

FRC Tips

by BoVLB

<https://bovlb.github.io/frc-tips/>

Table of Contents

FRC Tips	1
by BoVLB	1
https://bovlb.github.io/frc-tips/	1
Table of Contents	2
FRC tips	4
Burnout	5
Choosing a current limit	5
Current limiting	5
TalonSRX	5
TalonFX (including Falcon 500)	5
CAN SparkMAX	6
Victor SPX	6
Temperature Control	6
CAN SparkMAX	6
TalonSRX and TalonFX (including Falcon 500)	7
See also	7
CAN bus	8
What is your CAN bus utilization?	8
Check your wiring	8
Adjusting frame rates	8
REV Spark MAX	8
CTRE Phoenix (e.g. Talon/Falcon, Pigeon, CANcoder)	8
Motor controllers (Talon/Falcon)	8
Switch motors to PWM	9
Additional hardware	9
CANivore	9
References and further reading	9
Coast mode	10
Background	10
Method	10
Drive sybsystem	10
SparkMAX	10
Talon FX/SRX (including Falcon 500)	10
Set brake mode on init	11
Create trigger	11
References	11
Ramps	12
Slew Rate Limiter	12
References	12
Suggestions for FRC Safety Captains	14
Binder	14
General responsibilities at an event	14
Judged awards	14
Special event activities	14
Further reading	14
Youth Protection Policy	15
Commands	16
void initialize()	16
void execute()	16
boolean isFinished()	16
void end(boolean interrupted)	16
Command groups	17
SequentialCommandGroup	17
ParallelCommandGroup	17
ParallelRaceGroup	17
ParallelDeadlineGroup	17
Commands used in groups	17
Runnable wrappers	17
Command decorators	17
Running commands	18
Triggers	18
Default commands	18
Autonomous commands	19
Esoteric commands	19
See also	19
CommandScheduler	20
CommandScheduler.getInstance().run()	20

Table of Contents	2 of 2
Trigger methods	20
Command.schedule()	20
Command.cance()	20
Putting it all together	20
See also	20
Function References, Lambda Functions, and more	22
Function References	22
Lambda Expressions	22
Method Reference Operator	22
Interfaces	23
Suppliers	23
Runnables	24
Consumers	24
Callables	25
See also	25
Best Practices for Command-Based Programming	26
Summary	26
Add command factories	26
Add triggers	27
Bind commands to triggers	28
Combining Triggers	28
Binding Triggers to Commands	29
Combining commands	29
Default Commands	29
Autonomous routines	30
Exceptions	30
Pose Estimation	30
Performance	31
Incremental Adoption	31
Subsystem Periodic Methods	31
References	31
Acknowledgements	31
Some useful FRC links	32
FIRST	32
WPILIB	32
CTRE (Cross The Road Electronics)	32
REV Robotics	32
Other	32
Selected teams	32

FRC tips

In the course of volunteering with an FRC team (and at various events), I see the same problems come around again and again. Some of them have excellent resources already available to provide help; others take a little digging.

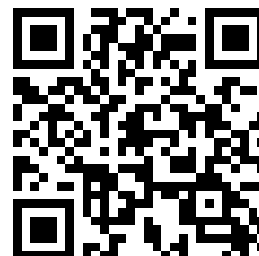
This project is a small collection of tips I've gathered to save redoing the same research effort. Largely I'm just repeating stuff other people already said. Example code is all in Java.

- [CAN bus](#): What is my utilization? How much is too much? How do I fix it?
- [Ramps](#): How do I stop my robot from falling over when they drive too fast?
- [Coast mode](#): How do I make my robot stop, stay stopped, and yet be easy to move?
- [Burnout](#): How do you stop your motors from burning out?
- [Safety Captain](#): Help! I just got appointed Safety Captain. What do I do now?
- [Commands](#): Short guide to WPILIB commands, including [lambda functions](#), and [CommandScheduler](#)
- [Links](#): What are some useful resources for FRC?

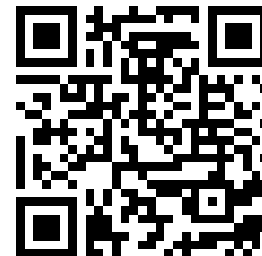
I decided to put this together as a Github repository, partly so I could incorporate code files if I needed to, but mostly to make it easier for others to correct my inevitable mistakes. You can download the entire website as [frc-tips.pdf](#)

Suggestions I have received for future notes:

- How to reset and persist motor controllers
- Some swerve bot gotchas - see [example swervebot](#)
- Mecanum Drive gotchas.
- How to add your first autonomous routine, including for non-command robots
- Checking for errors (e.g. CAN bus devices). Retrying?
- How should I configure and use IP addresses? (see [IP Configurations](#)) How should I wire the Ethernet on the robot?
- Current limits and voltage compensation
- How to debug "No robot code"
- Example code that uses all commands stuff



Burnout



We put motors on our robots because we want to transform electricity into motion. Unfortunately, because of internal inefficiencies and forces/torques that resist motion, much of that electrical energy is instead transformed into heat (and some noise). If a motor becomes too hot, it will fail, usually by burning off the thin enamel coating on the winding wires. This is often referred to as “letting out the magic smoke”.

If a motor is pushing against more force/torque than it is generating, then all of the input energy is transformed into heat. This is called “stalling”. Many motors are “self cooling”, which means that as they turn, air is drawn through the motor to dissipate the heat. Unfortunately, motors are not good at getting rid of heat when stationary, so stalling is the most common way to burn out motors. Don’t make the mistake of thinking that a stationary motor is not drawing any power.

Some motors (e.g. CTRE/VEX Falcon 500) have “thermal protection”, which means that they monitor their own internal temperature and simply stop working until they have cooled down a bit. This is good for protecting your investment, but bad if it happens in the middle of a match. It’s much better not to reach that point.

Here are some common scenarios when a motor might stall and therefore be at risk of burning out:

- A game piece (e.g. a squashy ball) is jammed in the serializer
- A robot experiences aggressive defence and is unable to move
- A limit switch fails on an intake or turret
 - Consider making limit switches “normally closed” (NC) so the more common failure mode (disconnection) is detected quickly.
- A servo motor is mechanically misaligned and cannot reach its set position
- An intake or elevator requires continuous power to stay in position
 - Consider using brake mode on the motors, adding a physical brake, or changing the mechanical design to reduce the torque
- The drive train receives continuous small signals that don’t result in (much) movement
 - While this is usually too small to cause burnout in the typical timescale of an FRC match, it’s a good idea to use [deadband](#) on joystick inputs

While in many cases there are other possible solutions, there are some simple things we can do in software to limit this risk. Primarily we can limit the current on motors. If you look at the manufacturer websites, you can often find charts showing how long a motor will sustain a specific current before failing. For example, the Falcon 500 will hit thermal protection after about 110 seconds at 50A, and 260 seconds at 40A.

Don’t assume that because a motor is on a 40A circuit, its current will never exceed 40A. It often makes sense to use a current limit that is greater than the rating of the fuse/circuitbreaker installed on the circuit.

Choosing a current limit

Before applying current limiting, you should get an idea of what your typical current draw is so that your limits don’t affect normal usage. You can monitor current draw by motors in several ways:

- Looking at the PDP/PDH in SmartDashboard/Shuffleboard.
 - `SmartDashboard.putData(new PowerDistribution());`
- Looking at your [Driver Station logs](#)

In both cases, the first thing you will realise is that you need to know which motor is connected to which circuit. Of course, all the wires on your PDB/PDH are already clearly labelled, but [a cool trick](#) is to use the same CAN bus id as the circuit number. If you do this, it means all of your circuit numbers are already documented in `Constants.java`.

For a drive train, [a useful technique](#) is put the robot on a carpet, touching a wall and then gradually increase the power until the wheels start to slip. Set the current limit to the peak current at the moment when the wheels start to slip.

I haven’t tried it myself, but the [ILITE Drive Train Simulator](#) is supposed to give some useful information about currents.

⋮ Warning The specific current values in the example code below are only for illustration. You should pick your own values.

Current limiting

How you limit the current depends on what sort of motor controller you’re dealing with. When you set a limit, the controller will reduce the control inputs to keep the current at that level.

As always, this sort of intervention makes the robot not do what the driver asks for, so should be applied with caution. If drivers find that current limiting makes it harder to drive the robot then relax (increase) the limits rather than just removing them.

TalonSRX

These controllers allow you to set supply, peak, and continuous current limits. The supply limit is primarily used to prevent breakers from tripping. The peak limit is engaged whenever the current tries to exceed that level. The continuous limit is engaged if the current tries to exceed that level for more than a certain time. See the [documentation](#) for more details.

```
// Use this to stop breakers from tripping
m_motor.configSupplyCurrentLimit(40); // Amperes

// Use these to prevent burnout
m_motor.configPeakCurrentLimit(35); // Amperes
m_motor.configPeakCurrentDuration(200); // Milliseconds
m_motor.configContinuousCurrentLimit(25); // Amperes
m_motor.enableCurrentLimit(true);
```

TalonFX (including Falcon 500)

These controllers allow you to limit supply and stator current. The supply current is what is going to pop the breaker. The stator current is what is going through the windings and will cause burnout. These have two current levels: a higher “threshold” level that activates the limit, and a lower level that the current is limited to when active. See the [documentation](#) for more details.

```
// Use this to stop breakers from tripping
m_motor.configSupplyCurrentLimit(
    new SupplyCurrentLimitConfiguration(
        true, // enable
        35, // current limit in Amperes
        40, // threshold current for activation in Amperes
        0.2 // time exceeding threshold for activation in seconds
    )
);

// Use this to prevent burnout
m_motor.configStatorCurrentLimit(
    new StatorCurrentLimitConfiguration(
        true, // enable
        35, // current limit in Amperes
        40, // threshold current for activation in Amperes
        0.2 // time exceeding threshold for activation in seconds
    )
);
```

CAN SparkMAX

This controller allows you to limit current as a function of the speed of the motor. The actual limit applied is a linear interpolation between the two values. This means that you can handle the stalling case without limiting the non-stalling current. See [the documentation](#) for variants on this method that allow slightly different control models.

```
m_motor.setSmartCurrentLimit(
    10, // stall limit in Amperes
    100 // free speed limit in Amperes
);
```

CANSparkMax also provides `setSecondaryCurrentLimit` which works somewhat differently, but you probably don't want to touch that for an FRC robot.

Victor SPX

The Victor SPX does not provide any current limiting feature.

TODO: Add code snippet on how to limit current based on PDB/PDH reporting.

Temperature Control

Another possible approach is temperature control. Some motor controllers will report the temperature of the motor and you can temporarily stop using a motor when it is too hot. This technique is not widely applicable in the FRC context, but you might use it, say, if it takes continuous power to hold an intake up and the consequences of letting the intake droop are fairly minor.

CAN SparkMAX

```
boolean m_stopped;

@Override
public void initialize() {
    // ...
    m_stopped = false;
}

@Override
public void periodic() {
    double motorTemperature = m_motor.getMotorTemperature()
    if(m_stopped) {
        if(motorTemperature <= 48) {
            m_stopped = false;
        }
    } else if(motorTemperature > 50) {
        m_stopped = true;
    }

    if(m_stopped) {
        m_motor.set(0)
    } else {
        // do stuff
    }
}
```

TalonSRX and TalonFX (including Falcon 500)

The Talon motor controller also has a `getTemperature()` method, but the documentation says that it returns the temperature of the controller, not the motor. This might be useful with integrated controllers like the Falcon 500, but I have not tested it.

See also

- [Dalton's CD reply](#)

CAN bus



What is your CAN bus utilization?

The Driver Station and the DS Logs will show you your CAN bus utilization rate. Often this is a fuzzy line that ranges up to 100%. This doesn't necessarily mean that your CAN bus utilization is actually that high. The real value can be seen by zooming in (in the DS Log Viewer) to find the common, stable value. Typically it will be near the bottom of the fuzzy band.

How much is too high? 70% is fine. If you're going much higher than that, you're likely to be experiencing lost packets and various errors. If not, you should look elsewhere for your problem.

There are four main ways to fix a high CAN utilization. At a competition, they should probably be tried in this order.

- Check your wiring and termination. Noisy CAN bus wiring or a missing terminator will reduce available bandwidth.
- Adjust the frame rates on your motors.
- Switch motors to PWM. This may be appropriate for, say, intake motors, when the precise speed is not important.
- Install additional hardware like the CANivore.

Check your wiring

See [CAN wiring basics](#)

- Do the wiggle check
- Check for bare metal or whiskers
- Check that the PDP is last in chain and has termination set
- Ensure the green and yellow cables are twisted together everywhere
- Use [Phoenix Tuner](#) and [Rev Hardware Client](#) to check device visibility
- Turn the robot off, disconnect both ends, and test each wire for continuity using a multimeter.

Adjusting frame rates

If you are having problems with high CAN bus utilization, consider turning down the frame rate. These are not general recommendations! Don't do it if you're not having trouble!

Motor controllers will typically have multiple types of status frame that can be controlled separately. Status frames have multiple uses:

- Motor safety watchdog: The roboRIO will shutdown any motor it has not heard from in 100ms.
- Following: The follower listens for certain information from the leader.
- Software PID: If you are using software PID, then you need to know motor velocity or position every 20ms.
- Odometry: If you are using dead reckoning, you need to know motor positions every 20ms.

The motor safety watchdog is an important case. It is recommended that you do not set such frames over 45ms. This means that you can drop the occasional frame without triggering it.

If a motor is under power/voltage control, or is using firmware PID control with no follower, then status frames can be less frequent.

REV Spark MAX

The Spark MAX has five different types of periodic status frame, but I believe a typical FRC setup will only use the first three.

- Type 0 is "Applied ... Output" and faults. Default 10ms. I believe this is the one used for the watchdog, so it should not be more than 45ms. I could be wrong, but I think is also what the follower relies on from the leader.
- Type 1 is velocity, temperature, voltage, and current. Default 20ms.
- Type 2 is position. Default 20ms.

E.g.

```
// Maximum reasonable values
leader.setPeriodicFrameRate(PeriodicFrame.kStatus0, 20);
leader.setPeriodicFrameRate(PeriodicFrame.kStatus1, 50);
leader.setPeriodicFrameRate(PeriodicFrame.kStatus2, 50);
follower.setPeriodicFrameRate(PeriodicFrame.kStatus0, 45);
follower.setPeriodicFrameRate(PeriodicFrame.kStatus1, 500);
follower.setPeriodicFrameRate(PeriodicFrame.kStatus2, 500);
```

CTRE Phoenix (e.g. Talon/Falcon, Pigeon, CANcoder)

Motor controllers (Talon/Falcon)

Ten different types, but two important ones:

- Type 1: Applied Motor Output, Fault Information, Limit Switch Information. Default 10ms.
- Type 2: Selected Sensor Position, Selected Sensor Velocity, Brushed Supply Current Measurement, Sticky Fault Information. Default 10ms.


```
// Maximum reasonable values
leader.setStatusFramePeriod(StatusFrameEnhanced.Status_1_General, 20);
leader.setStatusFramePeriod(StatusFrameEnhanced.Status_2_Feedback0, 50);
follower.setStatusFramePeriod(StatusFrameEnhanced.Status_1_General, 45);
follower.setStatusFramePeriod(StatusFrameEnhanced.Status_2_Feedback0, 500);
```

Switch motors to PWM

[WPILIB How to wire PWM cables](#)

```
private final PWMSparkMax m_leftDrive = new PWMSparkMax(0);
private final PWMSparkMax m_rightDrive = new PWMSparkMax(1);
```

Pass to `DifferentialDrive` or call `.set()`.

Similarly for `PWMTalonFX` (Falcon 500) and `PWMTalonSRX`.

[WPILIB Using PWN Motor Controllers](#)

Additional hardware

CANivore

Might help. Introduces its own complexity. Doesn't work with sysid.

[CTRE Canivore Hardware Manual](#)

[CTRE Bring Up: CANivore](#)

```
TalonFX motor = new TalonFX(deviceNumber, canivoreName);
```

References and further reading

- [Rev Spark MAX periodic status frames](#)
- [CTRE Setting status frame periods](#)
- [WPILIB Driver Station log viewer](#)
- [BaseTalon.setStatusFramePeriod](#)
- [CD comment: three types of interaction with a CAN device](#)

Coast mode

This document explains how to set neutral/idle mode so that the robot is safe in use, but moveable when disabled.

Background

When a motor controller is told to setting a motor's power to zero, this is known as idle mode (also known as neutral mode). There are two different flavors of idle mode: coast mode and brake mode.

- Coast mode effectively disconnects the motor wires so that the motor can turn freely. A spinning motor that is set to coast mode will slow down gradually as a result of friction. A stationary motor in coast mode provides little resistant to movement.
- Brake mode effectively connects the motor wires together so that the motor provides "back EMF" that resists motion. A spinning motor in brake mode will slow down very quickly when the power is set to zero. A stationary motor in brake mode will be hard to turn.

So which do we want to use on an FRC robot? The answer, in most cases, is brake mode. A drive train in brake mode will stop quickly when commanded (say before hitting an obstacle). This reduces wear and tear on your practice space and any unobservant humans present. A motor-actuated elevator or climber will, in brake mode, lock and resist descent, meaning you may be able to get away without adding brakes. A drive train stopped on a ramp will stay there even after being disabled. Remember that end game points are often assessed some number of seconds after the end of the match.

The main circumstance where you want your robot to be in coast mode is when you are positioning your robot by hand (while it is turned on, but disabled). It's very hard to move an FRC robot in brake mode. When you're perfecting your autonomous routines, you're going to be spending a lot of time repositioning your robot.

In this case, it is tempting to say that the robot's drive train should be in brake mode while enabled, but go into coast mode when disabled. You might get away with this, but one day you'll disable your robot when it is going at high speed, and then everyone will be unpleasantly surprised that it just keeps on going. This has actually happened in some FRC competition matches, where a robot in motion at the end of the match keeps moving, collides with other robots, and either loses end game points for their alliance or gains fouls as a result.

So what should we do? The best compromise seems to be to enable coast mode only once the robot has been disabled for several seconds, and keep it in brake mode at all other times. That gives it enough time to stop, allows enough time for post-match judging, but still makes it easy to move your robot. Fortunately, with appropriate configuration, idle mode commands can be sent to your motors while the robot is disabled.

Method

So how do we do this? You need to do three things:

- Create a method in your drive subsystem that can set the idle mode to either coast or brake
- Set brake mode in appropriate places
- Set coast after several seconds of being disabled

Drive subsystem

The exact command required will vary depending on your drivetrain and the motor controllers. Here I give code for Spark Max and Talon FX, assuming two motors on each side.

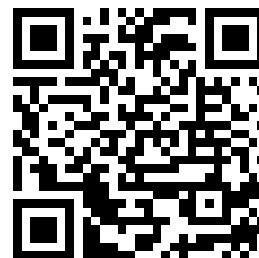
SparkMAX

```

/**
 * Sets idle mode to be either brake mode or coast mode.
 *
 * @param brake If true, sets brake mode, otherwise sets coast mode
 */
public Command setBrakeMode(boolean brake) {
    IdleMode mode = brake ? IdleMode.kBrake : IdleMode.kCoast;
    return Commands.runOnce(() -> { // Instant command will execute our
"initialize" method and finish immediately
        m_leftLeader.setIdleMode(mode);
        m_leftFollower.setIdleMode(mode);
        m_rightLeader.setIdleMode(mode);
        m_rightFollower.setIdleMode(mode);
    }).ignoringDisable(true); // This command can run when the robot is disabled
}

```

Talon FX/SRX (including Falcon 500)



```
public Command setBrakeMode(boolean brake) {
    NeutralMode mode = brake ? NeutralMode.Brake : NeutralMode.Coast;
    return Commands.runOnce(() -> {
        m_leftLeader.setNeutralMode(mode);
        m_leftFollower.setNeutralMode(mode);
        m_rightLeader.setNeutralMode(mode);
        m_rightFollower.setNeutralMode(mode);
    }).ignoringDisable(true);
}
```

Set brake mode on init

In `Robot.java`, in both `autonomousInit` and `teleopInit`, add the following line:

```
m_robotContainer.m_driveSubsystem.setBrakeMode(true).schedule(); // Enable brake mode
```

You may need to change this code, depending on where your drive subsystem is created and stored. You may also need to change `RobotContainer.m_driveSubsystem` to be `public`.

Create trigger

In `Robot.java`, at the end of `robotInit`, add the following code:

```
// Turn brake mode off shortly after the robot is disabled

new Trigger(this::isEnabled) // Create a trigger that is active when the robot is
    .negate() // Negate the trigger, so it is active when the robot is
    disabled
    .debounce(3) // Delay action until robot has been disabled for a certain
    time
    .onTrue( // Finally take action
        m_robotContainer.m_driveSubsystem.setBrakeMode(false)); //
Enable coast mode in drive train
```

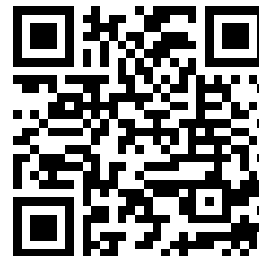
Notes:

- The `Trigger` can be constructed with a `BooleanSupplier`. Here `Robot` has a method `isEnabled` which takes no arguments and returns a `boolean`. Such methods can be treated as a `BooleanSupplier`. The syntax is a little tricky here because we're trying to pass in a class method in the context of this particular instance of `Robot`. We do this using the `this` implicit method variable and the (seldom-used) `::` method reference operator.
- We want to do something when the robot is disabled, but there is no `isDisabled` method, so we have to use `isEnabled` and negate it. This means the trigger will activate whenever the `isEnabled` method returns `false`. Notice that methods like `negate` return a new `Trigger`, so they can be chained in a terse style.
- We don't want to activate this trigger immediately the robot is disabled, but several seconds afterwards. The `debounce` creates a new trigger that does not activate until its input trigger has been consistently active for some number of seconds. (If this is new to you, think about using `Debouncer` the next time you have trouble with noisy beam break sensors.) Choose your own time here. It needs to be long enough that the robot will come to a stop, but not so long that you're standing around waiting for it. It's also a good idea to look at the rulebook and see if the robot has to stay in position on a ramp for some number of seconds.
- Change the call to `setBrakeMode` as necessary, depending on where your drive subsystem can be found.

References

- [Oblara's comment on Chief Delphi that started me on this path](#)
- [WPILIB Convenience Features](#)
- [WPILIB Binding Commands to Triggers](#)
- [CANSparkMax Java doc](#)
- [TalonFX Java doc](#)
- [Trigger Java doc](#)
- [Java 8 - Double Colon \(::\) Operator](#)

Ramps



This document describes ways to stop your robot from falling over in teleoperated driving.

An ideal FRC robot would have a centre of mass that is near the ground, and centred in the frame perimeter. In practice, the various game requirements make this hard to achieve, and you often end up with a top-heavy robot with all the weight at one end. The means that if you accelerate or decelerate too aggressively in the wrong direction, your robot could fall over. Even if the robot doesn't actually fall over, just picking the wheels up could be enough to allow another robot or a game piece to slide underneath.

As with many aspects of robot programming, the answer lies in not always doing what the driver asks for. When the driver slams the stick from full forwards to full reverse, we make the robot response lag very slightly. Ramping is usually measured in terms of the minimum time the robot will take to go between neutral and full power. Good values for this will range between about 0.1s and 0.5s depending on how top-heavy your robot is, and how much lag the driver will tolerate. I recommend that you set the ramp time as high as your driver will permit (but no higher).

You might add the following to `Constants`:

```
final static k_rampTimeSeconds = 0.25;
```

Ramping is generally used for the drive train, although it can sometimes apply to other subsystems. It is also generally used only for telemetric operation; it can also apply to autonomous routines, but that is better handled by setting maximum acceleration in path planning.

There are a number of different ways to implement ramps. [TODO: Add more than one.]

Slew Rate Limiter

The easiest and simplest way to add ramps is using a class called `SlewRateLimiter`. This can be applied directly to your control inputs (e.g. your joystick) inside your arcade drive command.

You probably have an arcade drive command where the `execute` method looks something like:

```
// Existing ArcadeDrive code
double forward = ...; // Probably from -Y on the joystick
double turn = ...; // Probably from X on the joystick

// Do something with forward and turn
// Probably looks like ONE of the following:
m_driveSubsystem.setPower(forward + turn, forward - turn);
m_driveSubsystem.setSpeed((forward + turn) * k_maxSpeed, (forward - turn) *
k_maxSpeed);
m_driveSubsystem.arcadeDrive(forward, turn);
m_driveSubsystem.getDrivetrain().arcadeDrive(forward, turn, true);
```

To add ramping, add a new member variable in the `ArcadeDrive` command:

```
// If the driver moves the joystick too fast, add a little lag
SlewRateLimiter m_filter = new SlewRateLimiter(1.0 /
Constants.k_rampTimeSecond);
```

Note that `SlewRateLimiter` does not take a time; instead it takes a rate (units of change per second). I have found that it's much easier to talk to the drivers about lag time, which is why I use that as the defined constant. Some other ramping techniques take a time directly instead of a rate.

In `ArcadeDrive.execute`, simply replace `forward` with `m_filter.calculate(forward)`, for example:

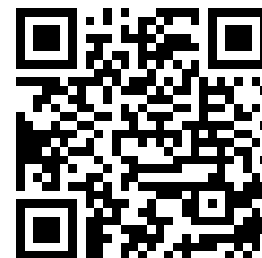
```
// Apply ramping filter to forward control
forward = m_filter.calculate(forward);
```

Note that we're only applying the ramp to the forwards/backwards axis and not the turn. Rapid turns alone are not usually enough to tip the robot. If you find that the robot turns too rapidly, remember that it's common practice to limit the maximum permissible rate of turn. This can be done simply by multiplying the turn value from the joystick by a constant like 0.5.

If you do decide to apply ramping to the turn control, you will need to create a second `SlewRateLimiter`; the filter has internal state, so don't try to use one filter for two data streams.

References

Suggestions for FRC Safety Captains



In 2020 I was a Safety Manager at a regional. I participated in pit inspections, interviews, and judging for the safety award. Here are some suggestions I have for any Safety Captain. These are particularly intended to be a good starting point for someone suddenly appointed as Safety Captain a few weeks or days before competition who doesn't know where to get started.

A team's safety does not come from one person alone. There are things the safety captain can do individually, but real success will come from having a team culture of safety, where people know what they have to do to be safe, and feel comfortable reminding each other.

Your team should be appointing a Safety Captain before Kickoff; see if you can make that happen next year. Does your team's website have a safety page? Do you have a "safety moment" at every meeting, explaining a single safety tip or describing a near miss? You'll never avoid making mistakes, so the important thing is to learn from them.

Binder

You should have a clearly-labelled and well-organised safety binder in the pit. This should contain:

- Relevant FIRST manuals, especially the Safety Manual and Event Rules. You may be the only person on the team to read these, so make sure you pass along any relevant information. Be prepared to demonstrate familiarity with these materials.
- Data sheets for hazardous chemicals (e.g. glue, lubricant, basically any bottles, tubes, or spray cans in the pit). It's fairly easy to find these online and print them out. Manuals for tools you brought to the event. Again, almost every manual can be found online. Search on [msds.com](https://www.msdsonline.com).
- Team safety manuals or training materials. Basic first aid information. How to operate a fire extinguisher (PASS=Pull, aim, squeeze, sweep). Safety guidelines for outreach events.
- Pit safety checklist List of equipment you should have. List of things you should remember to do or not do.
- Event-specific information such as the nearest emergency hospital
- List of people on the team with CPR/First Aid training
- List of people on the team certified to operate certain types of machinery
- Safety event log. Safety audit reports.
- Copies of any posters or flyers you are handing out.

General responsibilities at an event

You will be expected to attend daily Safety Captain meetings, probably early each morning. This is a good venue to speak up and raise any questions you may have, offer suggestions, raise general safety concerns, or offer praise. Keep your eyes open during the day to see if you can find something to share. You may be asked to vote for a daily Safety Captain award.

At an event, you are also supposed to be a safety ambassador for FIRST. If you see an issue, you should draw it to someone's attention. (Be careful if you find yourself in the position of criticizing a specific team. Remember you can involve one of your mentors.) Direct people with injuries to the EMT desk at pit admin. Offer (with any teammates you can gather) to take a turn on the entrance to give the volunteers there a break.

Judged awards

Since 2022, there is no longer a separate judged safety award. Instead safety is a consideration for all judged awards. This means that any judge may perform a pit safety inspection, interview the safety captain, or review your safety binder.

Some specific things judges may look for in a pit inspection: First aid kit, battery spill kit, data sheets, daisy-chained power strips, loose hair or clothing, food, any unsafe practices. Judges and other volunteers may also notice if people on your team are forgetting safety glasses, crowding their pit, playing football in the aisle, transporting a robot improperly (e.g. without human first), or lifting improperly (lift with the knees, not the back). Even if they don't say anything, they may note your team number and pass it along. Don't make the mistake of thinking that actions outside of the judge interview can't lose your team its chance at a judged award.

If you're interviewed about safety, resist the temptation to depict the team as having a perfect safety record. No-one will believe this. It's better to be able to talk about how you deal with the safety issues that arise. What does the team do after an injury or a near miss? Is there a safety event log? What have you had to change this season to improve safety?

Does this year's game or your team's robot design present any specific safety challenges? Have you developed any interesting approaches to mitigate them?

If someone comes to your pit and asks a random team member where the Safety Captain is, what will they say? (Several teams I spoke to had no idea what I was talking about.) This should be covered as part of "judge talking". Also, you should have some visible identification as Safety Captain, like a button, a cape, a hat, or a custom-decorated high-visibility vest. (Try to avoid selecting something that resembles a FIRST volunteer uniform, like an orange or yellow baseball cap, or a red or black vest. There's no rule against it, but you never want to make a judge frown. Also, you may want to avoid wearing retroreflective tape near the field.)

Special event activities

A lot of teams have special things they do to demonstrate their commitment to safety. Some examples:

- Putting up safety posters in other team's pits and handing out safety flyers. This should be done with caution. If another team wants to put posters and flyers in your pit, then by all means accept these as it shows good safety spirit. Don't do this yourself unless you are truly going to do a good job of it, as it easily becomes just more trash (with your team's name on it).
- Videos, websites, apps. This is a lot of work but it does impress judges. If you have a monitor in your pit, you can use it to show safety videos, including [ones you didn't make](#).
- [Safety mascots](#)
- Handing out first aid kits or other safety equipment with your team's logo on them.

Further reading

Many other people have offered advice, including:

- [Safety Captain purpose? - General Forum - Chief Delphi](#)
- [Safety Captain - Mrs. McKeon](#)
- [Advice for being a safety captain : r/FRC](#)
- [Safety - FRC TEAM 2471](#)

Youth Protection Policy

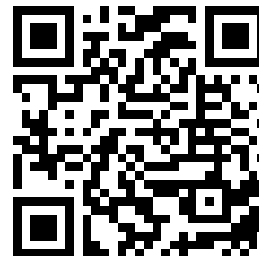
YPP is an important part of safety, but this is an area where adult mentors should be strongly involved. For everyone's protection, both adults and students should avoid being in a situation where they are alone one-on-one, or where there is extensive private communication. Students should know that they ought to report any situation that makes them uncomfortable, whether because of another student or a mentor.

- [253's YPP guide](#)
- [FIRST's Youth Protection Program](#)

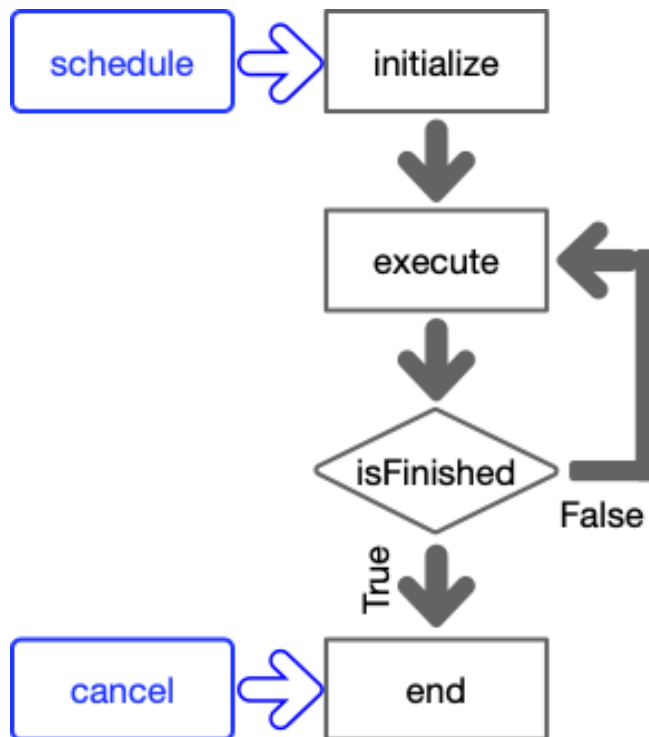
Commands

Although you can avoid it in some simple cases, doing anything complex with your FRC robot will involve creating commands. These respond to joysticks and buttons, run your autonomous routines, and do other maintenance tasks.

In addition to the usual constructor, commands have four lifecycle methods: `initialize`, `execute`, `isFinished`, and `end`. These methods are called by the `command scheduler` (and never by you). By overriding the implementation of these methods, you can change the behaviour of the command.



```
void
```



The scheduler calls the four lifecycle methods of a command. This starts with `initialize` when the command is first scheduled, then `execute` and `isFinished` are called in alternation. Finally `end` is called either when `isFinished` returns true, or when the command is interrupted.

```
initialize()
```

- Called once whenever a command is scheduled (including default commands).
- Use this to do anything your command needs to do once, such as running motors at constant speed, or initializing variables. It's a good idea to print a log message in this command.
- Default implementation does nothing.

```
void execute()
```

- Called every cycle for scheduled commands, in alternation with `isFinished`.
- Use this to do anything your command needs to do dynamically, like responding to joysticks or sensors, and updating internal state. You can also use this to update SmartDashboard (although that is usually better done in `subsystem periodic`)
- Default implementation does nothing.

```
boolean isFinished()
```

- Called every cycle for scheduled commands, in alternation with `execute`.
- Use this to tell the scheduler when your command is complete.
- Default implementation in `Command` returns `false`, so a command will run forever unless interrupted or canceled, but may be overridden (say in `InstantCommand`).

```
void end(boolean interrupted)
```

- Called when a command is descheduled, which means that `isFinished` has returned `true`, or that the command has been interrupted or cancelled. It's a good idea to print a log message in this command.
- Use this to tidy up after the command. The typical usage is to stop motors.
- Default implementation does nothing.

These methods (as well as `subsystem periodic` methods and any `Triggers` you have created) all run in a single shared thread, which is expected to run fifty times a second. This means that they all share a total time budget of 20ms. It is important that these commands execute quickly, so avoid using any long-running loops, sleeps, or timeouts. The scheduler will only run one command lifecycle method (`initialize`, `isFinished`, `execute`, `end`) or `subsystem periodic` at a time, so if you stay within this framework you don't have to worry about being thread-safe.

Generally commands exist in order to do something with a subsystem, like run motors. It's very important that you never have two commands trying to control the same motor. WPILIB's solution to this is called "subsystem requirements". Generally you will pass the subsystems into the command's constructor (along with any other configuration) to be stored for later use. In the constructor, you should also call `addRequirements(...)` with any subsystems used by the command. In some complex cases, you may use a subsystem without requiring it, say because you are only reading sensor data and setting any motor speeds.

A programmer needs to be familiar with the various command-related tricks available in WPILIB. I've divided them here into six groups:

- [Command groups](#): Classes that take one or more commands and execute them all.
- [Commands for use in groups](#): Commands that are useful when using command groups.
- [Runnable wrappers](#): Classes that turn runnables into commands
- [Command decorators](#): Methods provided by all commands to connect or change them.
- [Running commands](#): How to run a command
- [Esoteric commands](#): Commands that are used only in specialized circumstances

These might seem a little complex and daunting, but the good news is that if you use them effectively your code will become simpler and easier to read. There are many subtle gotchas about combining commands, and these help you to navigate them safely.

Command groups

These classes group together one or more commands and execute them all in some order. They inherit the subsystem requirements of all of their sub-commands. The sub-commands can be specified either in the constructor, or by subclassing and using `addCommands`.

SequentialCommandGroup

- Runs the sub-commands in sequence.
- See also `andThen` and `beforeStarting`.

ParallelCommandGroup

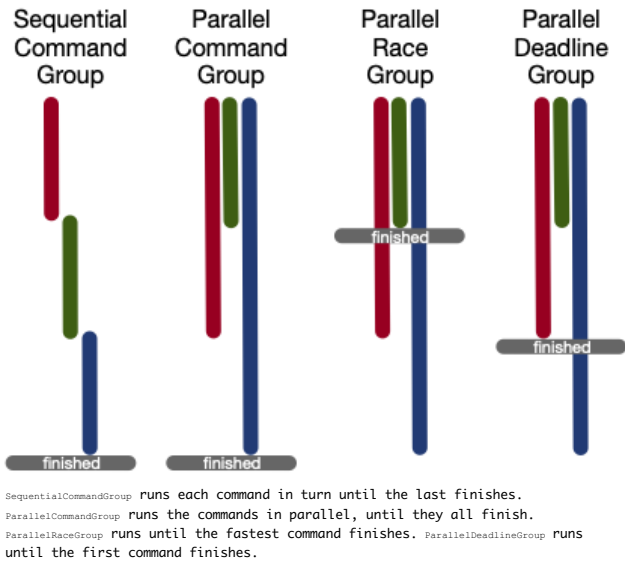
- Runs the sub-commands in parallel.
- Finishes when the slowest sub-command is finished.
- See also the decorator `alongWith`.

ParallelRaceGroup

- Runs the sub-commands in parallel.
- Finishes when the fastest sub-command is finished.
- See also the decorator `raceWith`.

ParallelDeadlineGroup

- Runs the sub-commands in parallel.
- Finishes when the first command in the list is finished.
- See also the decorator `deadlineWith`.



Commands used in groups

The following commands are useful to build command groups. Some of them take commands as arguments, and their subsystem requirements are inherited.

- **ConditionalCommand**: Given a condition (evaluated in `initialize`), runs one of two sub-commands. See also the decorator `unless`.
- **SelectCommand**: Takes a mapping from keys to commands, and a key selector. At `initialize`, the key selector is executed and then one of the sub-commands is run.
- **ProxyCommand**: This behaves exactly like the underlying command except that subsystem requirements are not shared between the child and parent commands. See also the decorator `asProxy`. Warning: `ProxyCommand` works by scheduling the command independently and waiting for it to complete. A consequence of this is that any scheduled commands with overlapping requirements will be interrupted. If this includes the command group that is using `ProxyCommand`, then the proxy command will also be canceled.
- **RepeatCommand**: Run the sub-command until it is finished, and then start it running again. See also the decorator `repeatedly`.
- **WaitCommand**: Insert a delay for a specific time.
- **WaitUntilCommand**: Insert a delay until some condition is met.

Runnable wrappers

Here are some wrappers that turn runnables (e.g. [lambda expressions](#)) into commands. These can be used in command groups, but they are also used in `RobotContainer` to create command on-the-fly. When using these methods, please remember to add the subsystem(s) as the last parameter(s) to make subsystem requirements work correctly.

- **InstantCommand**: The given runnable is used as the `initialize` method, there is no `execute` or `end`, and `isFinished` returns `true`. You will also sometimes inherit from `InstantCommand` instead of `BaseCommand`.
- **RunCommand**: The given runnable is used as the `execute` method, there is no `initialize` or `end`, and `isFinished` returns `false`. Often used with a decorator that adds an `isFinished` condition.
- **StartEndCommand**: The given runnables are used as the `initialize` and `end` methods, there is no `execute`, and `isFinished` returns `false`. Commonly used for commands that start and stop motors.
- **FunctionalCommand**: Allows you you set all four life-cycle methods. Not used if one of the above will suffice.

Wrapper	initialize	execute	end	isFinished
InstantCommand	arg 1	empty	empty	true
RunCommand	empty	arg 1	empty	false
StartEndCommand	arg 1	empty	arg 2	false
FunctionalCommand	arg 1	arg 2	arg 3	arg 4

Command decorators

These are methods that are provided by all `Commands` and allow you to create new commands that modify the underlying command in some way, or implicitly create command groups. These can be used as an alternative way to write command groups, but are also used when creating commands on-the-fly in `RobotContainer`.

- `alongWith`: Runs the base command and the sub-command(s) in parallel ending when they are all finished (c/ `ParallelCommandGroup`)
- `andThen`: Runs the base command and then the sub-command(s) or runnable. See also the class `SequentialCommandGroup`.
- `asProxy`: Blocks inheritance of subsystem requirements. See also the class `ProxyCommand`.

- `beforeStarting`: Runs the sub-commands or runnable and then the base command. See also the class `SequentialCommandGroup`.
- `deadlineWith`: Runs the base command and sub-commands in parallel, ending when the base command is finished. See also the class `ParallelDeadlineGroup`.
- `finallyDo`: Adds an (additional) `end` method to a command.
- `raceWith`: Runs the base command and sub-commands in parallel, ending when any of them are finished. See also the class `ParallelRaceGroup`.
- `repeatedly`: Runs the base command repeatedly. See also the class `RepeatCommand`.
- `unless`: Runs the command only if the supplied `BooleanSupplier` is `false`. See also the class `ConditionalCommand`.
- `until`: Overrides the `isFinished` method with a `BooleanSupplier`.
- `withTimeout`: Adds a timer-based `isFinished` condition.

(I have omitted a few of the more esoteric decorators for brevity.)

Running commands

There are generally three ways to run a command:

- Bind it to a trigger, usually a joystick button
- Run it by default
- Run it in autonomous mode

Triggers

Triggers are objects that run some command when some event takes place, like a button being pressed. The easiest way to create a trigger is by using a `CommandJoystick` or `CommandXboxController`. Trigger objects don't need to be stored.

```
CommandJoystick joystick = new CommandJoystick(0);

joystick.button(1).toggleOnTrue(new MyCommand(...))
```

It is also possible to create triggers from any `BooleanSupplier`:

```
new Trigger(() -> subsystem.getLimitSwitch()).whileTrue(...)
```

Some trigger methods should be passed a command to run:

- `onFalse`: Starts the command when the trigger becomes false, e.g. the button is released. Usually the command will have its own `isFinished` condition. Often used for instant commands.
- `onTrue`: Starts the command when the trigger becomes true, e.g. the button is pressed. Usually the command will have its own `isFinished` condition. Often used for instant commands.
- `toggleOnFalse`: Starts or stops the command when the trigger becomes false. Seldom used. Usually this command will otherwise run indefinitely (`isFinished` returns `false`).
- `toggleOnTrue`: Starts or stops the command when the trigger becomes true. For example, press a button and the intake starts running; it keeps running until the button is pressed a second time. Usually this command will otherwise run indefinitely (`isFinished` returns `false`).
- `whileFalse`: Starts the command when the trigger becomes false, and stops it when the trigger becomes true. Usually this command will otherwise run indefinitely (`isFinished` returns `false`).
- `whileTrue`: Starts the command when the trigger becomes true, and stops it when the trigger becomes false. For example, the robot feeds balls into the shooter while the button is pressed, and stops when it is released. Usually this command will otherwise run indefinitely (`isFinished` returns `false`).

Some trigger methods create new triggers:

- `and`: Combines the trigger with the parameter (often another trigger) to make a trigger that only activates when both triggers are true.
- `debounce`: Creates a new trigger that only activates when the underlying trigger has been true for some period of time. This is useful for physical sensors and buttons that may be jittery.
- `negate`: Creates a new trigger that is only true when the underlying trigger is false.
- `or`: Combines the trigger with the parameter (often another trigger) to make a trigger that only activates when either trigger is true.

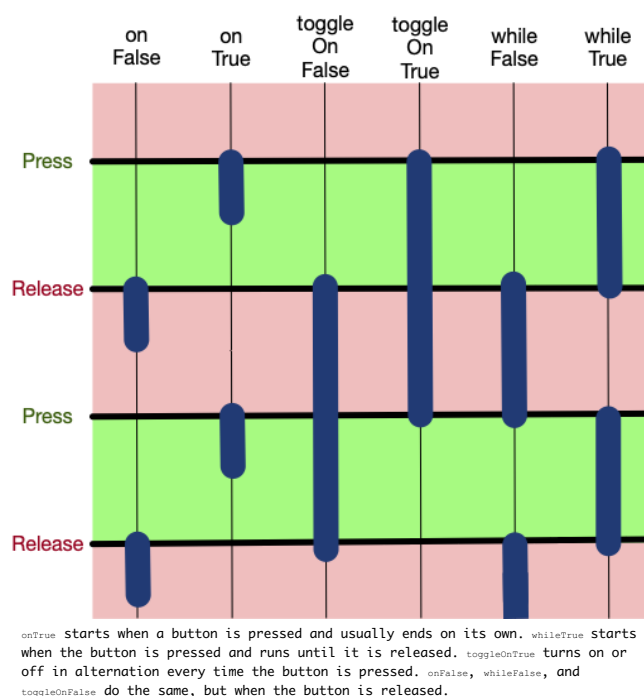
Of these, you will probably use `onTrue` (for instant commands), `whileTrue` (to run while pressed), `toggleOnTrue` (to turn on or off when pressed), and `debounce` (to smooth noisy signals) most often.

Default commands

Sometimes you want a command to run all the time on some subsystem, unless you have something more specific to run. This is the “default command” for that subsystem.

The most commonly encountered example of a default command is the “Arcade Drive” command, which connects a joystick to the drive subsystem. This will run all of the time, except when you engage some autonomous driving routine.

To set the default command for a subsystem, simply call `setDefaultCommand()`. Each subsystem can only have (at most) one default command. When using default commands, it is important that all commands using that subsystem have their requirements set correctly; this ensures that the scheduler will deschedule the default command when they are scheduled.



```
// in RobotContainer.java, in configureBindings()
m_driveSubsystem.setDefaultCommand(
    new ArcadeDriveCommand(m_driveSubsystem,
        () -> m_joystick1.getX(), // double supplier for turn
        () -> m_joystick1.getY()); // double supplier for speed
```

Autonomous commands

TODO

Esoteric commands

These commands are used only in very specific circumstances.

- `NotifierCommand`
- `PIDCommand`, `ProfiledPIDCommand`
- `RamseteCommand`
- `ScheduleCommand`
- `ProxyScheduleCommand`
- `WrapperCommand`
- `MecanumControllerCommand`
- `SwerveControllerCommand`
- `TrapezoidProfileCommand`

See also

- [Binding Commands to Triggers](#)
- [Command Compositions](#)
- [Command interface](#)

This page is part of BoVLB's FRC Tips. Find this page online at <https://bovlb.github.io/frc-tips/commands/commandscheduler.html>



CommandScheduler

When you are first taught to program, you are usually shown what is called the “imperative” style. That means that you are in control of what happens when. In a command-based robot, you have to use an “event-driven” style. You must learn how to break your code up into small pieces that execute quickly and rely on the `CommandScheduler` to call them at the right time.

The `CommandScheduler` will manage commands, calling their four lifecycle methods (`initialize`, `execute`, `isFinished`, `end`). It will also call the `periodic` methods of your subsystems and test any triggers you may have (mostly this will be joystick buttons). It is also responsible for managing the requirements of two commands, so two commands with overlapping requirements are never scheduled at the same time.

There are a number of ways to invoke the `CommandScheduler`:

`CommandScheduler.getInstance().run()`

- This makes the `CommandScheduler` perform its main loop for subsystems, triggers, and commands.
- This should be called from `Robot.robotPeriodic` and nowhere else.
- Most commands will not run while the robot is disabled, so will be automatically cancelled.

Trigger methods

- After you have bound a `Command` to a `Trigger`, the `CommandScheduler` will then test the trigger automatically every iteration. When the trigger activates, it will call `schedule` on the command.
- You’re probably already using `Triggers` in the form of joystick buttons.

`Command.schedule()`

- Attempts to add the command to the list of scheduled commands.
- If a scheduled command has overlapping requirements, then either the other commands will be cancelled or if the other commands are non-interruptible (rare), then the attempt will fail.
- This should be called by `Robot.autonomousInit` to set the autonomous command.
- It’s fairly rare for teams to call `schedule` in any other context. The main example is a pattern where you create a state machine by having each state be a separate command, with all of them sharing the same requirements, but it is usually better to [do the scheduling indirectly via Triggers](#). Outside that, if you find yourself wanting to call this anywhere else, you’re probably doing something wrong.
- Calls to `schedule` from inside a command lifecycle method are deferred until after all the scheduled commands have been run.

`Command.cancel()`

- Unchedules the command
- May cause a default command to be scheduled
- It is common to call `cancel` on the autonomous command inside `Robot.teleopInit`.
- There is also `CommandScheduler.getInstance().cancelAll`
- Calls to `cancel` from inside a command lifecycle method are deferred until after all the scheduled commands have been run, and after all the pending schedules have been scheduled.
- It’s pretty rare to have to call this. If you find yourself wanting to call this anywhere else, you’re probably doing something wrong.

Putting it all together

This is a rough outline of how everything gets run.

`TimedRobot.startCompetition` has an endless loop which polls its time-based priority queue for callbacks that are ready to run. `RunnableS` are added to that priority queue using `TimedRobot.addPeriodic`. By default, the only thing on that queue is `IterativeRobotBase.loopFunc`.

`IterativeRobotBase.loopFunc` does:

1. Calls `<oldMode>Exit` and `<newMode>Init` if the mode is changing
2. Calls `<mode>Periodic` (e.g. `autoPeriodic` and `telopPeriodic`)
3. Calls `robotPeriodic`
4. Updates `SmartDashboard` and `LiveWindow` properties
5. Does loop overrun reporting

By default, the only thing `robotPeriodic` does is to call `CommandScheduler.run`.

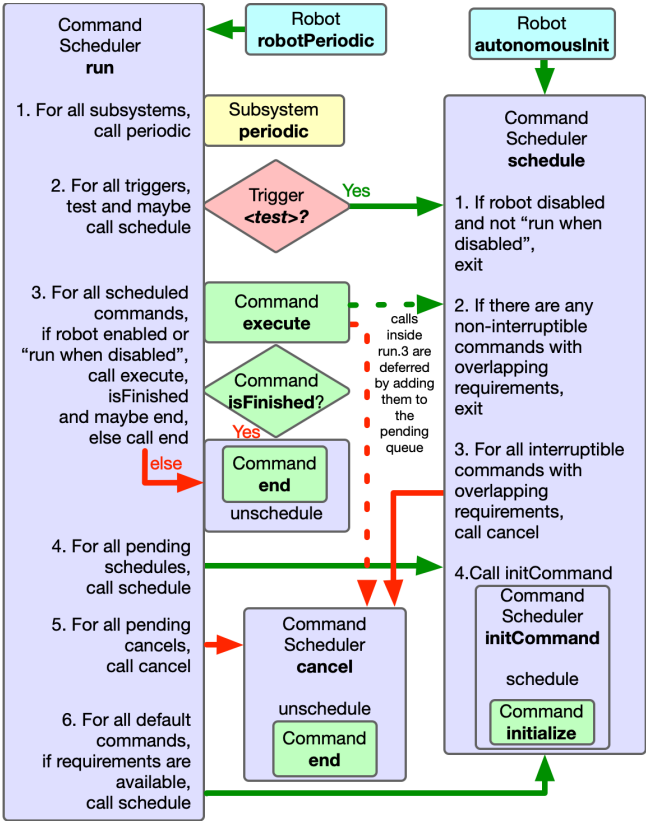
`CommandScheduler.run` does the following:

1. Calls subsystem `periodic` methods
2. Polls its `EventLoop`
3. For all scheduled commands, call `execute`, `isFinished` and/or possibly `end`.
4. Enact pending calls to `schedule` and `cancel`
5. Schedule default commands
6. Does its own loop overrun reporting

When `EventLoop.poll` is called, it runs every registered `Runnable`. `Runnables` are registered using `EventLoop.bind`. By default, the only way `RunnableS` are added to the `CommandScheduler`’s `EventLoop` is by calling one of the binding methods on a `Trigger`.

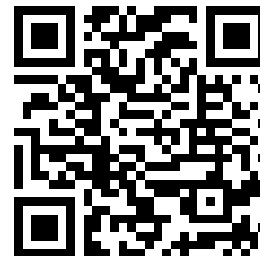
Note: You can call `Robot.addPeriodic` to add your own periodic tasks, possibly with a different period. Don’t rely on being able to use periods shorter than the main loop period of 20ms, because it’s all running in the same thread. Periodic methods registered with `addPeriodic` are not subject to overrun reporting, so you may not notice if they’re causing performance problems. If you are confident about thread-safe programming, you could also use [NotifierS](#).

See also



This shows the workflow of the CommandScheduler in Java. The C++ implementation has almost identical behaviour. This diagram does not show command event methods

Function References, Lambda Functions, and more



Function References

The normal way to write functions in Java is to put them in a class. Functions in a class are called “methods” and must be called either on an instance object or (for `static` methods) on the class itself.

Sometimes you want to be able to use a function as an argument to another function. Such an argument is often referred to as a “callback function”, and it allows other code to call parts of your code later in a very flexible way. An API that accepts a function reference can call your code without having to know anything about how it is written or what your methods are called.

Callback functions typically do one of three things: They supply information; they consume information; or they do something.

For a function to be used as an argument, it has to be a “function reference”. In WPILIB, many methods related to triggers and commands will accept function references. There are two ways to create function references: lambda functions (`->`) and the method reference operator `::`.

Lambda Expressions

The most common way to create such function references is using a lambda expression. This is a special expression that creates a function (reference) on-the-fly. Lambda expressions use the special operator `->`. (Compare with the `lambda` keyword in languages like Python.)

```
// Lambda expression for a function that takes three arguments
// and calculates their sum
(x, y, z) -> x + y + z

// Same thing, but with a statement block
(x, y, z) -> {
    return x + y + z;
}
```

On the left of the `->` operator is the parameter list. This has zero or more parameters in parentheses. The parentheses are optional when there is exactly one parameter. You may specify types of the parameters, but this is almost always optional because it can be deduced from the context.

On the right of the `->` operator is either an expression or a statement block in braces. A simple expression is used as the return value of the lambda expression. A statement block is executed like a normal function body. If there is a `return` statement, then this defines the value returned by the function; otherwise the function has `void` return.

Remember that the lambda expression only defines the function reference. Just like in a normal method, the function body is not evaluated until the function is actually invoked.

Lambda expressions have access to local and instance variables. This means that you can simply use these variables in the function body. This allows your lambda function to be configured in a way that is invisible to the code that calls it. It also allows it to have access to controls and information that the calling code does not. Your lambda function can do anything that you can do, but it is packaged up as something that other code can invoke without any explicit dependency on you.

Method Reference Operator

It is also possible to turn any method into a function reference using the `::` method reference operator.

```

class MySubsystem ... {
    ...
    boolean hasGamePiece() { ... }
}

...

private final MySubsystem subsystem = new Subsystem();

// Get an anonymous function reference with the same arguments
// and return type. Use this as a boolean supplier without
// having to know about the subsystem.
subsystem::hasGamePiece

```

On the left of the `::` operator is a reference to some object. In the example, `subsystem` is a reference to a `MySubsystem` object. It is also common to see the `this` keyword used here, which allows you to reference methods on the current object.

On the right of the `::` is the name of a (public) method defined on the object. In the example, `hasGamePiece` is a method defined in `MySubsystem`. It is also possible to use the `new` keyword here, which allows you to reference the constructor method.

Putting the object reference and the method name together with the `::` operator yields a function reference that has the same arguments and return type, but which can be invoked as a function reference without reference to the instance object.

The function reference in the example above takes no arguments and returns a `boolean`, so matches the pattern required for a `BooleanSupplier`.

Interfaces

If you look up the documentation for WPILIB functions, you'll see that they don't actually take function references explicitly. Instead they accept an instance of some interface like `Supplier`, `Consumer`, `Runnable`, or `Callable`.

These are all "functional interfaces" which means you can just use a function reference instead and Java will automatically convert it for you.

Type	Arguments	Return	Methods	Notes
<code>Supplier</code> <X> <code>Supplier</code>	None	thing supplied, e.g. <code>boolean</code>	<code>get</code> or <code>getAsX</code> , e.g. <code>getAsBoolean</code>	Supplies information, e.g. <code>BooleanSupplier</code>
<code>Runnable</code>	None	None	<code>run</code>	Does something, e.g. runs command
<code>Consumer</code>	thing consumed, eg. <code>Pose2d</code>	None	<code>accept</code>	Accepts inputs of some type
<code>Callable</code>	None	result of some type	<code>call</code>	Also supplies things, but has special uses

Suppliers (especially `BooleanSupplier`s) and runnables are often encountered with WPILIB, whereas consumers and callables are not.

Suppliers

A supplier is a class that supplies values of some specific type when you call the appropriate `getAsX` or `get` method. It can be created from an anonymous function that takes no argument and returns a value of that type. Suppliers that return objects are not required to return a fresh object every time (so would not be used for an anonymous command factory).

```

() -> subsystem.hasGamePiece() // BooleanSupplier
() -> joystick.getX()           // DoubleSupplier
() -> joystick.getX() > 0.0     // BooleanSupplier

```

These suppliers can then be used later to fetch the value:

```

class ArcadeDrive extends Command {
    private DriveSubsystem m_subsystem;
    // Keep suppliers in instance variables
    private DoubleSupplier m_speed;
    private DoubleSupplier m_turn;

    public ArcadeDrive(DriveSubsystem subsystem,
        DoubleSupplier speed, DoubleSupplier turn) {
        m_subsystem = subsystem;
        // Store these suppliers for later use
        m_speed = speed;
        m_turn = turn;

        addRequirements(subsystem);
    }

    @Override
    void execute() {
        // Get values from the suppliers
        double drive = m_drive.getAsDouble();
        double turn = m_turn.getAsDouble();
        m_subsystem.drive(drive+turn, drive-turn);
    }
}

...

// In RobotContainer we connect the suppliers to the joystick axes
m_drive.setDefaultCommand(new ArcadeDrive(m_drive),
    () -> adjustJoystick(-m_joystick.getY()),
    () -> adjustJoystick(-m_joystick.getX()));

```

See also the similar but different example at [Default Commands](#).

Unboxed types (non-objects) like `boolean` and `double` have special supplier types like `BooleanSupplier` and `DoubleSupplier` with methods like `getAsBoolean()` and `getAsDouble()` to get the value. Boxed types (objects) are treated differently: For example, a supplier for `Pose2d` objects would be `Supplier<Pose2d>`, and the accessor is simply `get()`.

Suppliers are a good way to isolate dependencies. In the code above, `speed` and `turn` come from a joystick, but this code doesn't need to know anything about joysticks. This means that you can change to a different type of joystick or even bring in semi-autonomous "driver assist".

In WPILIB, `Triggers` implement the `BooleanSupplier` interface and often accept it.

Runnablees

A `Runnable` is a class that has a `void run()` method. It can be created from an anonymous function that takes no arguments (and any return value is ignored). In WPILIB, it can be used to create commands on-the-fly using `InstantCommand`, `RunCommand`, `StartEndCommand`, or `FunctionalCommand`. It can also be passed to many trigger methods (along with a required subsystem).

```

// This expression can be used as a Runnable
() -> { /* do stuff here */ }

```

Because `Runnablees` have no parameters and no return, they are executed solely for their side-effects.

Consumers

A `Consumer` is similar to a supplier but in reverse. With a `Supplier`, the receiver gets to decide when and how often it is called. With a `Consumer`, the sender makes those decisions. Instead of having a `get` or `getAs<TYPE>` method, a `Consumer` has an `accept` method.

`Consumer`s are not much used in FRC programming, but they might be useful in a case where it's important that each value be processed exactly once. An example of this might be camera frames, or information derived from that such as robot location, or game piece locations.

```
// Define a record type to consume
public record VisionMeasurement(Pose2d pose,
    double timestamp, Matrix<N3,N1> stddevs) {}

// Expose a consumer that processes the records
public final Consumer<VisionMeasurement> visionMeasurementConsumer = (vm) -> {
    m_poseEstimator.addVisionMeasurement(vm.pose(),
        vm.timestamp(), vm.stddevs());
};

// ...

// Somewhere else, pass new records to the consumer
m_visionMeasurementConsumer.accept(
    new VisionMeasurement(pose, timestamp, stddevs));
```

Callables

A `Callable` is like a cross between `Runnable` and `Supplier`. It has a `call()` method that returns a value of some type. `Callable`s may throw exceptions.

`Callable`s are used when expecting a new result each time (like an anonymous factory), when the work involves things that might throw (like file or network input/output), or when passing function references between threads. These are not much used in FRC programming.

See also

- [Functions as Data](#)

This page is part of BoVLB's FRC Tips. Find this page online at <https://bovlb.github.io/frc-tips/commands/best-practices.html>



Best Practices for Command-Based Programming

In May 2024, [Oblarg posted an article on Chief Delphi](#) that laid down some principles for best practices to follow when doing command-based programming. Read the original post for the details, but it can be summarised in three key points:

- Control subsystems using command factories
- Get information from subsystems using triggers
- Co-ordinate between subsystems by binding commands to triggers

The idea is to reduce dependencies between subsystems and gather all cross-subsystem behaviour in one place. This makes your code easier to write, easier to maintain, less likely to have bugs, and more reusable.

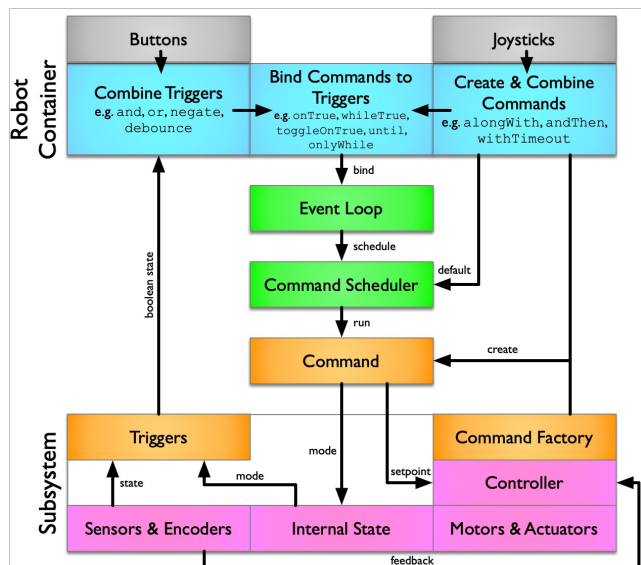
Summary

This diagram summarises the architecture. Starting at the bottom:

- The pink boxes are the private components of the subsystem, not directly accessible from outside.
- The orange boxes represent the public API of the subsystem.
 - State is exposed only through Triggers. These are usually public final member variables.
 - Control is exposed only through command factory methods. These are public instance methods that take zero or more configuration parameters and return a Command. Configuration parameters are commonly Suppliers.
 - The resulting Commands are considered part of the subsystem's API, so they are allowed to access the private components.
- The green boxes represent the scheduling components.
 - When we bind commands to triggers, it is registered with the EventLoop. This tests the triggers and schedules commands when appropriate.
 - Default commands are registered with the CommandScheduler. This runs scheduled commands.
- The blue boxes collectively represent the RobotContainer, where:
 - Triggers are combined (possibly between subsystems);
 - Commands are combined (again, possibly between subsystems); and
 - Commands are bound to triggers.

The RobotContainer is also responsible for setting the default commands of subsystems, again using the subsystem's command factories.

- The grey boxes represent the fact that DriverStation buttons are also available for triggers, and the joysticks can be used (in the form of `DoubleSupplierS`) for commands (typically default).



Add command factories

A “command factory” is simply a method that returns a new instance of a command. You should add these methods to each subsystem to represent all the basic actions someone might need the subsystem to perform. These commands should do small, well-defined tasks like changing a setpoint or setting an internal state. Think of them as basic building blocks to be plugged together. The implementation of these commands is entirely within the subsystem module, so they have access to all of the subsystem internals.

Take care in choosing which commands to implement and what to name them. Remember that the core idea here is to isolate client code from knowing anything about subsystem implementation, including any specific values for position or speed. If a subsystem has a number of positions, modes, or speeds then you will want to create a separate command for each one. Internally, you may want to use `enum` to name and configure the different setpoints. This allows you to use `setName` to give each one an appropriate label for debugging.

Ideally these commands should express their purpose in terms of the problem domain (e.g. `startShooting`, `intakeGamepiece`), not how they are implemented.

```
// Shared internal implementation
private Command setState(ShooterMode mode) {
    return Commands.runOnce(() -> {
        m_mode = mode
    }, this).WithName(mode.toString());
}

// Public command factory
public Command startShooting() {
    return setState(ShooterMode.SHOOTING);
}
```

Tip: If you don't understand why we're using `() -> { ... }` here as a `Runnable`, you might want to read up on [Lambda functions](#).

These commands should share implementation whenever possible. In this way, all the commands that control a setpoint can also run the control loop.

```
// Internal command factory used by all the public command factories
private Command setAngle(DoubleSupplier angle) {
    return Commands.run(() -> {
        m_setpoint = angle.getAsDouble();
        // Use m_setpoint to control the pivot angle here
        m_feedback = m_feedbackController.calculate(m_position, m_setpoint);
        m_feedforward = m_feedforwardController.calculate(m_position,
m_setpoint);


        m_power = MathUtil.clamp(feedforward + feedback, -1.0, 1.0);
        m_motor.setPower(m_power);
    }, this);
}
```

If a command needs configuration or other information from elsewhere, then either the factory or the Subsystem constructor should take a `Supplier`, e.g. a `BooleanSupplier` or a `DoubleSupplier`. Pass it to the constructor if it's a universal piece of information; pass it to the command factory if it's a detail of the specific command being constructed.

This supplier should be providing outside information from the problem space, not implementation specifics. We prefer to pass a `Supplier` rather than a specific value because we want to be able to support dynamic configuration where the value changes. We prefer to pass a `Supplier` rather than injecting a `Subsystem` because we don't want to tie the implementations together; we should assume the minimum possible about where the information comes from.

For example, an aiming system needs to pivot the shooter to a specific angle in order to launch a game piece into a target. The correct angle to use is determined empirically as a function of the distance from the target. In this case, we should not supply a specific pivot angle, but instead a distance to target. The relationship between distance and angle is an internal implementation detail of the aiming system. The distance to the target is an appropriate problem-space concept for communication.

```
public Command setAiming(DoubleSupplier distance) {
    return new setAngle(() -> {
        // distance to speaker in metres
        double distance = distance.getAsDouble();
        // m_angles is an InterpolatingDoubleTreeMap
        double angle = m_angles.get(distance);
        return angle;
    }).withName("Aiming");
}
```

 Tip: In the code above, we're using `() -> { ... return angle; }` to create a `DoubleSupplier` using a [Lambda function](#).

Internally, commands can be created in a number of ways, but the `Commands` class provides useful methods like `run` and `runOnce`. Use `run` for commands that need to keep running continually, either because they don't do exactly the same thing on each iteration, or because they need to block out a default command. (Triggers provide some useful alternatives to default commands.) Use `runOnce` for a command that runs and then immediately ends; this is useful for changing a setpoint or setting a mode when the command doesn't need to enact it continually.

These building-block commands will generally have a single method (`initialize` or `execute`) and will generally not have either an `end` or an `isFinished` implementation. Instead of adding an `isFinished` method, provide a `Trigger` that can be combined using `until`. Instead of adding an `end` method, rely on some other command being run, such as a `stop` command, possibly as a default command or using `andThen`, but quite likely because a different `Trigger` has become active.

Outside of the command factories, a subsystem should provide no other way for another class to change its behaviour. All motors, controllers, and state must be private. It should be impossible to write a command that requires a subsystem other than by using a command factory on that subsystem. The flip side of this is that the commands produced by the command factories should require only the subsystem that created them. Further, they shouldn't even know about any other subsystem. Subsystems constructors and other methods should never accept other subsystems as arguments.

Add triggers


A `Trigger` is essentially a wrapper for a `BooleanSupplier` that allows them to be bound to `Commands`. When the `BooleanSupplier` changes state, the `Command` will automatically be scheduled.

Think about what information other parts of the program need from your subsystem. Try to express that in terms of yes/no questions. Explain their meaning in terms of the problem domain (e.g. there's a game piece in a particular place, the shooter is ready to shoot, the subsystem is in some mode) instead of their implementation (front beam break is broken, wheels are at target speed). Publish them as `Triggers`.

```
// There will be a separate trigger for every possible mode.
public final Trigger isShooting =
    new Trigger(() -> m_mode == ShooterMode.SHOOTING);

// Is the arm at target position and zero speed?
private boolean isReady() {
    return MathUtil.isNear(m_encoder.getPosition(),
        m_setpoint, k_angleTolerance)
        && MathUtil.isNear(m_encoder.getVelocity(),
            0, k_velocityTolerance);
}

// Don't expose position and speed directly
public final Trigger isReady = new Trigger(this::isReady)
    .debounce(k_isReadyDelay, Debouncer.DebounceType.kFalling);
```

 **Tip:** When you see `() -> <boolean expression>`, we're creating a `BooleanSupplier` using a [Lambda function](#). `this::methodName` is using the [method reference operator](#).

Apart from the `Triggers`, a subsystem should not expose any other information. If the subsystem has modes, each mode can be a separate trigger. Don't expose position and speed; instead answer problem-related questions about them.

Bind commands to triggers

Now we move onto using the command factories and triggers. Most of this will take place in `RobotContainer`. This class is responsible for controlling the behaviour of subsystems and, in particular, anything that requires co-ordination between multiple subsystems.

Combining Triggers

Now that you have a good collection of `Triggers`, think about how they should be combined to make useful decisions about robot behaviour. For example, when deciding whether to shoot, you might think about various things:

- Is the driver pressing the "shoot" button? All drive buttons are already `Triggers`.
- Is there a game piece in the right location? This might be determined by beam-break sensors, but the subsystem will package this in a `Trigger`.
- Is the robot within shooting range of the target? This may be based on a distance calculation, which in turn uses the location of the robot (and of the target).
- Are the shooter wheels ready? This may be a `Trigger` that compares the setpoint with the current speed.
- Are we aimed at the target? This may be a `Trigger` that compares the pivot mechanism's angle to the current setpoint.

The most common way of combining triggers is using `and` to make something happen when all of the triggers are true. There is also `or` for when either `Trigger` is true and `negate` to flip the sense of a trigger. It's common to use `debounce` as well, either to stop a `Trigger` from flickering, or to ensure that the `Command` doesn't end too soon.

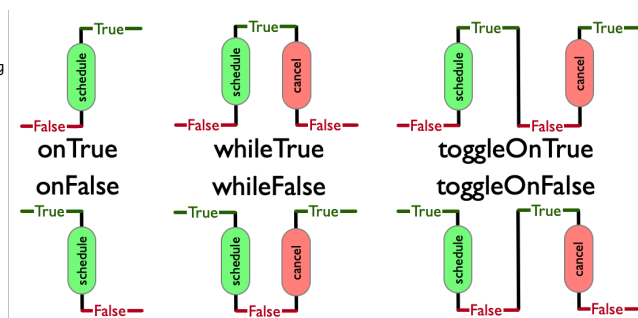
```
// When the driver presses the button, set the shooting mode
shooting_button.onTrue(m_shooter.startShooting());

// Combine the shooter mode and other triggers to shoot the game piece
m_shooter.isShooting
    .and(m_shooterSensors.hasGamePiece)
    .and(isInRange)
    .and(m_shooterFlywheel.isReady)
    .and(m_aimer.isReady)
    .whileTrue(m_feeder.feedGamePiece())

// Stop shooting when we don't have a gamepiece or we go out of range
m_shooter.isShooting
    .and(m_shooter.hasGamepiece.negate())
    .or(isInShootingRange.negate())
    .onTrue(m_shooter.stopShooting());
```

Binding Triggers to Commands

To bind a command to a trigger, simply use a method like `onTrue`, `whileTrue` or `toggleOnTrue`. `onTrue` is good for instant commands that do something and immediately stop. `whileTrue` is good for commands that should keep executing so long as the condition is true. `toggleOnTrue` is usually only used with driver buttons, so they can enable and disable some mode.



Combining commands

When using triggers, you'll find there's much less need to combine commands to get complex behaviours. If you do need to combine commands, you can easily do so with command decorators methods like `alongWith` and `andThen`.

```
// When we're intaking, do the three things
m_shooter.isIntaking
    .whileTrue(m_frontManipulator.setIntaking()
        .alongWith(m_backManipulator.setIntaking()
            .alongWith(m_pivot.setIntaking()))
        .withName("Intaking"));
```

Default Commands

Default commands should be fairly simple commands like stop or hold position. Such actions should already be available using command factories. Resist the temptation to put decision logic in the default command.

```

m_frontManipulator.setDefaultCommand(m_frontManipulator.stop());
m_backManipulator.setDefaultCommand(m_backManipulator.stop());
m_pivot.setDefaultCommand(m_pivot.setHome());

// Utility method to adjust joystick values
private DoubleSupplier adjustJoystick(DoubleSupplier input, boolean negate) {
    return () -> {
        double x = input.getAsDouble();
        if(negate) {
            x = -x;
        }
        x = MathUtil.applyDeadband(x, k_joystickDeadband);
        x = Math.pow(Math.abs(x), k_joystickExponent) * Math.signum(x)
        return x;
    };
}

m_drive.setDefaultCommand(
    // drive() is a command factory that takes three double suppliers
    m_drive.drive(
        adjustJoystick(m_gamepad::getLeftY, true),    // xSpeed
        adjustJoystick(m_gamepad::getLeftX, true),    // ySpeed
        adjustJoystick(m_gamepad::getRightX, true)    // rot
    ));

```

Avoid having complex default commands that make elaborate decisions. Instead put that complexity into triggers and bind the appropriate commands.

Autonomous routines

Old-school autonomous routines will be able to combine the output of command factories without much change in the way they are written. For using PathPlanner, you can use `NamedCommands.registerCommand`.

Both techniques commonly need additional information to know when commands should end. The best way to do this is attach a trigger using the `until` decorator (also `unless`, `onlyWhile`).

```

NamedCommands.registerCommand("Shoot",
    m_shooter.startShooting()
        .until(m_shooter.isIdle)
        .withName("Shoot"));

```

Exceptions

These best practices are general principles to improve your code. There are some case where it's appropriate to deviate from them. For example, what if multiple subsystems need co-ordination that can't be achieved with booleans?

If this is the case, we should still try to cast that communication in the language of the problem-space, and not in the language of subsystem implementation. Usually this involves exposing a Supplier, although occasionally a Consumer is more appropriate. This decouples subsystems from each other and makes the code base more flexible and reusable.

Pose Estimation

A good example of an exception is pose estimation. Commonly we will have some subclass of `PoseEstimator` and we have multiple places we want to access it:

- The drive subsystem updates it with wheel odometry
- One or more vision systems update it with vision measurements
- Path following needs to know the current location
- Shooters need to know the distance to target

This doesn't really fit in with the best practices outlined above. My current recommendation is to put the `PoseEstimator` inside the drive subsystem, and then expose two public fields:

- A `pose` supplier that yields a `Pode2d`. This can be injected where required.

- A `visionMeasurement` consumer that accepts a `record`. This can be injected into the vision systems. See [Consumers](#).

Performance

A potential issue with pervasive use of triggers is that you may end up asking the same question multiple times per iteration. There are two problems with this:

- The first is that, if answering the question requires hardware access or expensive calculation, then you can end up paying that cost repeatedly.
- The second is that, if you're using the same underlying trigger for multiple bindings, then they could get different answers within one iteration and therefore schedule inconsistent commands.

The solution to this is to make sure that selected triggers are using a cached value that is calculated once per iteration, perhaps in a periodic method.

Incremental Adoption

As presented here, the best practices are hard-and-fast rules that come as a package. It's certainly possible to implement them incrementally within an existing project, or take on some of these without others. They still provide benefit, even if you don't implement all of them everywhere. Each of three key points can be pursued separately, maybe even one subsystem at a time:

- You might start by introducing command factories and try to wean yourself off controlling subsystems directly.
- You could then start adding triggers (and other suppliers) and use them to decouple subsystems from each other.
- Strictly speaking, neither command factories nor subsystem triggers are required for `RobotContainer` to bind commands to triggers; you're already doing that with driver buttons, but you need to move the decision complexity out of commands by breaking them into simpler commands and triggers.

Subsystem Periodic Methods

Separately from the discussion of these best practices, there has been some discussion about moving away from using `Subsystem.periodic` methods. There are a number of reasons for this:

- If you follow these best practices, your decision-making and will happen using triggers, and your control loops will end up in commands, so you'll naturally end up with less for `periodic` to do.
- Having a single `periodic` method per subsystem encourages programmers to throw everything together into a big mess. If you instead call `addPeriodic` to add periodic tasks when required, then it forces you to think about what the separate tasks are and how often they need to run. See [Putting it all together](#) for more advice about calling `addPeriodic`.
- The `periodic` method runs at the start of the iteration. This is a good time to do pre-command actions like caching per-iteration values and updating odometry. Unfortunately, this is a bad time to do post-command actions like servicing control loops. This could instead be done in commands, if you can guarantee that there is always one command running.
- Dashboard updates can be done more elegantly through `initSendable`.

In summary, the `Subsystem.periodic` method is probably a good place to update input caches, maybe to update odometry, and possibly for logging, but other uses should be fading away.

References

Examples of following these best practices can be seen in:

- [Command-based best practices for 2025: community feedback](#) – The Chief Delphi thread that started it all.
- [Command Based Best Practices example](#) – A Chief Delphi thread where people try to work through some examples of following best practices.
- [Some ideas for our Crescendo shooter](#) – Many of the examples in this article are based on this code.

TODO: more

Acknowledgements

Thanks to [@Oblara](#) for the original insights and many others for feedback and examples.

Some useful FRC links

FIRST

- [Season Materials](#)
- [Crescendo Game Manual](#)
- [Discussion Fora](#)
- [YouTube playlists](#)
- [Portal for Incident Reporting](#)

WPILIB

- [FIRST Robotics Competition Control System](#)
 - [WPILib Installation Guide](#)
 - [Status Lights](#)
- [Java docs](#)
- [C++ docs](#)
- [Python docs](#)
- [Github](#)

CTRE (Cross The Road Electronics)

For Victor, Talon (including Falcon 500), Pigeon IMU, CAN Coder

- [Java docs](#)
- [C++ docs](#)
- Vendor Library JSON: [2023 Phoenix5](#), [2024 Phoenix 5 beta](#), [2024 Phoenix 6 beta](#)
- [Software & Downloads](#): Including Phoenix Tuner and Firmware download

REV Robotics

For Spark MAX

- [Java docs](#)
- [C++ docs](#)
- [REV Hardware Client](#)
- [REV Software Resources](#)
- Vendor Library JSON: [2023 REVLib](#), [2024 REVLib](#)

Other

- [Chief Delphi](#)
- [Reddit FRC](#)

Selected teams

- 2102: Team Paradox: [TBA](#), [website](#), [Github](#), [programming tutorials](#), [PID demo](#)

TODO: Add more ...

