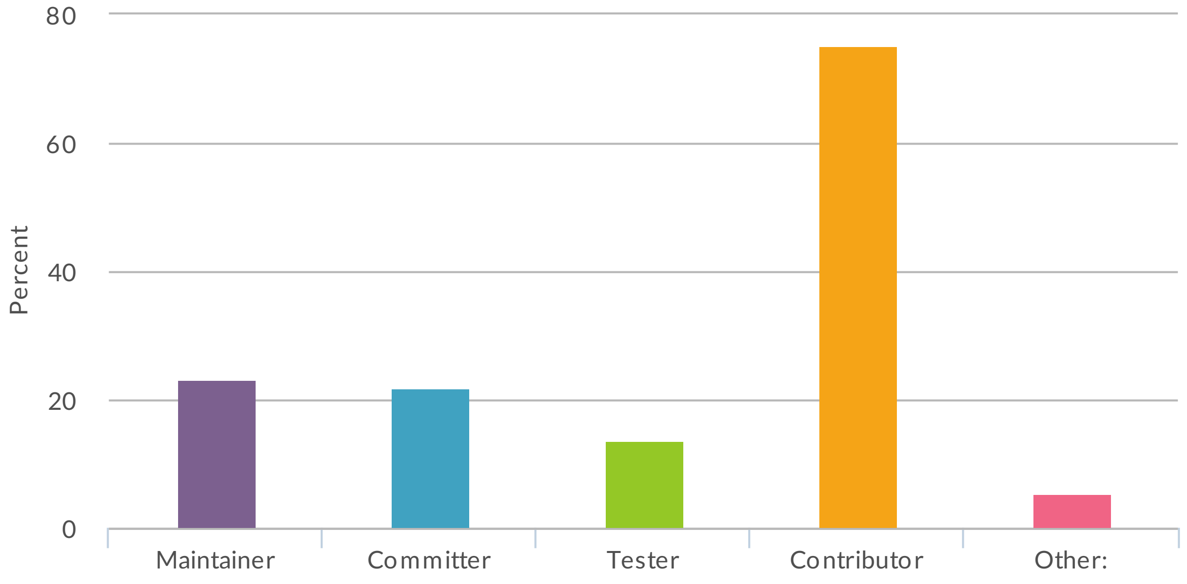
Standard Report_ 74 responses_April 30th

1. Response Counts

Completion Rate: 77%

Complete		57
Partial		17
Disqualified		0
	Total	74

2. What is your role in working with the Linux kernel?

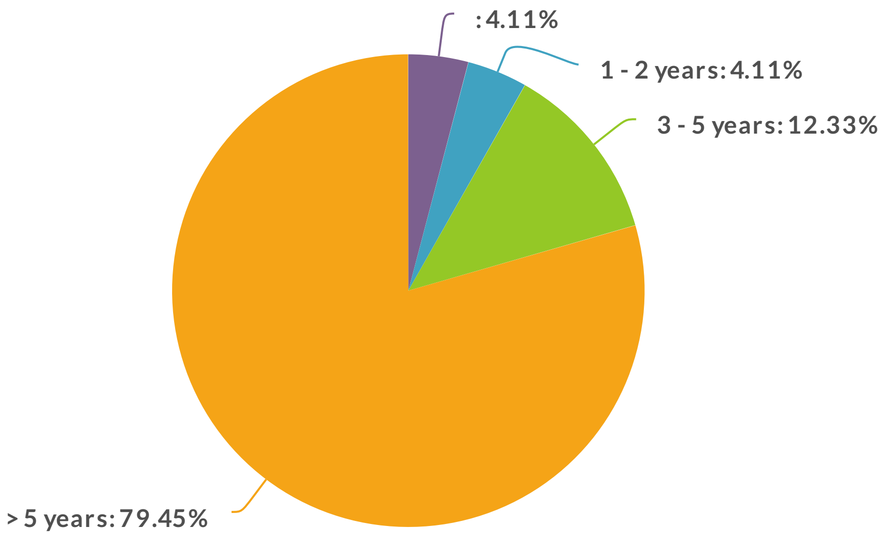


Value	Percent	Count
Maintainer	23.29%	17
Committer	21.92%	16
Tester	13.70%	10
Contributor	75.34%	55
Other:	5.48%	4

Statistics	
Total Responses	73
Skipped	1

Other:	Count
Commercial software developer that uses the linux kernel	1
Debugger	1
Researcher	1
very occasional patches	1
Total	4

3. How many years of experience do you have in working with the Linux kernel?

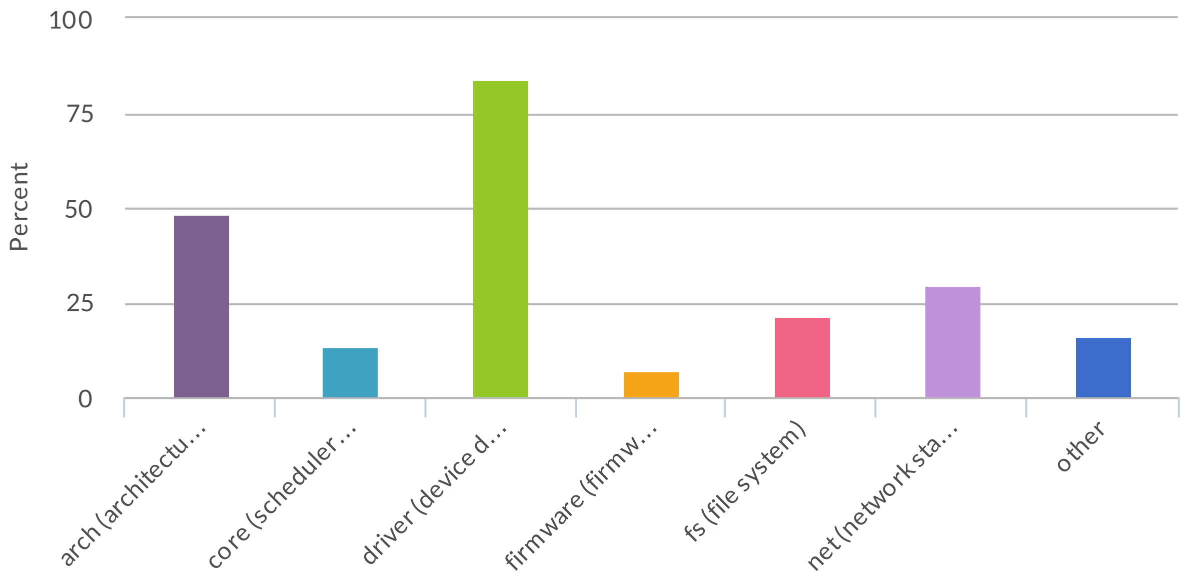


Value	Percent		Count
< 1 year	4.11%		3
1 - 2 years	4.11%		3
		Total	73

Value	Percent		Count
3 - 5 years	12.33%		9
> 5 years	79.45%		58
		Total	**73**

Statistics	
Total Responses	73
Min	0.00
Max	3.00
Sum	30.00
Average	0.41
StdDev	0.99
Skipped	1

4. Which kernel subsystems do you work on?

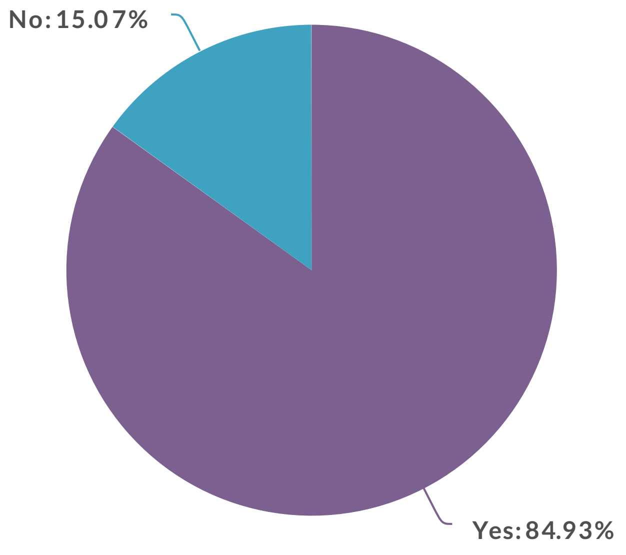


Value	Percent		Count
arch (architecture dependent code)	48.65%		36

Value	Percent	Count
core (scheduler, IPC, memory management, etc)	13.51%	10
driver (device drivers)	83.78%	62
firmware (firmware required by some device drivers)	6.76%	5
fs (file system)	21.62%	16
net (network stack)	29.73%	22
other	16.22%	12

Statistics	
Total Responses	74
Skipped	0

5. Do you work with kernel code that uses ifdefs?

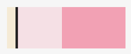
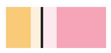
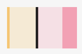
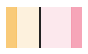


Value	Percent	Count
Yes	84.93%	62
No	15.07%	11
	Total	73

Statistics

Total Responses	73
Skipped	1

6. Which of the following methods do you use to create a new device driver? Please order applicable methods according to the frequency you apply them.(if you do not use one of the listed methods, you can leave it in the left column)

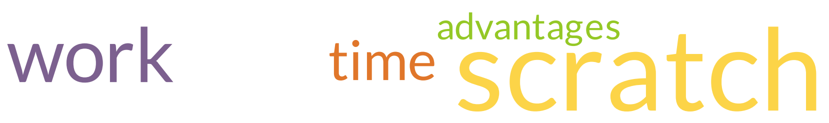
Overall Rank	Item	Rank Distribution	Score	Total Respondents
1	Clone an existing driver and adapt it		180	52
2	Develop from scratch		126	44
3	Use a template		87	32
4	Extend an existing driver by introducing variants using ifdefs (without cloning)		84	33

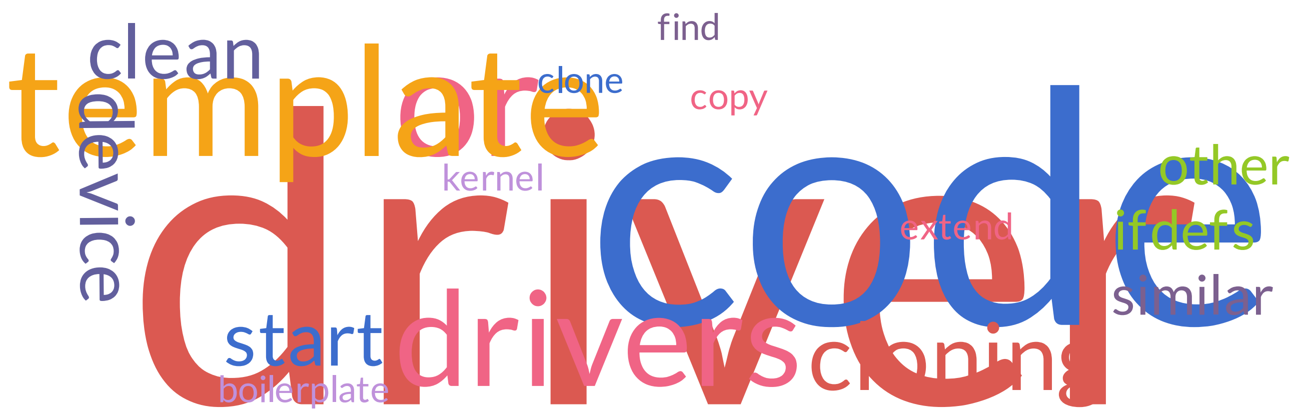
🟧 ⬜ | ⬜ 🟥
Lowest Rank — Highest Rank

Statistics

Total Responses	62

7. Please briefly explain your most frequent method when creating a new device driver. What are its advantages and disadvantages?



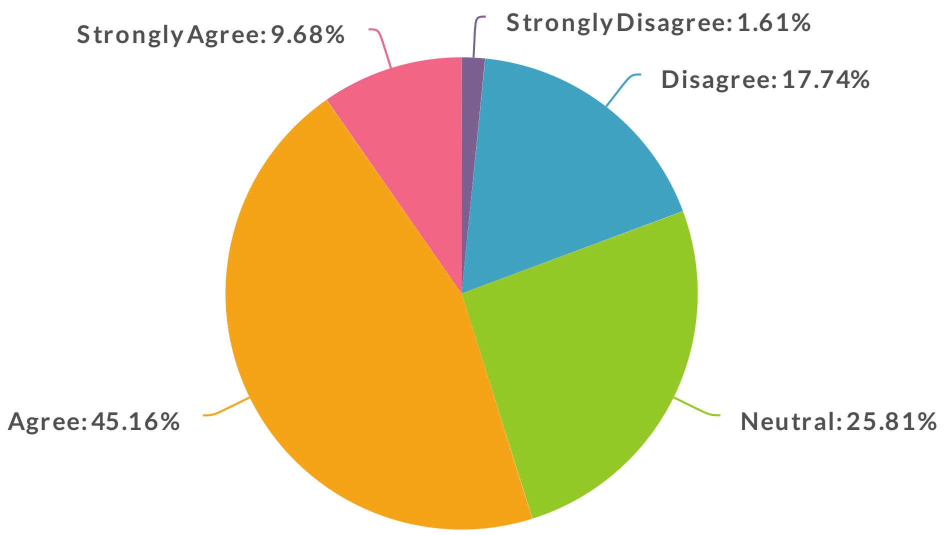


Count	Response
1	+ Understandable code, easy to maintain for one driver. - Bad if needed to maintain complete subsystem.
1	A lot of kernel-specific initialization code (driver registration, binding, specifying callbacks etc.) is almost identical among drivers so copying an existing one is a reasonable starting point. The disadvantage is that most of the code is deleted anyway and if the source had some bugs they get copied to the new driver.
1	Adapting an existing driver means you don't have to deal with all the boilerplate It also means it follows the style of that class of driver so is familiar to other developers.
1	Advantage is that it is known working code. Disadvantage is that you copy possibly bad habits unknowingly
1	Alrhough cloning and changing is the quickest option, I find it usually only acceptable for PoC prototypes....it is always better afterwards to have a clean from scratch implementation. Having tons of identical boilerplate across multiple drivers is a sign that the interfaces could use some work and the shared bits lushed up or out. Sometimes you add features to an existing subsystem, which means extending existing code. Since sometimes these are optional features that are not necessary or are dependant on hw/sw support, these get the ifdef treatment.
1	Based on existent one. Adv: easy to get it working. Disadv: too many clean-ups to do.
1	Clean-room reverse engineering of a binary WLAN driver. Disadvantage is that you never really know what a specific line really does with the hardware.
1	Cloning an existing driver and adapt it. Saves the time of figuring out all the template code.
1	Cloning an existing driver to get to results quicker, which has the advantage of already tested code and the disadvantage of having bugs
1	Complete new device type -> clone. Extend existing device with new version => extend however avoid ifdef as much as possible. However sometimes ifdef can prevent unnessesary inclusion of unused data objects.
1	First try to find the possibility that I can use and extend existing driver. If not, create a new one.
1	Firstly, one can't really use ifdefs nowadays. instead we use device tree compatibles. Now, depending on the device, there could be driver already available by vendor in some android-vendor source tree but usually with really bad code quality. But sometimes it's really hard to find a datasheet...
1	From a template or from scratch. This is the best way to "know" the code.

Count	Response
1	I made a two drive-by commits to an existing driver. I modified the existing driver to work with the new hardware without the use of ifdefs.
1	I prefere to create a new device driver refering to similar driver. This saves development time.
1	I tend to copy-paste relevant code pieces to get enough skeleton for my needs so that I don't have to figure out how to get the basics up and running. Then I continue with the actual functionality.
1	I usually prefer to start from clean sheet, from a template. This allows me to think the structure by myself (within the frame of the template of course). This allows me to keep the drivers clean, short and simple. The main disadvantage is the code duplication, or the number of very similar drivers that grow.
1	I usually start with looking at what is already out there, first of all to see if there is an existing driver that will suit my needs, usually not, because I work with custom HW which needs custom drivers. In that particular case, I use a combination of let's look at other drivers to see what they did, how they got structured, and write from scratch, since only a few things will be re-usable.
1	I work as a volunteer to adapt wireless drivers written by Realtek Corp. Usually, this means to convert their code to meet kernel coding standards. I also maintain these drivers and debug such things as memory leaks and kernel oops conditions. A disadvantage is that I have no knowledge of the inner workings of the wireless chips, thus I cannot fix performance issues.
1	If a template exists (and ca be found easily), I will use this one, but as most of the time this isn't true, cloning and adapting a existing driver is the most common way. I would use ifdefs in a driver only, if I have configurable parts, that can only be enabled/disabled before compiling it or if it has dependencies, which I cannot control.
1	Initially providing low-level access to peripherals only. This allows for quick hardware verification without having to develop an interface or data model.
1	It depends on how different the new hardware is. If the new hardware is just a variation of another part, extend with ifdefs or runtime checks. If it's not that close, cloning and gutting code that doesn't fit the new hardware. That let's me focus on writing code specific to the new hardware and not worrying about either getting the interfaces correct or the implementation that can be reused.
1	It is depend on how new driver is similar to existing one. For new HW I clone old driver or use template For new version of the same HW I can use ifdef
1	It's quick to get started as all the boilerplate is there already. It may be better to use a template, but even then I'm copying in a lot of other code that would have been there for me already if I copied another driver.
1	Kernel policy is to extend existing drivers (over clone-and-own) wherever possible.
1	Most often I start from making a copy of an existing driver, then I modify it. Sometimes differences are so significant that I end up with a code that seems hard to manage, then I use it as a template and create a new code from it. Advantages: less time spent on development, consistency with existing code patterns. Disadvantages: sometimes the existing driver used occures not in line with latest API developments and the resulting work gets outdated soon.
1	Most often there is a somewhat working vendor-driver available, so it's easiest to clean this up to make it conform to kernel standards.

Count	Response
1	Most times there is no need to #ifdefery in extending an existing driver. That is my most frequent method for creating a new device driver. The second one is to write it from scratch if there is no extensible driver for this IP.
1	The advantage is that I start with a reasonable structure, leverage APIs like module initialization. sysfs support, etc.
1	The advantages is that lot of code is already done, the disadvantage is that sometimes this copy/paste technique introduces some small errors. Also you can not always use this method
1	The advantages of code reuse apply to device drivers more than elsewhere because of the difficulty of testing hardware interfaces. Shared code (at link-time or compile-time) is better than duplicated code for reasons of code maturity and less maintenance.
1	The general form is already laid out, and many of the required methods can be immediately identified. Unlike cloning an existing driver, one is not distracted by old code. A template can be a disadvantage if the structure of the template does not really align will with the hardware being developed for. That is, it assumes that a particular driver structure will work for a new piece of hardware. While this is true in many cases, sometimes the assumption will break down.
1	Use existing driver as a template. It will have all necessary functions, and will be up to date, unlike template. Developing from scratch is most cumbersome. I would never add ifdefs to some other driver - if they are so similar, it should be possible to have one driver for both sets of hardware.
1	Usually, my preferred approach is to modify an existing driver so it supports the new hardware as well as any existing hardware without having to rebuild it. If no existing driver is close enough to my needs, writing a new driver from scratch is usually best (using snippets and boilerplate from existing drivers where possible).
1	While developing from scratch, you are motivated to concentrate on the essential features/issues. This often leads to clean and lean code. If your new driver fits into an existing category, then cloning might be the best. However, within the field of embedded devices often you need special functionality, so focus is on "from scratch". A disadvantage is however, you may oversee things as well as you may have to find all of the pitfalls yourself.
1	Writing from scratch using the existing code as documentation. Advantages are you start from first principles, and understand everything about how the driver is structured (including locking and how it interacts with other subsystems.) The disadvantage is it may not be as easy to get it brought up, as sometimes what seems like poor decisions on the original author are working around hardware bugs or limitations that are insufficiently documented.
1	clone or template, Rubinis' ldd volume 2 or 3?
1	drivers generally have a common basic structure. As there are rarely pristine skeleton driver examples available, developers will often start from a driver thought to be similar and adapt it as per the needs of the device being supported

8. "Working with ifdefs is challenging, regardless of scattered code."

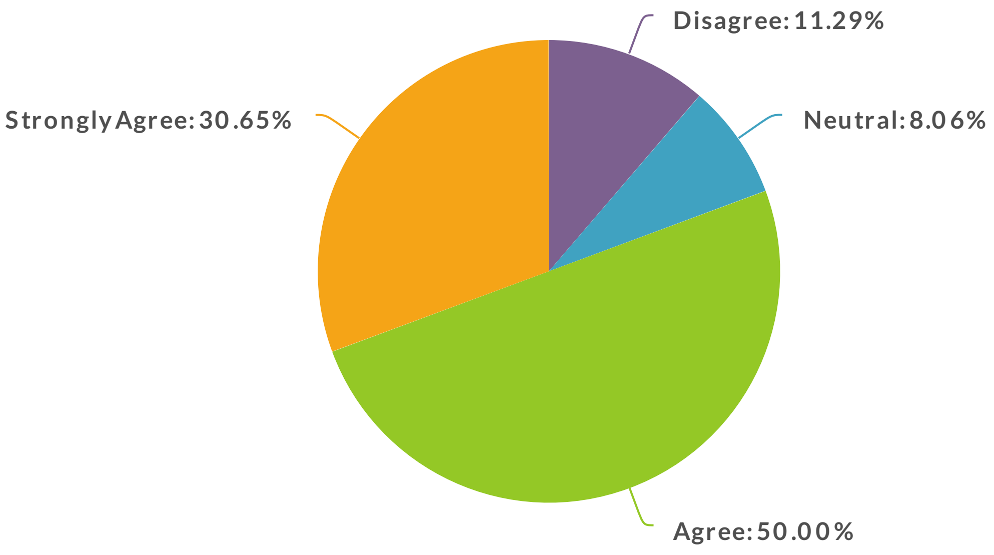


Value	Percent		Count
Strongly Disagree	1.61%		1
Disagree	17.74%		11
Neutral	25.81%		16
Agree	45.16%		28
Strongly Agree	9.68%		6
		Total	**62**

Statistics

Total Responses	62
Skipped	1

9. "Working with highly scattered code is challenging."



Value	Percent		Count
Disagree	11.29%		7
Neutral	8.06%		5
Agree	50.00%		31
Strongly Agree	30.65%		19
		Total	**62**

Statistics

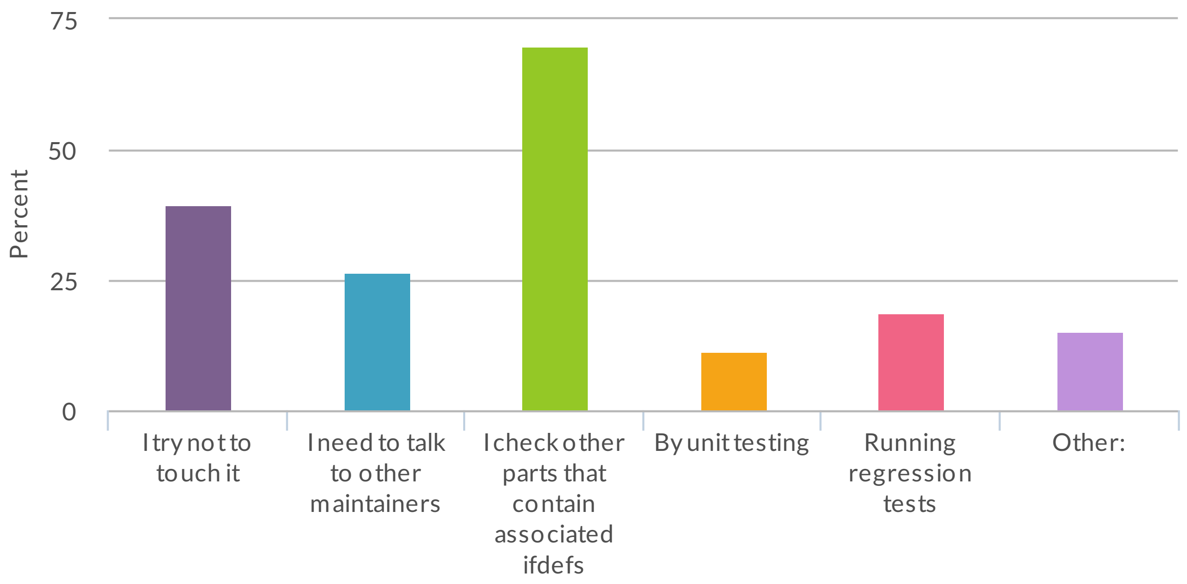
Total Responses	62
Skipped	1

10. Does highly scattered code impact the following software-development aspects?

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree	Responses
Increases maintenance effort	2 3.28%	4 6.56%	5 8.20%	35 57.38%	15 24.59%	61
Lowers testability	2 3.28%	8 13.11%	12 19.67%	27 44.26%	12 19.67%	61

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree	Responses
Complicates program comprehension	0 0.00%	2 3.28%	8 13.11%	34 55.74%	17 27.87%	61
Hinders evolution	0 0.00%	13 21.31%	20 32.79%	24 39.34%	4 6.56%	61
Increases communication effort with other developers	2 3.33%	13 21.67%	26 43.33%	16 26.67%	3 5.00%	60
Introduces more bugs	1 1.64%	8 13.11%	20 32.79%	27 44.26%	5 8.20%	61
Lowers patch acceptance	0 0.00%	10 16.39%	26 42.62%	19 31.15%	6 9.84%	61
Decreases code quality	0 0.00%	9 14.75%	21 34.43%	23 37.70%	8 13.11%	61
Decreases frequency of code changes	2 3.28%	18 29.51%	32 52.46%	7 11.48%	2 3.28%	61
Decreases performance	0 0.00%	0 0.00%	0 0.00%	1 100.00%	0 0.00%	1
Failure modes and effects analysis	0 0.00%	0 0.00%	0 0.00%	0 0.00%	1 100.00%	1
Knowledge transfer	0 0.00%	0 0.00%	0 0.00%	0 0.00%	1 100.00%	1
Total	9	85	170	213	74	551
Average %	1.63%	15.43%	30.85%	38.66%	13.43%	

11. How do you maintain scattered code? Add more activities in the three textfields if you want.



Value	Percent		Count
I try not to touch it	39.62%		21
I need to talk to other maintainers	26.42%		14
I check other parts that contain associated ifdefs	69.81%		37
By unit testing	11.32%		6
Running regression tests	18.87%		10
Other:	15.09%		8

Statistics

Total Responses	53
Skipped	10

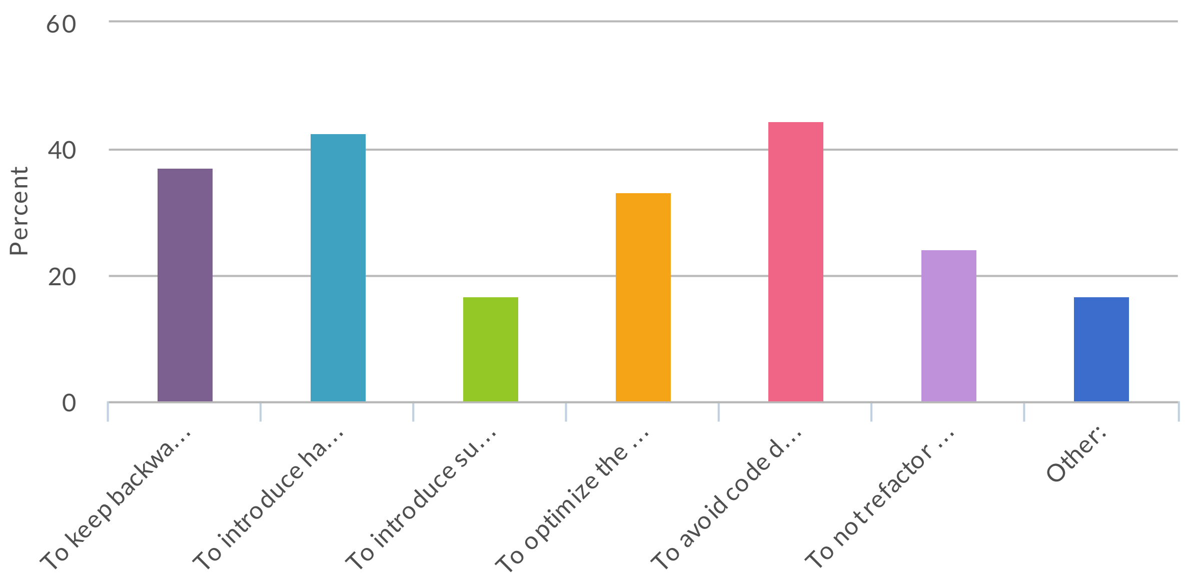
Other:	Count
0-day tests	1
I try to make them non-scattered	1
Replace ifdefs with conditional code (if (xxx) { ...})	1
Testing with randcondig	1
Total	8

Other:	Count
Try to clean up (if possible)	1
Try to get rid of it	1
Try to kill off all the ifdeffery - it\'s mostly historic	1
refactor it	1
Total	8

Other:	Count
Total	0

Other:	Count
Total	0

12. Why do you introduce or maintain scattered code? Add more reasons in the three textfields if you want.



Value	Percent		Count
To keep backwards compatibility	37.04%	▨	20
To introduce hardware variability (due to limitations in hardware detection)	42.59%	▨	23

Value	Percent		Count
To introduce support for a generic set of devices (not related to specific vendors)	16.67%	🟩	9
To optimize the code for performance or binary size	33.33%	🟧	18
To avoid code duplication	44.44%	🟥	24
To not refactor existing code with scattered configuration options	24.07%	🟪	13
Other:	16.67%	🟦	9

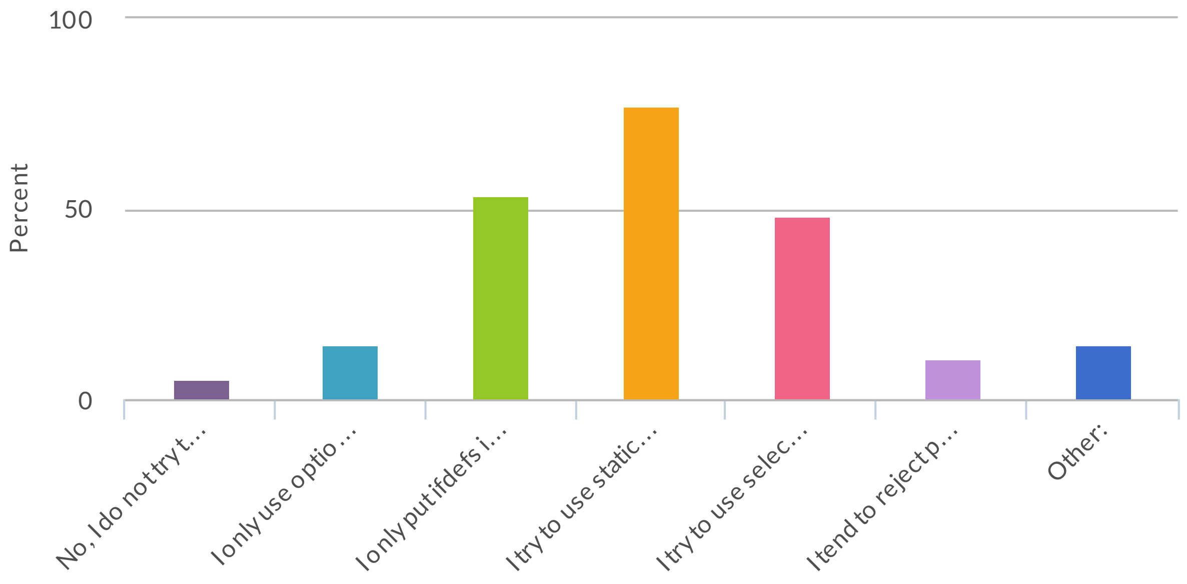
Statistics	
Total Responses	54
Skipped	9

Other:	Count
I sometimes avoid rewriting existing scattered code that I cannot test.	1
Minimize source changes	1
Provide for debug options which don\'t affect production code	1
Sometimes it the one-eyed among blind option	1
Support OSes other than Linux	1
To group pieces of code on the functional basis where it is more natural to place them	1
To introduce debugging version of a function	1
To modify code according to high-level options	1
various options	1
Total	9

Other:	Count
Total	0

Other:	Count
Total	0

13. Aligned with the above guidelines, do you try to avoid using ifdefs? If so, how?



Value	Percent		Count
No, I do not try to avoid using ifdefs	5.36%		3
I only use optional code confined in one single ifdef	14.29%		8
I only put ifdefs into header files (as explained in the guidelines above)	53.57%		30
I try to use static inline functions (as explained in the guidelines above)	76.79%		43
I try to use selection statements (such as if and case statements) from the C language instead of ifdefs	48.21%		27
I tend to reject patches containing ifdefs	10.71%		6
Other:	14.29%		8

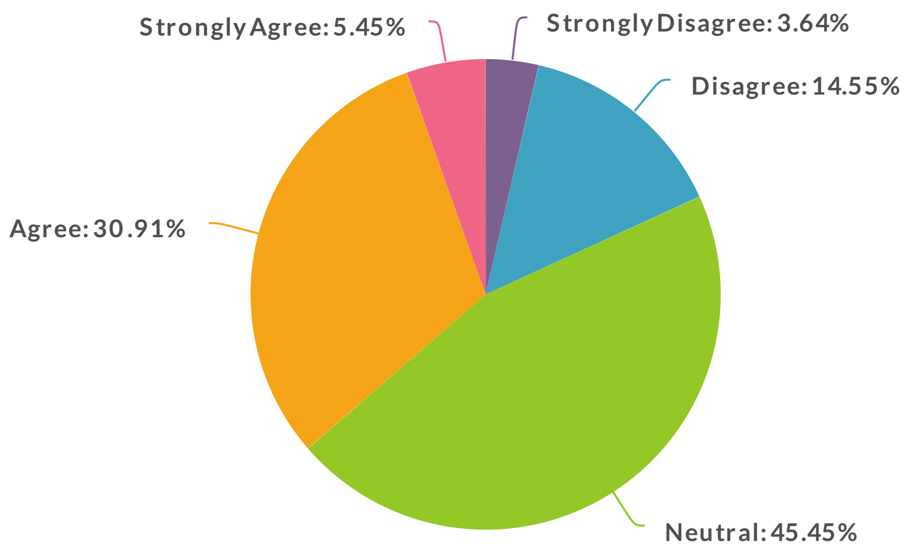
Statistics

Total Responses	56
Skipped	2

Other:	Count
Total	8

Other:	Count
Existing code does not necessarily conform to the guidlines. One has to take a case-by-case approach.	1
For HW-related conditionals, I always want to have a single source running and detecting at run-time which HW it runs on, so no need for ifdefs (most of the time)	1
I come from C++ and dislike ifdefs very strongly due to type system breaking	1
I only use ifdefs if there is no efficient alternative	1
I put ifdefs into header files where applicable	1
I try not to use it	1
It really depends on the situation; sometimes an ifdef is appropriate. I certainly try to keep them to a minimum though, for the reasons stated in the guidelines.	1
i try to avoid ifdefs, but it can avoid redundant inconsistencies if used wisely	1
Total	8

14. To what extent do you agree with the statement? "The scattering practices would differ when using selection statements (e.g., if and case statements) instead of ifdefs."

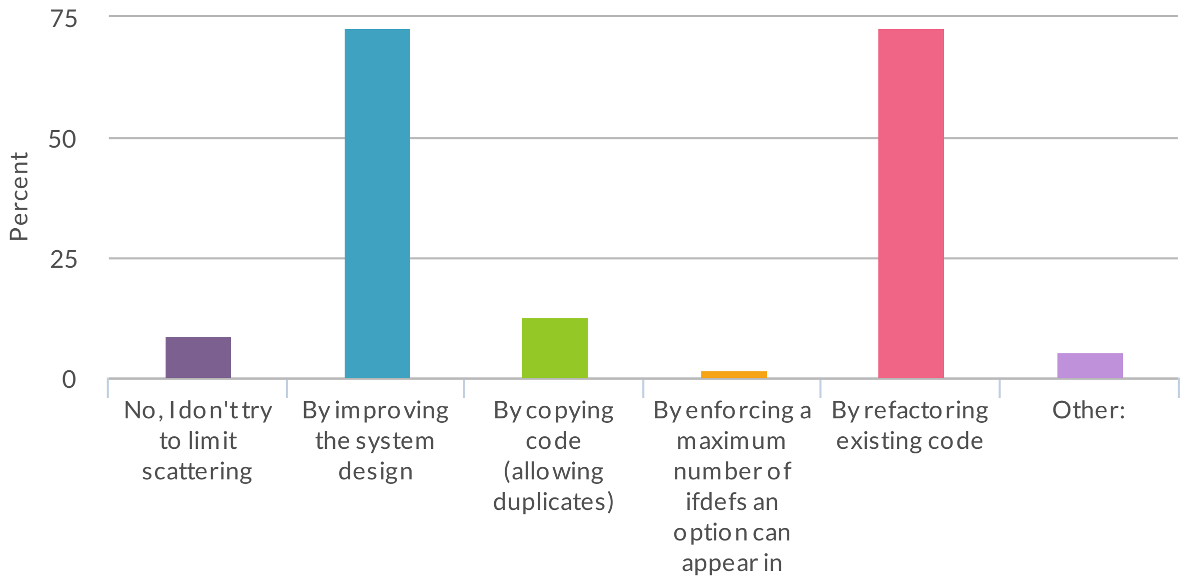


Value	Percent	Count
	Total	55

Value	Percent		Count
Strongly Disagree	3.64%		2
Disagree	14.55%		8
Neutral	45.45%		25
Agree	30.91%		17
Strongly Agree	5.45%		3
		Total	**55**

Statistics	
Total Responses	55
Skipped	3

15. Do you try to limit scattering? If so, how?



Value	Percent		Count
No, I don't try to limit scattering	9.09%		5
By improving the system design	72.73%		40
By copying code (allowing duplicates)	12.73%		7

Value	Percent		Count
By enforcing a maximum number of ifdefs an option can appear in	1.82%		1
By refactoring existing code	72.73%		40
Other:	5.45%		3

Statistics	
Total Responses	55
Skipped	3

Other:	Count
I don\'t worry about introducing multiple #ifdef blocks within a header file; sometimes multiple blocks can make the header easier to understand, or may be necessary. However, in the rare case when I must put an #ifdef in a C file, I make sure it\'s just one block rather than being \"scattered\".	1
Moving code to headers where possible	1
Scattering is sometimes appropriate, but IMHO it should be kept to a minimum.	1
Total	3

16. Do you try to limit scattering? If so, how? - Text Analysis

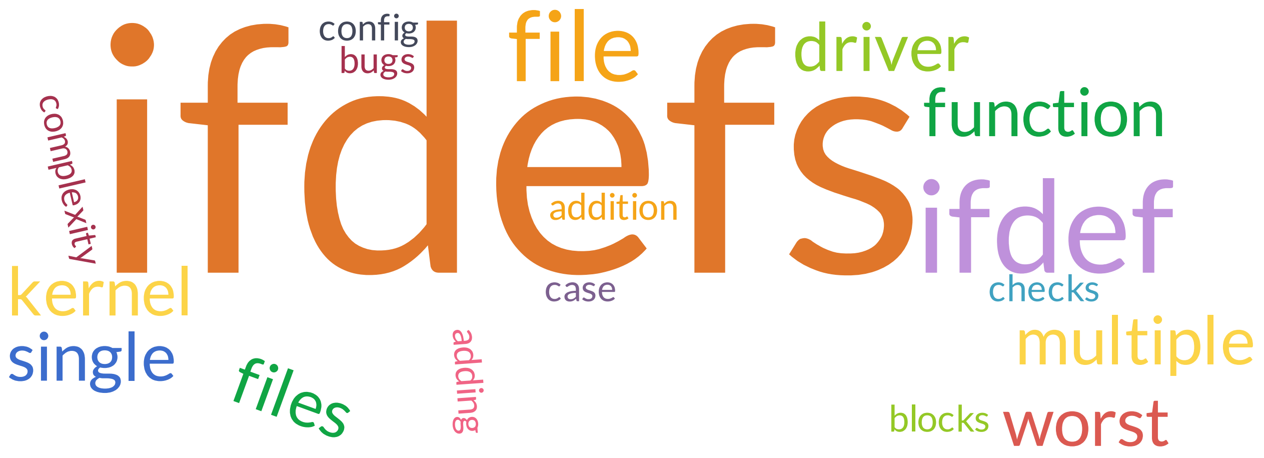
No data: No responses found for this question.

Statistics	
Skipped	0

17. To what extent do you agree with the following strategies to avoid scattering?

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	Responses
Use better modularity mechanisms in the C language	0 0.00%	3 5.26%	19 33.33%	28 49.12%	7 12.28%	57
Use a better system design	0 0.00%	1 1.75%	7 12.28%	31 54.39%	18 31.58%	57
Supporting cloning (i.e., maintaining code duplication)	13 24.07%	26 48.15%	10 18.52%	5 9.26%	0 0.00%	54
Use more tools to enforce guidelines	2 3.51%	12 21.05%	19 33.33%	20 35.09%	4 7.02%	57
Use of component frameworks	1 1.79%	9 16.07%	28 50.00%	16 28.57%	2 3.57%	56
I don't see any alternative	13 24.07%	10 18.52%	20 37.04%	11 20.37%	0 0.00%	54
I guide other people via review comments on how to avoid code scattering	0 0.00%	0 0.00%	0 0.00%	0 0.00%	1 100.00%	1
Use headers and static inline functions	0 0.00%	0 0.00%	0 0.00%	0 0.00%	1 100.00%	1
Total	29	61	103	111	33	337
Average %	8.61%	18.10%	30.56%	32.94%	9.79%	

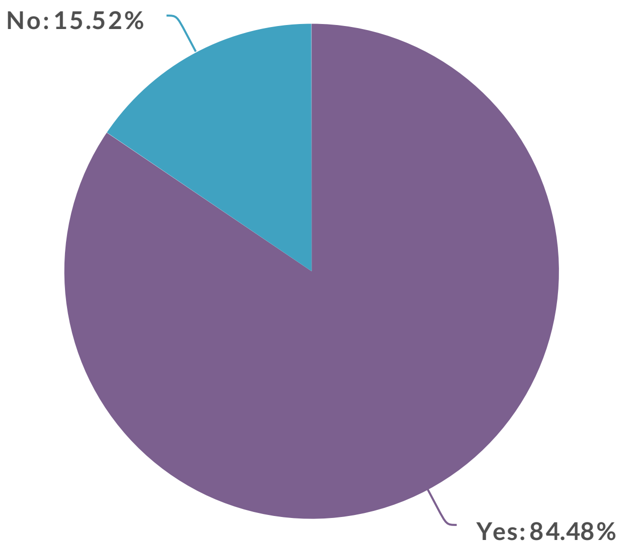
18. Could you please elaborate a bit on your practices, both with respect to using ifdefs and to dealing with code scattering?



Count	Response
1	Coding exp[illegible]nce ma[illegible]ers, [illegible]ility fo[illegible]eating[illegible]tterns, a[illegible]ering t[illegible]RY (do[illegible]repeat yourself)
1	Generally [illegible]tracting out th[illegible]ommon co[illegible] Wh[illegible]e we have [illegible]s of i[illegible]fs or runtime if () checks on specific hardware v[illegible]ants it normall[illegible]eans that t[illegible]driv[illegible]s the wro[illegible]way u[illegible]hould be a library of common helper code for driv[illegible]to ca[illegible]t a d[illegible]r) or [illegible]s som[illegible]ctio[illegible]p[illegible]ters in[illegible]d
1	I don't see much difference in conditionally defining one of two versions of a static inline function compared to having a single function with an if condition inside. As long as ifdefs are on the top-level of a file (as opposed inside of a function), I don't think they incur much complexity or problems.
1	I don't worry about introducing multiple #ifdef blocks within a header file; sometimes multiple blocks can make the header easier to understand, or may be necessary. For example, placing the #ifdef block inside the static inline function rather than around it can make it easy to see at a glance that the function signatures match, and only the body of the function (or lack of body) differs; in addition to avoiding duplication, that can make the code easier to read. However, in the rare case when I must put an #ifdef in a C file, I make sure it's just one block rather than being "scattered". And I never put an #ifdef around a portion of a function body rather than the whole thing. That would make the code flow difficult to understand.
1	I have found that runtime checks are usually preferable to ifdefs. In fact, new code with ifdefs is probably masking a problem with the overall design.
1	I only use them to include/exclude features that significantly increase the size of the driver and are not needed in all cases. In addition, the only ifdefs that are allowed are those for which kernel configuration parameters exist. In most cases, elimination of code scattering would lead to considerable obfuscation of the functions involved and would be far worse than the negative effects of scattering.
1	I try to avoid ifdefs as much as possible and prefer other techniques, such as IS_ENABLED() macro, that allow compile time code validation regardless whether option is set or not. If there are a lot of references (either via ifdefs or any other way) to some config definition, then that is a clear sign that the code design should be thought again.
1	I try to use selection statements and structure the code to avoid duplication
1	If there are a number of ifdefs in the code files (not header files), it's usually an excellent red flag indication that the code has problems and needs to be refactored, or in the worst case, redesigned. If I possibly can, I should focus on code cleanup rather than adding more ifdefs.
1	In non-kernel code, where others would use ifdefs, I define an interface with multiple implementations. Each implementation goes in a separate .c file, and then the build system selects the correct implementation.
1	In regard to #ifdef in new code I agree with the guidelines. My own practices predominantly concern existing code and not new code. More important to me than following the latest guidelines is to avoid introducing regressions in deployed code.
1	Limited to header files. Provide suitable substitutes with documentation when called the ifdef is unselected. Attempt to completely eliminate ifdef codepaths from stack traces.
1	Most often, supporting out of tree modules, I use conditional compilation to support multiple kernel versions.

Count	Response
1	Mostly I try to follow the Linux guidelines, since they are good practice hints. However there may be exceptions when scattered ifdefs are needed for goals like performance, code size and so on.
1	Sometimes both practices (ifdefs and scattering) can be useful and appropriate. However, I believe they should be kept to a minimum, with a minimum amount of code within a given ifdef. If the ifdefs become too long or too frequent it affects readability and is a sign that some of the avoidance strategies mentioned earlier should be investigated.
1	The biggest reason to use scattered ifdefs is to make code merging easier. #ifdef blocks are easier to merge than complex code flows. IMO this is the single biggest barrier towards removing scattered ifdefs. In an ideal world, the code would be structured such that hardware dependencies can be centralized, but that is an impossible continuously moving target. You can almost never predict where your very cleverly designed code will need to handle a bunch of different types of hardware until it is too late. And then you need a huge refactoring effort which increases complexity, may introduce bugs, increases testing difficulty, increases regression testing load, and worst of all, makes commit/branch merges very very complex and time consuming.
1	When supporting various config options that affect the same code, I usually put the code into header files, but there's times that makes things worse. Also, there's times when adding ifdefs in code actually documents what its there for.
1	as before I come from C++ and hate preprocessor macros with quite a passion. Worst bugs I have had to deal with have been due to type system breakage from #defines
1	ifdefs are used sometimes to enforce consistency in definitions see yealink.h header file :). Too much code scattering in driver code fork new code file for device rather than fit everything in a single framework.
1	it is boring!!!

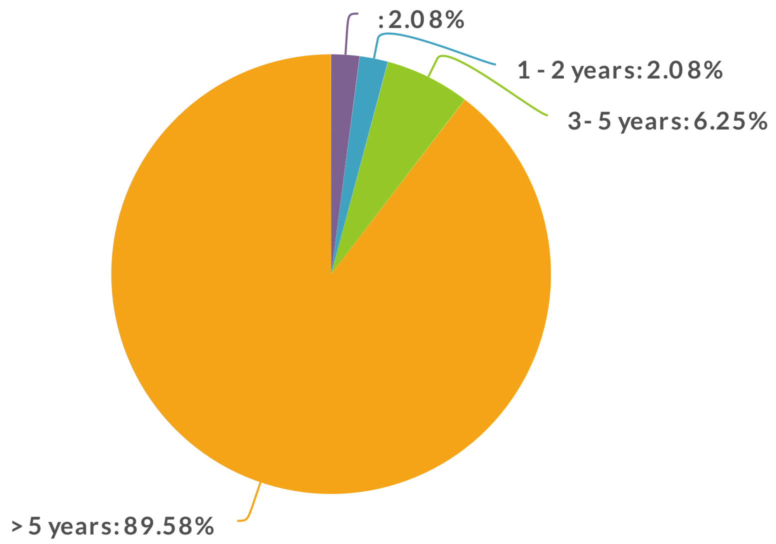
19. Are you also working as a professional developer in a company?



Value	Percent		Count
Yes	84.48%		49
No	15.52%		9
		Total	**58**

Statistics	
Total Responses	58
Skipped	0

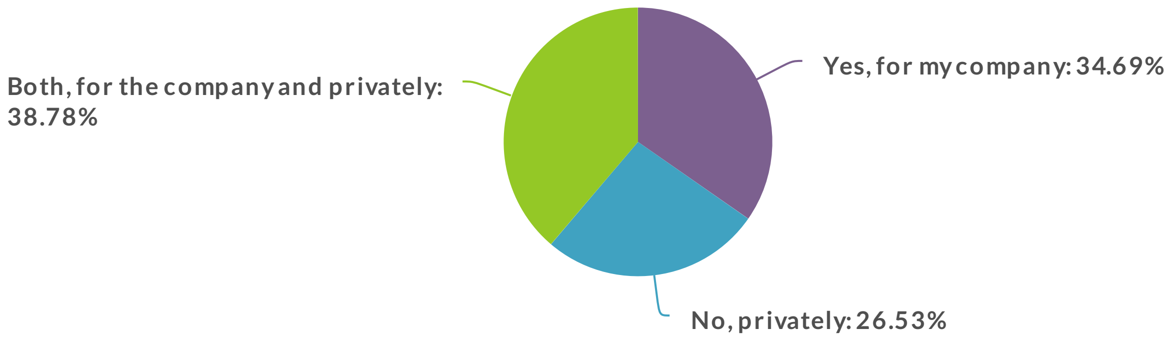
20. If yes, how long already?



Value	Percent		Count
< 1 year	2.08%		1
1 - 2 years	2.08%		1
3- 5 years	6.25%		3
> 5 years	89.58%		43
		Total	**48**

Statistics	
Total Responses	48
Min	0.00
Max	3.00
Sum	10.00
Average	0.21
StdDev	0.73
Hidden	25
Skipped	1

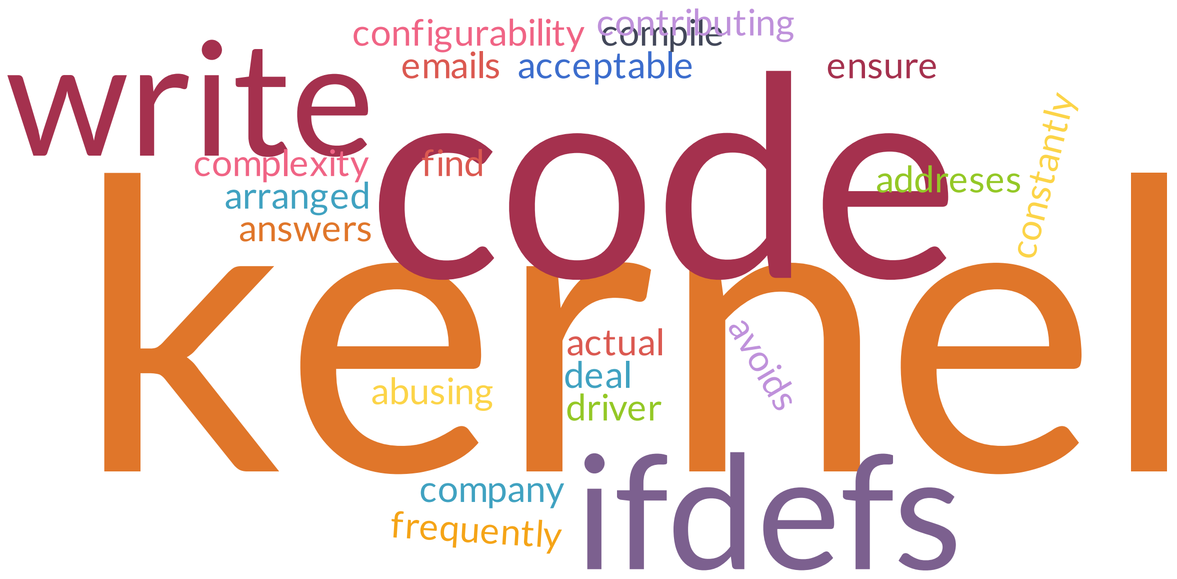
21. Are you working for the Linux kernel as part of your job at your company?



Value	Percent		Count
Yes, for my company	34.69%	🟪	17
No, privately	26.53%	🟦	13
Both, for the company and privately	38.78%	🟩	19
		Total	**49**

Statistics	
Total Responses	49
Hidden	25
Skipped	0

22. If you have additional comments or recommendations for us, we'd be happy to hear about them. Additionally, you can also contact Rodrigo Queiroz, rqueiroz@gsd.uwaterloo.ca (student researcher) or Michael Godfrey, migod@uwaterloo.ca (faculty supervisor).



Count	Response
1	(like a lot of people my kernel contributions are under a variety of addreses so trying to tie patches to emails is trickier than it might appear!)
1	As the maintainer of the Linux "tiny" tree, this is a topic that I deal with very frequently, as my primary job is making it possible to compile more code out of the kernel and make it smaller. I constantly have to deal with the tension between the complexity of #ifdef and the configurability of the kenel. And I have to ensure that my code and the code of others contributing to tinification avoids abusing #ifdef, both to keep the code itself reasonable and to make sure maintainers will find it acceptable.
1	Survey was sometimes unclear whether it was talking about: Kernel vs non-kernel code. ifdefs+scattered code vs scatteded code If you look at my actual contributions to the kernel, you will see they are pretty insignificant - so perhaps my answers are moot, however I do write highly portable .c code professionally (supporting more platforms than the linux kernel) so perhaps my opinions still matter.
1	Unfortunately, working for a company, the vast majority of the Linux kernel code I write and maintain is not available publicly. If you'd like to see some of the driver code I maintain that's heavy with ifdefs for multiple Linux kernel versions and other OSes, that can probably be arranged.