```

in the minibuffer. You are now a creator of new Agroups actions!

12.2 Simple Action Template Properties

As promised in the previous section Simple Action Template Example we will now go through each of the properties in the template we created. Just to recall our simple action template

```
(agroups-add-action-template
 '((action simple-message)
   (id "Simple Message")
   (keys "sm")
   (afun (message text))
   (slots (text "Text of message"))))
```

Actually the only property that is required is `action` and all others are optional and serve to make more and more complex actions. For example you could create an action with nothing more than

```
(agroups-add-action-template
 '((action not-much)))
```

Of course if you have the time to experiment you will find that the action is well named, ie. what it does is not-much!

The `action` property makes the action distinct from any other and is needed behind the scenes to indicate an action of this type. This action symbol will appear in the Agroups save file when the simple-message action entry was created. If you were to look you would see that it appears as u-simple-message instead of simple-message. Agroups given actions are guaranteed to never begin with "u-" which stands for User defined. Agroups adds this prefix to every user defined template action and does the book keeping to maintain this so that the user never has to worry about creating an action that clashes with any predefined actions.

The `id` property is a short one line identifying description of the action. We say "short one line" since this will appear in the user defined actions list that you saw above. We say "identifying" because the id is the string that completion will work on if the user chooses to use completion instead of the accelerator keys you specify. And the `keys` property is the accelerator keys you specify.

The `afun` property is the action function that gets applied to the slot variables. So in the simple-message action that we defined we had

```
(afun (message text))
```

The value of the `afun` property is more or less what you would expect with an ordinary Elisp function call on variables. In this example the function `message` is applied to the

variable `text`. We say "more or less what you would expect" because instead of just allowing variables as arguments you can have any kind of list structure or structures where the slot variables get substituted on entry creation. For example if the `message` function allowed a list of text strings instead of just a text string, we could have specified

```
(afun (message (text)))
```

Note that the list `(text)` is not quoted. No part of any nested list structures have to be quoted since nothing in such lists get evaluated. Just the variables are substituted for values.

The substitution of the variable values brings us to the `slots` property. The whole purpose of the slots property is to provide a list of variables that will be bound to the values determined by a specific entry instance of the action template. So our simple-message action template we have

```
(slots (text "Text of message"))
```

and this defined the variable `text` that on action entry creation gets a value that is bound to the variable `text` in the above `afun message` form. In our further example of a user creating a simple-message entry the `text` slot variable got bound to the string value "Hello World".

That completes the simple understanding of action templates. The rest of the capabilities of action templates are achieved through more of these action templates properties and slot properties. They expand on those additional properties. But first, in the next section we give a formal but brief description of the full syntax the top level action template form and in subsequent sections we give the full syntax and semantics of each template property form.

12.3 Form of Action Templates

We saw in the previous section that the function

```
agroups-add-action-template
```

takes an action template as an argument and adds that in the form of an action to the user defined collection of actions. This template argument itself is quoted so that the template can be passed as a variable. Even though we quote the top level template list structure, none of the nested lists or symbols need to be quoted. An exception would be if the template designer explicitly wants a quote for some reason.

The action template itself is simply a list of properties of the form

```
(<property 1> <property 2> ... <property n>)
```

Each property is a list of the form

```
(<property name>  <value 1> <value 2> ... <value n>)
```

where <property name> is a Template Property symbol listed below and the <value n> are the specific values that make this template distinct from other templates. When we use just the <property name> to refer to the property in this document it is meant to connote the entire property above with property name and values.

The only required property is the `action` property all others are optional and have defaults. The following table lists all template properties and their defaults on the right:

```
  Template Property            Default
  ------------------------------------------------------------
* action              No default, required
* id                  An id would be constructed from action
* keys                No keys, user would have to use completion
* afun                No afun, action entry will not execute
* slots               No data for template, all entries the same
  pfun                No preprocessing before template expansion
  exists              No existing condition check
  default-slot        Default slot when user hits RET is first slot
```

Although the only property required is the `action` property all good action templates should have the properties marked with * above. The syntax and semantics of each of these is described in the following sections.

12.4 action Template Property

The action template property is a list of the form

```
(action <action>)
```

Where <action> should be a symbol (or list see below) unique from other user action symbols unless the enclosing user template is intended to replace another existing template with the same action symbol. In the context of the action template we refer to the action symbol as simply "the action". The action is what distinguishes one action from another in the collection of all Agroups actions.

When the template is added to the system, Agroups changes this symbol to u-<action>. The reason for this is so that the user does to not have to worry about clashes with Agroups given actions. Agroups given action symbols are guaranteed not to begin with "u-".

It is also possible for <action> to be a list of action symbols and in this case all rules above apply to each symbol in this list. For example the Agroups given "File or dired" action is specified in its template as

```
(action (file dired))
```

When such an action template is added with a list of actions it must have a `pfun` property to set the action, See Section 12.8 [pfun Template Property], page 56. Usually this is done by the `pfun` function based on the current environment and most likely the current buffer. For example, the Agroups given "File or dired" action sets the action of the user created entry based on whether the buffer is a dired or a buffer associated with a file. If no `pfun` property is specified or a `pfun` function does not set the action then Agroups will just assume the first symbol of the list is the action.

12.5 id Template Property

The id template property is a list of the form

```
(id <descriptive string>)
```

where <descriptive string> is a short string designed to fit on one line describing the action. There should be no line terminators in this string. If it is more than one line Agroups will just take the first line of the string without the terminators. It should be

descriptive but relatively short since it appears in action lists and edit buffers. This is also the string that will be used by the Agroups user when using completion on an action id.

12.6 afun Template Property

The term "afun" means action function. The action function is the function that brings about the final result of executing an action entry. The `afun` template property has the form

```
(afun <function spec>)
```

Where <function spec> is what you would normally expect with a function call. So the form of <function spec> is

```
(<function> . <s-expression list>)
```

where <function> is any function symbol or lambda. Anywhere in the <s-expression list> a slots property slot symbol may appear which results in that symbol being substituted for the slot values when the action is expended into an entry. There is no evaluation either before or after the slot variable substitutions. Then only the substituted s-expressions in the <s-expression list> along with the action symbol are placed in the user's newly created entry instance. When that entry instance is executed the <function> is looked up based on the action in the entry instance and applied to the instance s-expressions.

The no evaluation before or after clause above indicates that nothing in the <s-expression list> *needs* to be quoted. However there are cases where the template designer might want to include quotes. An example is where the <function> is `eval` and the <s-expression list> is just a single s-expression containing a necessarily quoted entity.

To illustrate implications of all of the above we give two action templates that would have the same action entry execution effect but would create different entries in the Agroups save file. Then we make several specific points about these action templates that follow from the above general description.

The action template

```
(agroups-add-action-template
 '((action compose-message)
   (afun (message "%s %s" first second))
   (slots
    (first "First part of message")
    (second "Second part of message"))))
```

would have the same action entry execution effect as

```
(defun myfunction (first second) (message "%s %s" (car first) second))

(agroups-add-action-template
 '((action compose-message)
   (afun (myfunction (first) second))
   (slots
    (first "First part of message")
    (second "Second part of message"))))
```

Lets say for the sake of illustration that the user creates two entries from these action templates and in both cases enters the date "Hello" for the first slot and "World" for the

second slot. Then both of those entries when executed will result in the message "Hello World" being displayed in the minibuffer. It might help to actually try adding these two templates and then two corresponding action entries to see what gets generated in the action entries themselves that you can see in the save file with "KA KO v s". Now for the specific points:

[point 1: `afun` s-expressions can be any s-expression]

In the second version of the compose-message action the `afun` spec

```
(afun (myfunction (first) second))
```

is kind of silly because all that the myfunction does is take the car of the first argument which appears in the entry as the list ("Hello"). The following would make more sense in this case

```
(afun (myfunction1 first second))
```

but the more awkward myfunction version helps visualize the idea of substituting the slot variable `first` in the <s-expression list> form.

[point 2: The `afun` function is any function symbol or lambda]

The first version of the compose-message action used a standard existing Elisp function `message` and the second version uses a user defined function `myfunction`. Also note that we can use a lamda instead of `myfunction`

```
(agroups-add-action-template
 '((action compose-message)
   (afun ((lambda (first second) (message "%s %s" (car first) second))
          (first) second))
   (slots
    (first "First part of message")
    (second "Second part of message"))))
```

[point 3: no evaluation either before or after the slot variable substitutions]

In the version

```
(afun (myfunction (first) second))
```

the resulting user entry appears as

```
(... compose-message ("Hello") "World")
```

which shows that neither (first) nor ("Hello") was evaluated, just the slot variable `first` was substituted with the value "Hello" and the variable `second` was substituted with the value "World".

[point 4: only the substituted s-expressions in the <s-expression list> along with the action symbol are placed in the user's newly created entry instance]

Generally speaking it is better to use an <s-expression list> that is as least complicated as possible. So `myfunction1` function <s-expression list> is better than the `myfunction` function <s-expression list> which is better than the message <s-expression list>. You can see this by the lists appearing in each user entry instance for each function version:

```
message version:  (... compose-message "%s %s" "Hello" "World")
myfunction version:  (... compose-message ("Hello") "World")
myfunction1 version:  (... compose-message "Hello" "World")
```

The myfunction1 version not only has the least amount of data in the user's entry collection but it is easier to change or add to the effects of the action later without having

the user to recreate his entries derived from the action. In fact the message version is horrible because the "%s %s" string would occur in every user's entry needlessly.

[point 5 <function> is looked up based on the action in the entry instance and applied to the instance s-expressions]

You can see from the entry instances listed in point 4 that nowhere does the actual action function appear an the entry instance. However note that the s-expression lists are quite different. This means that without the user having to change his entry instances of your action you can change the function but you can't change the topology of the s-expression lists. This is yet another reason in support point 4 that the less complex list of s-expressions is better.

12.7 slots Template Property

The slots template property is the heart of the template since the variety of values of the slots will allow one general action template to provide an infinite variety of specific action entry instances to the Agroups user.

The form of the slots property is a list of the form

```
(slots . <slots list>)
```

Where <slots list> is a list of zero or more individual slot forms. You can actually specify no slots but that would not be a very useful template since every action entry generated from such a template of the same action would do the same thing. The section in this document on the `afun` property describes how slots are actually used when an action entry is executed, See Section 12.6 [afun Template Property], page 50.

The subsections below expand on the individual slot form. The Form of a Template Slot subsection describes the general syntax and semantics of a slot form and subsequent subsections describe the specific slot properties.

12.7.1 Form of a Template Slot

An individual template slot is a list of the form

```
(<slot symbol> <slot id> . <slot properties>)
```

The <slot symbol> is a symbol that should appear in the `afun` form, See Section 12.6 [afun Template Property], page 50. A specific value will be substituted for the <slot symbol> in the `afun` form when the template is expanded to create an entry.

The <slot id> is either a short but descriptive single string or if more description is necessary a list of strings. This string or strings should not have any kind of line terminators. They will automatically be interpreted as one line or multiple lines. The <slot id> will appear to the user when creating or editing an action entry slot.

The <slot properties> is a list of zero or more slot properties that we will describe in subsequent subsections. If a slot form specifies zero slot properties then Agroups defaults will be chosen. The following table lists all slot properties with their defaults on the right:

```
Slot Property              Default
-------------------------------------------------------------
type            Slot data type will be of type string
select          Slot data will not be selected by user but entered
sfun            No slot data function, user will be asked for data
editor          Default Agroups slot editor
printer         Default Agroups slot data printer based on type
```

In the following subsections that have specific descriptions of these slot properties each property applies to the enclosing slot form that we defined at the beginning of this section. In particular it applies to how the slot data is acquired and manipulated.

12.7.2 type Template Slot Property

The `type` template slot property will indicate the Elisp data type of the expected value of the slot data. The type template slot property has the form

```
(type <Elisp type symbol or t>)
```

Where <Elisp type symbol or t> is an unquoted symbol. Elisp data types are permissible such as `string`, `number` ... etc. with the addition of the Agroups special type `t` which means any Elisp data type. The default if no type slot property is specified is `string`.

The type property is used by the default Agroups editor and printer so that it knows how to edit or present the data. Also the `afun` function is guaranteed to receive the type of data that is specified for a slot or `string` if not specified.

12.7.3 select Template Slot Property

The `select` template slot property allows the template designer to prompt the user for some number of fixed values with descriptions when creating or editing the slot. When the user chooses a value it becomes the slot value. It has the form

```
(select . <list of select choices>)
```

where <list of select choices> is a list select choices of the form

```
(<choice description> <value>)
```

The <choice description> is a string or list of strings exactly like the <slot id> above and the same rules apply. The value must be the type specified by the type slot property or if type property is not specified a string. Here is an example

```
(agroups-add-action-template
 '((action select-message)
   (afun (message text))
   (slots
    (text "Text of message"
     (select ("Message says Hello" "Hello")
             ("Message says Goodbye" "Goodbye"))))))
```

Note that no type is specified for the text slot. Since the default is the type `string` the select data "Hello" and "Goodbye" are not in error since they are strings. If instead we had the symbols `hello` and `goodbye` as the select values in the above

```
(select ("Message says Hello" hello)
        ("Message says Goodbye" goodbye))
```

Then when expanding that template Agroups would give us, the template designers, an error

```
Error: Template slot: text is not of type: string, it has value: hello
```

We could fix this by adding a (type symbol) property but then the user would get an error when trying to execute an entry generated from our template since the function `message` takes a string and not a symbol and it would then complain. So here is a similar example that uses symbols as values but uses `eval` as an action function with some code that uses the selected select symbol

```
(agroups-add-action-template
 '((action select-message)
   (afun (eval (message (symbol-name 'sym))))
   (slots
    (sym "Text of message"
     (type symbol)
     (select ("Message says Hello" hello)
             ("Message says Goodbye" goodbye))))))
```

Once again however this is a poor action since what it would put in the user's entry is too complex and specific. Template designer written functions are usually better for processing select values like numbers, symbols lists etc.. So in the above case

```
(afun (somefunction sym))
```

would be much better where `somefunction` processes the selected value of sym in some way and the user's entry for example would be simply

```
(... select-message hello)
```

which is far less complex and easier to deal with in the future than

```
(... select-message (message (symbol-name (quote hello))))
```

12.7.4 sfun Template Slot Property

The `sfun` template slot property which stands for Slot Value Function will return a value that becomes the value of a slot based on the environment at action entry creation time. This overrides any other method for getting the slot value, such as a `select` template slot property. However the `sfun` property is ignored when editing the slot value. It has the form

```
(sfun <slot value function>)
```

where <slot value function> is a function symbol or lambda. For example the following

```
(defun buffer-region ()
  (buffer-substring (region-beginning) (region-end)))

(agroups-add-action-template
 '((action region-message)
   (afun (message text))
   (slots (text "Text of message" (sfun buffer-region)))))
```

sets the text slot of a new action entry to the current buffer's region. But then when the user edits such an entry the `sfun` would be ignored and the existing text slot would be place in the standard Agroups edit buffer for editing.

12.7.5 editor Template Slot Property

The `editor` template slot property is used when the standard Agroups editor is not suitable for a slot and a special editor would be better. It has the form

```
(editor <editor function>)
```

where <editor function> is a function symbol or lambda representing the function that Agroups will pass as the single argument to <editor function> the value of the slot to edit. This function must return the edited value. For example the Agroups given keyboard macro action template stores the keyboard macro string in a slot called `macro` that uses the Standard Emacs keyboard macro editor

```
(macro "Keyboard macro" (editor <kbd-macro editor function>))
```

Agroups prefix arguments can be passed to this editor using the Agroups argument variables, See Section 12.11 [Accessing Agroups Prefix Arguments], page 58. In fact the Agroups given keyboard macro action does this and in the above the <kbd-macro editor function> is a function that calls the Emacs keyboard macro editor and passed the Agroups prefix argument to it.

12.7.6 printer Template Slot Property

Normally Agroups presents the current value of a slot in the standard Agroups editor by printing the value in the Agroups buffer according to the type of the slot data. For example if the slot type is `string` it just prints the string verbatim, but if the type is `t` then it pretty prints the expected Elisp.

The `printer` template slot property allows this behavior to be overridden by the template designers printer. It has the form

```
(printer <print function>)
```

where <print function> is a function symbol or lambda that prints the slot data and takes as an argument the slot data.

12.7.7 edit-printer Template Slot Property

There are two ways that Agroups prints a template slot value in the standard Agroups editor. It prints the slot value in an Agroups buffer when displaying the whole entry object to be edited and also when editing the individual slot it places the existing slot value in the Agroups edit buffer by printing it. Usually these get printed the same way. But occasionally those two get printed in different ways. For example the Keys Macro action prints the keys in a the whole entry object view as standard Emacs keys display form. But this form is not suitable for editing the actually slot keys, the standard display form has embedded spaces for example.

The default printer or if specified for the slot the `printer` template slot property will be used for both ways unless the `edit-printer` template slot property is specified. When this property is specified it will be used for printing into the edit buffer of the individual slot where the user will actually edit it. It has the form

```
(edit-printer <print function>)
```

where <print function> is a function symbol or lambda that prints the slot data and takes as an argument the slot data.

12.7.8 edit-receiver Template Slot Property

When the user completes a slot edit by typing `C-cC-c` the Agroups edit buffer is parsed for the entered data based on the default or specified type of the slot. If the `edit-receiver` template slot property is specified for the slot then the parsed resulting object will be passed to the `edit-receiver` function. This slot property has the form

```
(edit-receiver <receiver function>)
```

where <receiver function> is the receiver function that takes one argument of the parsed object and returns the object that will be stored and associated with the slot in the user's save file. Usually this works in conjunction with the `edit-printer` template slot property whose function converts the stored representation back to the form that the receiver function gets before the printer function prints it.

12.8 pfun Template Property

The `pfun` template property specifies a preprocessing function that gets called before template slot expansion takes place. This function can set action attributes and slot values before template expansion. The common use for this is to collect information about the current environment, like the current buffer, and set that information as the entry data when an entry is created.

The `pfun` property has the form

```
(pfun <preprocessing function>)
```

where <preprocessing function> is a function symbol or lambda. This function is run before any slot processing is done and it can optionally set the template action and any of the slots using the functions

```
agroups-set-entry-attribute
agroups-set-entry-slot
```

The agroups-set-entry-attribute function call has the form

```
(agroups-set-entry-attribute <entry attribute> <value>)
```

where <entry attribute> is one of 'action or 'entry-id and <value> is a symbol in the case of 'action and a string in the case of 'entry-id.

Use 'action when you want to preset the action of an entry. This is almost exclusively used when the template action property specifies a list of two or more actions. For example the "Buffer associated File or dired" action has template that specifies two actions

```
(action (file dired))
```

Then in the `pfun` function that it specifies it sets the action depending on whether the current buffer has a file associated with it or a dired. If the template was specified with multiple actions and does not specify a `pfun` that sets one of these actions then Agroups will just use the first action in the list of actions when an action entry is generated.

Use 'entry-id when you want to preset the default id of an entry. Normally when the user is prompted for an entry id and just hits RET an Agroups default id is constructed from the action symbol, but if the entry-id attribute is set by the `pfun` then that will be used instead of the Agroups default.

The agroups-set-entry-slot function call has the form

```
(agroups-set-entry-slot <slot symbol> <slot value>)
```

where <slot symbol> is a quoted symbol matching one of the slot symbols of one of the slots of the template and <value> is an Elisp value of the type specified in that template slot. The template designer can preset a slot value using a `pfun` in this way or an `sfun`. The only difference is that it may be more convenient to use the `pfun` method to set a slot value in circumstances where the action is also computed by the `pfun` or a number of slots share the same computations.

12.9 exists Template Property

The exists template property is used to trigger something when an one or more entries exist in the current group with the template action that specifies the exists property and some existing environment condition exists. It is typically used when creating a new action entry and some entry already exists of the same action and value of one or more template slots. When a template specifies the exists property and such a condition exists the user is given the option of triggering something that the exists property specifies. If the user accepts and then there are more than one such existing entry then the user is given a choice of those entries.

The exists template property has the form

```
(exists <exists predicate> <exists trigger>)
```

The <exists predicate> is a function call form that returns non-nil if the condition exists. The form of the <exists predicate> is

```
(<function> . <slot symbol list>)
```

where <function> is any function symbol or lambda and <slot symbol list> is zero or more <slot symbol> symbols that appear in the template slots property.

The <exists trigger> for the time being has only one form

```
(update . <slot symbol list>)
```

where <slot symbol list> is a list of one or more slot symbols in the template slots property. The idea is that when the <exists predicate> returns non-nil an update of the existing slots gets triggered. In other words instead of creating a new entry the chosen existing entry slots get updated based on the specified slot's value properties.

A good example of an action using the exists property is the given Agroups "File Point" action. When that operation is executed by virtue of its template action `exists` property the current group is checked if any actions of `point` exists and if those actions have the same file as the current buffer's file then the the user chooses to update the existing entry or not. It does this with an exists property of

```
(exists (<predicate> file) (update point))
```

where <predicate> is a function that returns non-nil if its argument is the same file as the current buffer's file. The template has two slots `point` being the point within the file of slot `file`. So then what the above exists property in effect says is: If the current groups has a `point` action entry then ask the user if he wants to update that entry and if the user chooses then update the `point` slot. And since the template has for the slot `point`

```
(point "Point to go to" (type integer) (sfun point))
```

the `sfun` function is the standard Elisp function `point` which returns the current point in the buffer. Consequently the exiting entry's point is updated to the current point in the current buffer.

Here is a not quite so good example of using the `exists` template slot property but is easier to understand and try

```
(agroups-add-action-template
 '((action file-message)
   (afun ((lambda (file) (message "File reminder: %s" file)) file))
   (exists (buffer-file-name) (update file))
   (slots (file "File to show in message" (sfun buffer-file-name)))))
```

12.10 default-slot Template Property

When the Agroups editor prompts for a slot to edit it gives a default in parentheses so that if the user just hits RET it selects that slot to edit. Normally this default slot is the first slot in the slots property but the template designer can change this with the default-slot property.

The default-slot template property has the form

```
(default-slot <slot symbol>)
```

where <slot symbol> is the slot symbol whose choice will appear as the default slot when the Agroups editor asks for a slot.

12.11 Accessing Agroups Prefix Arguments

Any function or lambda specified in an action template `afun`, `pfun` or `sfun` or any function subsequently invoked from those can access two special variables to capture the prefix argument given when executing an action entry of this action type. The variable

```
agroups-arg
```

will contain the actual prefix argument. So, for example if the prefix argument was C-u this variable would contain (4). The variable

```
agroups-numeric-arg
```

contains the standard Emacs numeric interpretation of agroups-arg. So, for example if the prefix argument was C-u this variable would contain 4. Or to be very precise its value is always

```
(prefix-numeric-value agroups-arg)
```

Here is an example of an `afun` property function using agroups-arg

```
(agroups-add-action-template
 '((action compose-message)
   (afun ((lambda (first second)
            (message "%s %s"
                     (if agroups-arg second first)
                     (if agroups-arg first second)))
          first second))
   (slots
    (first "First part of message")
    (second "Second part of message"))))
```

So if you create an entry of this action and try it it should display a message in the minibuffer with the first slot and then the second slot and then try it with a C-u prefix argument before KA it should display the message with the second slot first and then the first slot.

13 Customizing Agroups Environment

Here are some things that users can do to change the behavior of Agroups. It is possible to set options for various preferences. You can set the accelerator keys for all given predefined operations and actions to suit your typing. And finally for what ever reason you may want, you can create your own styled new Agroups command.

13.1 Setting Options

You can set various kinds of options with the operations

```
Set Options (keys: s o)
Set Options back to defaults (keys: s O)
```

When you select the first operation above, Agroups prompts you to choose an option. The choice is a toggle between "no" and "yes", so each time you choose a particular option it will toggle back and forth between "no" and "yes". All choices default to "no". When you select the second operation above it sets all options back to the predefined "no" defaults. When you make the choice it is persistence between Emacs sessions. These options have the following meanings:

Option: `show-groups-when-change-current`: When you change the current group Agroups notifies you in the minibuffer. If you also want to automatically see a list of top level groups (like the "Display all groups" operation) *every* time you change the current group, toggle this option to "yes".

Option: `show-matching-action-entries`: Some actions will ask you if you want to update existing entries if there are entries in the current group of the same kind. For example the Point Actions do this, See Section 7.3.2 [Point Actions], page 29. If such existing entries exist they will prompt in the minibuffer

```
Do you want to update existing? (y or n)
```

If in addition to this prompt you also want to see a list of such existing entries each time, then set this option to "yes".

Option: `numeric-choice-always-get-return`: Many menus ask the user to type a number to select a choice. If the number of menu choices is 10 or less then just hitting the keys 0-9 will cause the choice to take effect. When more than 10 choices a RET is required after the number is entered. If the user prefers to always have to consistently enter a RET after the number then set this option to "yes".

Note that in all cased and with this option or not, hitting a RET alone will always select the default choice.

Option: `resolve-keys-to-zzzap-target`: Normally when moving entries to the zzzap group accelerator keys resolution does not take place, See Section 6.5 [Ambiguous Keys Resolution], page 27. If If don't want this you can set this option to "yes".

Options: `dont-ask-for-new-entry-keys` `dont-ask-for-new-group-keys`: Normally when creating a new entry or group the user is asked for accelerator keys for that entry or group. Some users may prefer to use *only* completion for selecting an entry or group. If this is their choice it may be annoying to have to respond to this "asking for accelerator keys" every time. If you set these options to "yes", `dont-ask-for-new-entry-keys` for entries and `dont-ask-for-new-group-keys` for groups, then Agroups will not ask. The

default on creation of an entry or group then will be no accelerator keys. When this is done accelerator keys can still be added by editing entries.

13.2 Changing Predefined Keys

Accelerator keys help speed up executing actions and part of the philosophy of Agroups is to do what ever it can to ease executing actions. But part of this concept is the how well predefined accelerator keys combine with user defined accelerator keys. So Agroups adopts the same philosophy of Emacs to allow the user to completely customize all keys bindings.

You can change any existing predefined keys with the operation

```
Set predefined Keys (keys: s k)
```

When you select this operation you are asked which keys to change with the prompt

```
The following type of changes on predefined keys are possible:
The Clear types require a reload of agroups to take effect.
 0 = Change Meta key
 1 = Remap Operation key
 2 = Remap Predefined Action key
 3 = Clear all Operation key Remaps
 4 = Clear all Predefined Action key Remaps
```

Choosing "Change Meta key" allows you to persistently change the default Meta key bindings that we refer to in this document as KO, KG, and KC and explain in an earlier section, See Section 1.2 [Must Read First], page 2. After choosing "Change Meta key" you get prompted with the choices

```
Meta key choices:
 0 = Select an operation
 1 = Group selection operations
 2 = Enter completion mode
```

which correspond to KO KG KC respectively. There are many reasons why you might want to change these, but one may be that if you define KA as a control key sequence it may be convenient to also have KO and KG bound to something close to the control key sequence. In the Must Read First section we give an example of binding KA to "\C-z\C-a". In this case it might be convenient then to bind KO to "\C-z" and KG to \C-a" for example. It is not recommended to change the KC, completion, binding which defaults to SPC since the Emacs default for word by word completion is bound to SPC and it combines nicely when using completion in Agroups.

The "Remap Operation key" and "Remap Predefined Action key" choices in the "changes on predefined keys" menu above work exactly the same except that the first allows you to remap keys from the collection of Operations and the second allows you to remap keys from the collection of Predefined Actions. These work by first asking you to select an operation (or action) which you can do by using accelerator keys or by completion. Then they ask for a keys to remap to.

Again when you make these mappings they are saved persistently for you so that the next time you run Emacs you will get the new key bindings. The "Clear all Operation key Remaps" and "Clear all Predefined Action key Remaps" choices allow you to undo *all* of one of the two remaps described above respectively. It does this by removing all key

mappings for that collection in your Agroups save file so that you can start over with the default predefined key bindings.

13.3 Creating New Agroups Commands

Much like the `agroups` command that we bound to `"\C-z\C-a"` as an example in an earlier section, See Section 1.2 [Must Read First], page 2, it is possible to create new Agroups commands. This can be done with the `agroups-command` macro. In fact the `agroups` command itself is created with this macro. The form of this macro is

```
(agroups-command <command symbol> <save file>
                 &optional <existing command symbol> <prefix keys>)
```

where <command symbol> is the function symbol that you want your new Agroups command called and <save file> is a string representing the new Agroups save file. The optional arguments allow creating an Agroups command that shares a piece of an existing Agroups command where <existing command symbol> is the existing command function symbol and <prefix keys> is a string of accelerator keys that can take you into the existing piece of the existing Agroups command group structure. When you create a shared command in this way you should use the same save file that the existing command uses.

First, lets take the simpler example. Suppose that we want to create a new Agroups command called `agroups-1`. The default `agroups` command save file is `"~/.agroups"` so lets call our new save file `"~/.agroups-1"`. We would then create the new Agroups command with

```
(agroups-command agroups-1 "~/.agroups-1")
```

We should then be able to execute the new Agroups command with "M-x agroups-1" or bind agroups-1 to some keys in the standard Emacs way.

Second, lets take a shared example using a prefix. Using the prefix is equivalent to "KA <prefix keys>" of the existing command. Which allows you to bind any operation, entry or group of the existing Agroups command to your new Agroups command. In our introduction examples we showed how Agroups gives you a staring group called global that by default has accelerator keys ".". One useful thing to do is to create a new Agroups command that shares the global group with your existing Agroups command. We can do this easily with

```
(agroups-command agroups-g agroups-file agroups "\C-a.")
```

After doing this we will have a new command function `agroups-g`. Since we want to share the existing command `agroups` global group we want to also use the same Agroups save file. Since the default save for `agroups` is stored in the variable `agroups-file` we want to use that for the <save file> argument instead of a string since if we change this variable for the existing command we want the shared command to change as well. We next give the <existing command symbol> argument as `agroups` which is the `agroups` command that we have been using in this document that we are going to share. And finally the <prefix keys> argument which is `"\C-a."` assumes that we changed the KG meta key in this chapter from the default of TAB to C-a, so that "C-a." would take us into the global group. Finally to complete this idea, since in our Introduction examples we bound `agroups` to `C-zC-a` lets bind this new command to `C-za`

```
(global-set-key "\C-za" 'agroups-g)
```

Now, if you followed our key binding example including the change of key bindings example in this chapter, then typing

```
C-z a
```

would now be equivalent to typing

```
C-z C-a C-a .
```

14 Installing Agroups

If you are reading this document it is most likely that Agroups has already been installed. This chapter is only included in this document for completeness.

To make Agroups unpack it in some directory with

```
tar -zxvf tar-file
```

where *tar-file* is the Agroups tar file with the `tgz` file extension. This will create an `agroups` sub-directory there. Change to the `agroups` sub-directory and type either

```
./configure
```

or

```
./configure --prefix=PATH
```

The first will prepare to install Agroups under the default directory `/usr/local`. The second will prepare to install Agroups under a directory PATH that you provide.

Then type

```
make
```

To install the made Agroups system depending on the operating system and installation PATH you may need to log in as a super user. Then type

```
make install
```

The key files that get installed are

```
PATH/share/emacs/site-lisp/agroups.el
PATH/share/emacs/site-lisp/agroups.elc
PATH/info/agroups.info
PATH/doc/agroups/agroups.html
```

where PATH is the installation directory.

To get started you should read the Agroups user document. A quick start can be tried by just reading the Introduction chapter. The `agroups.html` file can be read in a web browser. So for example if the above PATH was `/usr/local` you can use the URL

```
file:/usr/local/doc/agroups/agroups.html
```

Or if you want to read the Agroups user document in Emacs with the `info` command, while in Emacs type

```
C-u C-h i /usr/local/info/agroups
```

You can also get a PostScript copy of the Agroups document by changing to the directory where you typed `./configure` and then change to the sub-directory `doc` and type

```
make ps
```

which will create a PostScript version of the Agroups user document called

```
agroups.ps
```

Operation Index

A

C

D

E

L

M

P

R

S

V

W

Concept Index

Table of Contents