

# Subscribe to the Ardan Labs Insider

You'll get our **FREE** Video Series & special offers on upcoming training events along with notifications on our latest blog posts.

email address

SUBSCRIBE

X



LIVE

## Courses Available

LIVE STREAM TRAINING

ENROLL NOW!

([https://www.eventbrite.com/o/ardan-labs-7092394651?utm\\_source=ardan\\_website&utm\\_medium=blog\\_banner&utm\\_campaign=website\\_livestream\\_promo](https://www.eventbrite.com/o/ardan-labs-7092394651?utm_source=ardan_website&utm_medium=blog_banner&utm_campaign=website_livestream_promo))

 List All Posts (<https://www.ardanlabs.com/all-posts>)

 RSS

## The Behavior Of Channels

William Kennedy October 24, 2017



(<mailto:bill@ardanlabs.com>)



(<https://github.com/ardanlabs/gotraining>)



(<https://twitter.com/goinggodotnet>)

## Introduction

When I started to work with Go's channels for the first time, I made the mistake of thinking about channels as a data structure. I saw channels as a queue that provided automatic synchronized access between goroutines. This structural understanding caused me to write a lot of bad and complicated concurrent code.

I learned over time that it’s best to forget about how channels are structured and focus on how they behave. So now when it comes to channels, I think about one thing: **signaling**. A channel allows one goroutine to signal another goroutine about a particular event. Signaling is at the core of everything you should be doing with channels. Thinking of channels as a signaling mechanism will allow you to write better code with well defined and more precise behavior.

To understand how signaling works, we must understand its three attributes:

- Guarantee Of Delivery
- State
- With or Without Data

These three attributes work together to create a design philosophy around signaling. After I discuss these attributes, I will provide a number of code examples that demonstrate signaling with these attributes applied.

## Guarantee Of Delivery

The Guarantee Of Delivery is based on one question: “Do I need a guarantee that the signal sent by a particular goroutine has been received?”

In other words, given this example in listing 1:

### Listing 1

```
01 go func() {
02     p := <-ch // Receive
03 }()
04
05 ch <- "paper" // Send
```

Does the sending goroutine need a guarantee that the `paper` being sent over the channel on line 05 was received by the goroutine on line 02 before moving on?

Based on the answer to this question, you will know which of the two types of channels to use: **Unbuffered** or **Buffered**. Each channel provides a different behavior around guarantees of delivery.

Figure 1 : Guarantee Of Delivery

	Unbuffered	Buffered
Delivery	Guaranteed	Not Guaranteed

Guarantees are important, and, if you don't think so, I have a ton of things I want to sell you. Of course, I'm trying to make a joke, but don't you get nervous when you don't have guarantees in life? Having a strong understanding of whether or not you need a guarantee is crucial when writing concurrent software. As we continue, you'll learn how to decide.

## State

The behavior of a channel is directly influenced by its current State. The state of a channel can be **nil**, **open** or **closed**.

Listing 2 below shows how to declare or place a channel into each of these three states.

### Listing 2

```
// ** nil channel

// A channel is in a nil state when it is declared to its zero value
var ch chan string

// A channel can be placed in a nil state by explicitly setting it to nil.
ch = nil


// ** open channel

// A channel is in a open state when it's made using the built-in function make
ch := make(chan string)


// ** closed channel

// A channel is in a closed state when it's closed using the built-in function
close(ch)
```

The state determines how the **send** and **receive** operations behave.

*Signals are sent and received through a channel. Don't say read/write because channels don't perform I/O.*

### Figure 2 : State

	NIL	Open	Closed
Send	Blocked	Allowed	Panic
Receive	Blocked	Allowed	Allowed

When a channel is in a **nil** state, any send or receive attempted on the channel will block. When a channel is in an **open** state, signals can be sent and received. When a channel is placed into a **closed** state, signals can no longer be sent but it's still possible to receive signals.

These states will provide the different behaviors you need for the different situations you encounter. When combining **State** with **Guarantee Of Delivery**, you can begin to analyze the costs/benefits you are incurring as a result of your design choices. In many cases, you will also be able to quickly spot bugs just by reading the code, because you understand how the channel is going to behave.

## With and Without Data

The last signaling attribute that needs to be taken into account is whether you need to signal **with** or **without** data.

You signal **with** data by performing a send on a channel.

### Listing 3

```
01 ch <- "paper"
```

When you signal with data, it's usually because:

- A goroutine is being asked to start a new task.
- A goroutine reports back a result.

You signal **without** data by closing a channel.

### Listing 4

```
01 close(ch)
```

When you signal without data, it's usually because:

- A goroutine is being told to stop what they are doing.
- A goroutine reports back they are done with no result.

- A goroutine reports that it has completed processing and shut down.

*There are exceptions to these rules, but these are the major use cases and the ones we will focus on in this post. I would consider exceptions to these rules to be an initial code smell.*

One benefit of signaling without data is a single goroutine can signal many goroutines at once. Signaling with data is always a 1 to 1 exchange between goroutines.

## Signaling With Data

When you are going to signal with data, there are three channel configuration options you can choose depending on the type of guarantee you need.

Figure 3 : Signaling With Data

	Guarantee	No Guarantee	Delayed Guarantee
Channel	Unbuffered	Buffered >1	Buffered =1

The three channel options are **Unbuffered**, **Buffered >1** or **Buffered =1**.

- **Guarantee**
  - An **Unbuffered** channel gives you a **Guarantee** that a signal being sent has been received.
    - Because the **Receive** of the signal **Happens Before** the **Send** of the signal completes.
- **No Guarantee**
  - A **Buffered** channel of size **>1** gives you **No Guarantee** that a signal being sent has been received.
    - Because the **Send** of the signal **Happens Before** the **Receive** of the signal completes.
- **Delayed Guarantee**
  - A **Buffered** channel of size **=1** gives you a **Delayed Guarantee**. It can guarantee that the previous signal that was sent has been received.
    - Because the **Receive** of the **First Signal**, **Happens Before** the **Send** of the **Second Signal** completes.

*The size of the buffer must never be a random number, It must always be calculated for some well defined constraint. There is no infinity in computing, everything must have some well defined constraint whether that is time or space.*

# Signaling Without Data

Signaling without data is mainly reserved for cancellation. It allows one goroutine to signal another goroutine to cancel what they are doing and move on. Cancellation can be implemented using both Unbuffered and Buffered channels, but using a Buffered channel when no data will be sent is a code smell.

Figure 4 : Signaling Without Data

	First Choice	Second Choice	Smell
Channel	context.Context	Unbuffered	Buffered

The built-in function `close` is used to signal without data. As explained above in the **State** section, you can still receive signals on a channel that is closed. In fact, any receive on a closed channel will not block and the receive operation always returns.

In most cases you want to use the standard library `context` package to implement signaling without data. The `context` package uses an Unbuffered channel underneath for the signaling and the built-in function `close` to signal without data.

*If you choose to use your own channel for cancellation, rather than the context package, your channel should be of type `chan struct{}`. It is the zero-space, idiomatic way to indicate a channel used only for signalling.*

## Scenarios

With these attributes in place, the best way to further understand how they work in practice is to run through a series of code scenarios. I like thinking of goroutines as people when I am reading and writing channel based code. This visualization really helps, and I will use it as an aid below.

### Signal With Data - Guarantee - Unbuffered Channels

When you need to know that a signal being sent has been received, two scenarios come into play. These are **Wait For Task** and **Wait For Result**.

#### Scenario 1 - Wait For Task

Think about being a manager and hiring a new employee. In this scenario, you want your new employee to perform a task but they need to wait until you are ready. This is because you need to hand them a piece of paper before they start.

#### Listing 5

<https://play.golang.org/p/BnVEHRCcdh> (<https://play.golang.org/p/BnVEHRCcdh>)

```

01 func waitForTask() {
02     ch := make(chan string)
03
04     go func() {
05         p := <-ch
06
07         // Employee performs work here.
08
09         // Employee is done and free to go.
10     }()
11
12     time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
13
14     ch <- "paper"
15 }

```

On line 02 in listing 5, an Unbuffered channel is created with the attribute that `string` data will be sent with the signal. Then on line 04, an employee is hired and told to wait for your signal on line 05 before doing their work. Line 05 is the channel receive, causing the employee to block while waiting for the piece of paper you will send. Once the paper is received by the employee, the employee performs the work and then is done and free to go.

You as the manager are working concurrently with your new employee. So after you hire the employee on line 04, you find yourself (on line 12) doing what you need to do to unblock and signal the employee. Note, it was unknown just how long it would take to prepare this piece of paper you need to send.

Eventually you are ready to signal the employee. On line 14, you perform a signal with data, the data being that piece of paper. Since an Unbuffered channel is being used, you get a guarantee that the employee has received the paper once your send operation completes. The receive happens before the send.

*Technically all you know is that the employee has the paper by the time your channel send operation completes. After both channel operations, the scheduler can choose to execute any statement it wants. The next line of code that is executed either by you or the employee is nondeterministic. This means using print statements can fool you about the order of things.*

## Scenario 2 - Wait For Result

In this next scenario things are reversed. This time you want your new employee to perform a task immediately when they are hired, and you need to wait for the result of their work. You need to wait because you need the paper from them before you can continue.

### Listing 6

<https://play.golang.org/p/VFAWHxIQTP> (<https://play.golang.org/p/VFAWHxIQTP>)

```

01 func waitForResult() {
02     ch := make(chan string)
03
04     go func() {
05         time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
06
07         ch <- "paper"
08
09         // Employee is done and free to go.
10     }()
11
12     p := <-ch
13 }

```

On line 02 in listing 6, an Unbuffered channel is created with the attribute that `string` data will be sent with the signal. Then on line 04, an employee is hired and is immediately put to work. After you hire the employee on line 04, you find yourself next on line 12 waiting for the paper report.

Once the work is completed by the employee on line 05, they send the result to you on line 07 by performing a channel send with data. Since this is an Unbuffered channel, the receive happens before the send and the employee is guaranteed that you have received the result. Once the employee has this guarantee, they are done and free to go. In this scenario, you have no idea how long it is going to take the employee to finish the task.

### Cost/Benefit

An Unbuffered channel provides a guarantee that a signal being sent was received. This is great, but nothing is free. The cost of this guarantee is unknown latency. In the **Wait For Task** scenario, the employee has no idea how long it's going to take for you to send that paper. In the **Wait For Result** scenario, you have no idea how long it's going to take the employee to send you that result.

In both scenarios, this unknown latency is something we have to live with because the guarantee is required. The logic doesn't work without this guaranteed behavior.

## Signal With Data - No Guarantee - Buffered Channels >1

When you don't need to know that a signal being sent has been received, these two scenarios come into play: **Fan Out** and **Drop**.

A Buffered channel has a well defined space that can be used to store the data being sent. So how do you decide how much space you need? Answer these questions:

- Do I have a well defined amount of work to be completed?
  - How much work is there?
- If my employee can't keep up, can I discard any new work?
  - How much outstanding work puts me at capacity?
- What level of risk am I willing to accept if my program terminates unexpectedly?
  - Anything waiting in the buffer will be lost.



If these questions don't make sense for the behavior you are modeling, it's a code smell that using a Buffered channel any larger than 1 is probably wrong.

## Scenario 1 - Fan Out

A fan out pattern allows you to throw a well defined number of employees at a problem who work concurrently. Since you have one employee for every task, you know exactly how many reports you will receive. You can make sure there is the right amount of space in your box to receive all those reports. This has the benefit of your employees not needing to wait for you to submit their report. They do however need to each take a turn placing the report in your box if they arrive at the box at or near the same time.

Imagine you are the manager again but this time you hire a team of employees. You have an individual task you want each employee to perform. As each individual employee finishes their task, they need to provide you with a paper report that must be placed in your box on your desk.

### Listing 7

[https://play.golang.org/p/8Hlt2sabs\\_](https://play.golang.org/p/8Hlt2sabs_) ([https://play.golang.org/p/8Hlt2sabs\\_](https://play.golang.org/p/8Hlt2sabs_))

```
01 func fanOut() {
02     emps := 20
03     ch := make(chan string, emps)
04
05     for e := 0; e < emps; e++ {
06         go func() {
07             time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
08             ch <- "paper"
09         }()
10     }
11
12     for emps > 0 {
13         p := <-ch
14         fmt.Println(p)
15         emps--
16     }
17 }
```

On line 03 in listing 7, a Buffered channel is created with the attribute that `string` data will be sent with the signal. This time the channel is created with 20 buffers thanks to the `emps` variable declared on line 02.

Between lines 05 through 10, 20 employees are hired and they immediately get to work. You have no idea how long each employee is going to take on line 07. Then on line 08, the employees send the paper report but this time the send does not block waiting for a receive. Since there is room in the box for each employee, the send on the channel is only competing with other employees that may want to send their report at or near the same time.

The code between lines 12 through 16 is all you. This is where you wait for all 20 employees to finish their work and send their report. On line 12, you are in a loop and on line 13 you are blocked in a channel receive waiting for your reports. Once a report is received, the report is printed on line 14 and the local counter variable is decremented to indicate an employee is done.

## Scenario 2 - Drop

A drop pattern allows you to throw work away when your employee(s) are at capacity. This has the benefit of continuing to accepting work from your clients and never applying back pressure or latency in the acceptance of that work. The key here is knowing when you are truly at capacity so you don't under or over commit to the amount of work you will attempt to get done. Usually integration testing or metrics is what you need to help you identify this number.

Imagine you are the manager again and you hire a single employee to get work done. You have an individual task you want the employee to perform. As the employee finishes their task you don't care to know they are done. All that's important is whether you can or can't place new work in the box. If you can't perform the send, then you know your box is full and the employee is at capacity. At this point the new work needs to be discarded so things can keep moving.

### Listing 8

<https://play.golang.org/p/PhFUN5itiv> (<https://play.golang.org/p/PhFUN5itiv>)

```
01 func selectDrop() {
02     const cap = 5
03     ch := make(chan string, cap)
04
05     go func() {
06         for p := range ch {
07             fmt.Println("employee : received :", p)
08         }
09     }()
10
11     const work = 20
12     for w := 0; w < work; w++ {
13         select {
14             case ch <- "paper":
15                 fmt.Println("manager : send ack")
16             default:
17                 fmt.Println("manager : drop")
18         }
19     }
20
21     close(ch)
22 }
```

On line 03 in listing 8, a Buffered channel is created with the attribute that `string` data will be sent with the signal. This time the channel is created with 5 buffers thanks to the `cap` constant declared on line 02.

Between lines 05 through 09 a single employee is hired to handle the work. A `for range` is used for the channel receive. Every time a piece of paper is received it is processed on line 07.

Between lines 11 through 19 you attempt to send 20 pieces of paper to your employee. This time a `select` statement is used to perform the send inside the first `case` on line 14. Since the `default` clause is being used inside the `select` on line 16, if the send is going to block because there is no more room in the buffer, the send is abandoned by executing line 17.

Finally on line 21, the built-in function `close` is called against the channel. This will signal without data to the employee they are done and free to go once they completed their assigned work..

## Cost/Benefit

A Buffered channel greater than 1 provides no guarantee that a signal being sent is ever received. There is a benefit of walking away from this guarantee, which is the reduced or no latency in the communication between two goroutines. In the **Fan Out** scenario, there is a buffer space for each employee that will be sending a report. In the **Drop** scenario, the buffer is measured for capacity and if capacity is reached work is dropped so things can keep moving.

In both options, this lack of a guarantee is something we have to live with because the reduction in latency is more important. The requirement of zero to minimum latency doesn't pose a problem to the overall logic of the system.

## Signal With Data - Delayed Guarantee - Buffered Channel 1

When it's necessary to know if the previous signal that was sent has been received before sending a new signal, the **Wait For Tasks** scenario come into play.

### Scenario 1 - Wait For Tasks

In this scenario you have a new employee but they are going to do more than just one task. You are going to feed them many tasks, one after the other. However, they must finish each individual task before they can start a new one. Since they can only work on one task at a time there could be latency issues between the handoff of work. If the latency could be reduced without losing the guarantee that the employee is working on the next task, it could help.

This is where a Buffered channel of 1 has benefit. If everything is running at the expected pace between you and the employee, neither of you will need to wait for the other. Every time you send a piece of paper, the buffer is empty. Every time your employee reaches for more work, the buffer is full. It is a perfect symmetry of work flow.

The best part is this. If at any time you attempt to send a piece of paper and you can't because the buffer is full, you know your employee is having a problem and you stop. This is where that delayed guarantee comes in. When the buffer is empty and you perform the send, you have the guarantee that your employee has taken the last piece of work you sent. If you perform the send and you can't, you have the guarantee they haven't.

### Listing 9

<https://play.golang.org/p/4pcuKCcAK3> (<https://play.golang.org/p/4pcuKCcAK3>)

```

01 func waitForTasks() {
02     ch := make(chan string, 1)
03
04     go func() {
05         for p := range ch {
06             fmt.Println("employee : working :", p)
07         }
08     }()
09
10     const work = 10
11     for w := 0; w < work; w++ {
12         ch <- "paper"
13     }
14
15     close(ch)
16 }

```

On line 02 in listing 9, a Buffered channel of size 1 is created with the attribute that `string` data will be sent with the signal. Between lines 04 through 08 a single employee is hired to handle the work. A `for range` is used for the channel receive. Every time a piece of paper is received it is processed on line 06.

Between lines 10 through 13 you begin to send your tasks to the employee. If your employee can run as fast as you can send, the latency between you two is reduced. But with each send you perform successfully, you have the guarantee that the last piece of work you submitted is being worked on.

Finally on line 15, the built-in function `close` is called against the channel. This will signal without data to the employee they are done and free to go. However, the last piece of work you submitted will be received (flushed) before the `for range` is terminated.

## Signal Without Data - Context

In this last scenario you will see how you can cancel a running goroutine using a `Context` value from the `context` package. This all works by leveraging an Unbuffered channel that is closed to perform a signal without data.

You are the manager one last time and you hire a single employee to get work done. This time you are not willing to wait for some unknown amount of time for the employee to finish. You are on a discrete deadline and if the employee doesn't finish in time, you are not willing to wait.

### Listing 10

<https://play.golang.org/p/6GQbN5Z7vC> (<https://play.golang.org/p/6GQbN5Z7vC>)

```

01 func withTimeout() {
02     duration := 50 * time.Millisecond
03
04     ctx, cancel := context.WithTimeout(context.Background(), duration)
05     defer cancel()
06
07     ch := make(chan string, 1)
08
09     go func() {
10         time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
11         ch <- "paper"
12     }()
13
14     select {
15     case p := <-ch:
16         fmt.Println("work complete", p)
17
18     case <-ctx.Done():
19         fmt.Println("moving on")
20     }
21 }

```

On line 02 in listing 10, a duration value is declared which represents how long the employee will have to finish the task. This value is used on line 04 to create a `context.Context` value with a timeout of 50 milliseconds. The `WithTimeout` function from the `context` package returns a `Context` value and a cancellation function.

The `context` package creates a goroutine that will close the Unbuffered channel associated with the `Context` value once the duration is met. You are responsible for calling the `cancel` function regardless of how things turn out. This will clean things up that have been created for the `Context`. It is ok for the `cancel` function to be called more than once.

On line 05, the `cancel` function is deferred to be executed once this function terminates. On line 07 a Buffered channel of 1 is created, which is going to be used by the employee to send you the result of their work. Then on lines 09 through 12, the employee is hired and immediately put to work. You have no idea how long it is going to take the employee to finish.

Between lines 14 through 20 you use the `select` statement to receive on two channels. The receive on line 15, you wait for the employee to send you their result. The receive on line 18, you wait to see if the `context` package is going to signal that the 50 milliseconds is up. Whichever signal you receive first will be the one processed.

*An important aspect of this algorithm is the use of the Buffered channel of 1. If the employee doesn't finish in time, you are moving on without giving the employee any notice. From the employee perspective, they will always send you the report on line 11 and they are blind if you are there or not to receive it. If you use an Unbuffered channel, the employee will block forever trying to send you the report if you move on. This would create a goroutine leak. So a Buffered channel of 1 is being used to prevent this from happening.*

## Conclusion

The attributes of signaling around guarantees, channel state and sending are important to know and understand when using channels (or concurrency). They will help guide you in implementing the best behavior you need for the concurrent programs and algorithms you are writing. They will help you find bugs and sniff out potentially bad code.

In this post I have shared a few sample programs that show how the attributes of signaling work in different scenarios. There are exceptions to every rule but these patterns are a good foundation to start.

Review these outlines as a summary of when and how to effectively think about and use channels:

### Language Mechanics

- Use channels to orchestrate and coordinate goroutines.
  - Focus on the signaling attributes and not the sharing of data.
  - Signaling with data or without data.
  - Question their use for synchronizing access to shared state.
    - There are cases where channels can be simpler for this but initially question.
- Unbuffered channels:
  - Receive happens before the Send.
  - Benefit: 100% guarantee the signal has been received.
  - Cost: Unknown latency on when the signal will be received.
- Buffered channels:
  - Send happens before the Receive.
  - Benefit: Reduce blocking latency between signaling.
  - Cost: No guarantee when the signal has been received.
    - The larger the buffer, the less guarantee.
    - Buffer of 1 can give you one delayed send of guarantee.
- Closing channels:
  - Close happens before the Receive (like Buffered).
  - Signaling without data.
  - Perfect for signaling cancellations and deadlines.
- nil channels:
  - Send and Receive block.
  - Turn off signaling
  - Perfect for rate limiting or short term stoppages.

### Design Philosophy

- If any given Send on a channel CAN cause the sending goroutine to block:

- Not allowed to use a Buffered channel larger than 1.
  - Buffers larger than 1 must have reason/measurements.
- Must know what happens when the sending goroutine blocks.
- If any given Send on a channel WON'T cause the sending goroutine to block:
  - You have the exact number of buffers for each send.
    - Fan Out pattern
  - You have the buffer measured for max capacity.
    - Drop pattern
- Less is more with buffers.
  - Don't think about performance when thinking about buffers.
  - Buffers can help to reduce blocking latency between signaling.
    - Reducing blocking latency towards zero does not necessarily mean better throughput.
    - If a buffer of one is giving you good enough throughput then keep it.
    - Question buffers that are larger than one and measure for size.
    - Find the smallest buffer possible that provides good enough throughput.

## Go Training

We have taught Go to thousands of developers all around the world since 2014. There is no other company that has been doing it longer and our material has proven to help jump start developers 6 to 12 months ahead of their knowledge of Go. We know what knowledge developers need in order to be productive and efficient when writing software in Go.

Our classes are perfect for both experienced and beginning engineers. We start every class from the beginning and get very detailed about the internals, mechanics, specification, guidelines, best practices and design philosophies. We cover a lot about "if performance matters" with a focus on mechanical sympathy, data oriented design, decoupling and writing production software.

Interested in Ultimate Go Corporate Training and special pricing?

Let's Talk Corporate Training! (mailto:hello@ardanlabs.com?)

Subject=Let's%20Talk%20Ultimate%20Go%20Corporate%20Training%20and%20special%20pricing!)

## Join Our Online Education Program

Our courses have been designed from training over 4,000 engineers since 2013 and they go beyond just being a language course. Our goal is to challenge every student to think about what they are doing and why.

**✚ENROLL NOW (HTTPS://EDUCATION.ARDANLABS.COM)**


[\(/\)](#) [Training \(/training\)](#)  
[Development \(/development\)](#)  
[DevOps \(/devops-consulting\)](#)  
[Staffing \(/staffing\)](#)  
[UI/UX \(/ui-ux\)](#)  
[Machine Learning \(/machine-learning\)](#)


[About \(/about\)](#)  
[Blog \(/blog\)](#)  
[Contact \(/my/contact-us\)](#)  
[Careers \(/careers\)](#)  
[My Lab \(/my/lab\)](#)  
[Terms \(/terms-service\)](#)  
[Privacy \(/privacy-policy\)](#)

### Reach out to us

 ([https://www.instagram.com/ardan\\_labs/](https://www.instagram.com/ardan_labs/))

 (<https://twitter.com/ardanlabs>)

 (<https://github.com/ardanlabs>)

 (888) 72 ARDAN  
888 722-7326

[info@ardanlabs.com](mailto:info@ardanlabs.com) (mailto:info@ardanlabs.com)