Navegação por Encoder

Na seção anterior, você aprendeu sobre como usar o Elapsed Time para permitir que seu robô navegue autonomamente pelo ambiente ao seu redor. Ao começar, muitas das ações do robô podem ser realizadas ligando um motor por um tempo específico. Eventualmente, essas ações baseadas em tempo podem não ser precisas ou repetíveis o suficiente. Fatores ambientais, como o estado da carga da bateria durante a operação e o desgaste dos mecanismos pelo uso, podem afetar todas as ações baseadas em tempo. Felizmente, há uma maneira de fornecer feedback ao robô sobre como ele está operando, usando sensores; dispositivos usados para coletar informações sobre o robô e o ambiente ao seu redor.

Com o Elapsed Time, para mover o robô para uma distância específica, você teve que estimar a quantidade de tempo e a porcentagem de ciclo de trabalho necessária para ir do ponto a ao ponto b. No entanto, os motores REV vêm com encoders embutidos, que fornecem feedback na forma de ticks (ou contagens) por revolução do motor. As informações fornecidas pelos encoders podem ser usadas para mover o motor para uma posição-alvo ou uma distância-alvo.

Mover os motores para uma posição específica, usando os encoders, elimina possíveis imprecisões ou inconsistências do uso do Elapsed Time. O foco desta seção é mover o robô para uma posição-alvo usando encoders.

> Existem dois artigos que abordam os conceitos básicos dos encoders. "Using Encoders" explora os fundamentos dos diferentes tipos de modos de motor, bem como alguns exemplos de aplicação desses modos no código. Nesta seção, concentraremos no uso de RUN_TO_POSITION.

O outro artigo, "Encoders", concentra-se na funcionalidade geral de um encoder.

Recomenda-se que você revise ambos os artigos antes de prosseguir com este guia. Links abaixo.

Utilizando encoders

Encoders

Fundamentos da programação com Encoders

Comece criando um Op Mode simples chamado **HelloRobot_EncoderAuton**

> Ao criar um Op Mode, uma decisão precisa ser tomada quanto a defini-lo ou não como modo autônomo. Para aplicações com duração inferior a 30 segundos, geralmente exigidas para a jogabilidade competitiva, recomenda-se alterar o tipo de Op Mode para autônomo. Para aplicações com duração superior a 30 segundos, definir o código como o tipo de Op Mode autônomo limitará seu código autônomo a 30 segundos de tempo de execução. Se você planeja exceder os 30 segundos incorporados ao SDK, é recomendável manter o código como um tipo de Op Mode teleoperado. Para obter informações sobre como os Op Modes funcionam, visite a seção de Introdução à Programação. Para obter mais informações sobre como alterar o tipo de Op Mode, confira a seção Banco de Testes - OnBot Java.

> A estrutura do Op Mode abaixo é simplificada e inclui apenas os componentes necessários para criar o código baseado em encoders.

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;

@Autonomous //sets the op mode as an autonomous op mode

public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;

    @Override
    public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
```

```
        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()){

        }
    }
}
```

Assim como em toda navegação relacionada ao drivetrain, a direção de um dos motores precisa ser invertida para que ambos os motores se movam na mesma direção. Como a Classe Bot V2 ainda está sendo usada, adicione a linha rightmotor.setDirection(DcMotor.Direction.REVERSE); ao código abaixo da linha de código rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        waitForStart();
```

> Para obter mais informações sobre a direcionalidade dos motores, confira a página inicial desse capítulo.

Lembrando do uso de encoders que usar o modo RUN_TO_POSITION requer um processo de três etapas. O primeiro passo é definir a posição alvo. Para definir a posição alvo, adicione as linhas leftmotor.setTargetPosition(1000); e rightmotor.setTargetPosition(1000); ao Op Mode após o comando waitForStart();. Para obter uma posição alvo que corresponda a uma distância alvo, são necessários alguns cálculos, que serão abordados posteriormente. Por enquanto, defina a posição alvo como 1000 ticks.

```
waitForStart();

leftmotor.setTargetPosition(1000);
rightmotor.setTargetPosition(1000);

while (opModeIsActive()){
```

```
    }
```

O próximo passo é definir ambos os motores para o modo RUN_TO_POSITION. Adicione as linhas leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION); e rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION); ao seu código, abaixo das linhas de código setTargetPosition.

```
waitForStart();

leftmotor.setTargetPosition(1000);
rightmotor.setTargetPosition(1000);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

while (opModeIsActive()){

    }
```

O foco principal do processo de três etapas é definir uma meta, dizer ao robô para se mover para essa meta e a que velocidade (ou velocidade) o robô deve atingir essa meta. Normalmente, o próximo passo recomendado é calcular a velocidade e definir uma velocidade alvo com base nos ticks. No entanto, isso requer bastante matemática para encontrar a velocidade apropriada. Para fins de teste, é mais importante garantir que a parte principal do código esteja funcionando antes de se aprofundar demais na criação do código. Uma vez que a função setPower foi abordada em seções anteriores e comunicará ao sistema qual velocidade relativa (ou, neste caso, ciclo de trabalho) é necessária para atingir a meta, ela pode ser usada no lugar de setVelocity por enquanto.

Adicione as linhas para definir a potência de ambos os motores para 80% do ciclo de trabalho.

```
waitForStart();

leftmotor.setTargetPosition(1000);
rightmotor.setTargetPosition(1000);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setPower(0.8);
rightmotor.setPower(0.8);
```

```
while (opModeIsActive()){

    }
```

Agora que todas as três etapas do RUN_TO_POSITION foram adicionadas ao código, o código pode ser testado. No entanto, se você deseja aguardar o motor atingir sua posição alvo antes de continuar no programa, pode usar um loop while que verifica se o motor está ocupado (ainda não atingiu seu destino). Para este programa, vamos editar o while (opModeIsActive()) {}.

> Lembre-se de que, dentro de um Op Mode linear, um loop while deve sempre ter o booleano opModeIsActive() como condição. Essa condição garante que o loop while será encerrado quando o botão de parada for pressionado.

Edite o loop while para incluir as funções leftmotor.isBusy() e rightmotor.isBusy(). Isso verificará se o motor esquerdo e o motor direito estão ocupados se movendo para uma posição alvo. O loop while será interrompido quando qualquer um dos motores atingir a posição alvo.

```
while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {


}
```

> Atualmente, o loop while está esperando que qualquer um dos motores atinja a posição alvo. Pode haver ocasiões em que você deseja aguardar que ambos os motores atinjam sua posição alvo. Nesse caso, o seguinte loop pode ser usado:
>
> ```
> while (opModeIsActive() && (leftmotor.isBusy() || rightmotor.isBusy()))
> ```

> Grave e execute o modo de operação duas vezes seguidas. O robô se move conforme esperado na segunda vez? Tente desligar e depois ligar o Control Hub. Como o robô se move?

Na seção Conceitos Básicos de Encoder, é esclarecido que todas as portas de encoder começam em 0 ticks quando o Control Hub é ligado. Como você não desligou o Control Hub entre as execuções, na segunda vez que você executou o modo de operação, os motores já estavam, ou próximos, à posição alvo. Ao executar um código, é desejável garantir que certas variáveis comecem em um estado conhecido. Para os ticks do encoder, isso pode ser alcançado configurando o modo como STOP_AND_RESET_ENCODER. Adicione este bloco ao modo de operação na seção de inicialização. Cada vez que o modo de operação é inicializado, os ticks do encoder serão resetados para zero.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");


        rightmotor.setDirection(DcMotor.Direction.REVERSE);


        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);


        waitForStart();
```

> Para obter mais informações sobre o modo de motor STOP_AND_RESET_ENCODERS, consulte a seção STOP_AND_RESET_ENCODERS no guia de Uso de Encoders.

Converter Ticks do encoder para uma distância

Na seção anterior, a estrutura básica necessária para usar RUN_TO_POSITION foi criada. A inserção de leftmotor.setTargetPosition(1000); e rightmotor.setTargetPosition(1000); no código define a posição alvo como 1000 ticks. Qual é a distância do ponto inicial do robô até o ponto para o qual o robô se move após a execução deste código?

Em vez de tentar medir ou estimar a distância que o robô percorre, os ticks do encoder podem ser convertidos de quantidade de ticks por revolução do encoder para quantos ticks do encoder são necessários para mover o robô por uma unidade de distância, como milímetros ou polegadas. Saber a quantidade de ticks por unidade de medida permite definir uma distância específica. Por exemplo, se você passar pelo processo de conversão e descobrir que um conjunto de rodas leva 700 ticks para mover uma polegada, isso pode ser usado para calcular o número total de ticks necessários para mover o robô por 24 polegadas.

> Lembre-se de que o guia tem como base a Class Bot V2. O REV DUO Build System utiliza o sistema métrico. Como parte do processo de conversão faz referência ao diâmetro das rodas, esta seção fará a conversão para ticks por milímetro.

Para o processo de conversão as seguintes informações são necessárias:

- Ticks que ocorrem em uma revolução completa do encoder.
- Redução total de engrenagens no motor
  - Incluindo todas as reduções proporcionadas por caixas de engrenagens e componentes de transmissão de movimento, como engrenagens, correntes e coroas,

ou correias e polias.
- Circunferência das rodas

Ticks por revolução

A quantidade de ticks por revolução do eixo do encoder depende do motor e do encoder. Os fabricantes de motores com encoders embutidos fornecerão informações sobre a quantidade de ticks por revolução. Para os Motores HD Hex, o encoder conta com 28 ticks por revolução do eixo do motor.

> Visite o site do fabricante do seu motor ou encoder para obter mais informações sobre a contagem de ticks do encoder. Para os Motores HD Hex ou Motores Core Hex, consulte a documentação do motor em nosso site.

Redução total

Como os ticks por revolução do eixo do encoder ocorrem antes de qualquer redução de engrenagem, é necessário calcular a redução total da engrenagem. Isso inclui a caixa de engrenagens e qualquer redução adicional proveniente dos componentes de transmissão de movimento. Para encontrar a redução total da engrenagem, use a fórmula de Engrenagem Composta.

Para o Class Bot V2, existem duas Carcaças UltraPlanetary, 4:1 e 5:1, e uma redução adicional de engrenagem da Saída UltraPlanetary para as rodas, na proporção de 72 dentes para 45 dentes.

> As UltraPlanetary usam a relação de engrenagem nominal como descritor. As relações de engrenagem reais podem ser encontradas no Manual do Usuário do UltraPlanetary.

Utilizando a fórmula de engrenagens compostas para o robô V2 a redução total é:

3.61/1 * 5.23/1 * 72/45 = 30.21

> Ao contrário das engrenagens helicoidais usadas para transferir movimento para as rodas, as Carcaças de Caixa de Engrenagens UltraPlanetary são sistemas de engrenagens planetárias. Para facilitar os cálculos, as relações de engrenagem para as Carcaças já estão reduzidas.

Circunferência da roda

O Class Bot V2 utiliza as Rodas de Tração de 90mm. 90mm é o diâmetro da roda. Para obter a circunferência apropriada, utilize a seguinte fórmula:

cirncunferência = diãmetro * π

Você pode calcular isso com papel e caneta, mas para os propósitos desse guia, isso pode ser calculado pelo código.

> Devido ao desgaste e tolerâncias de fabricação, o diâmetro de algumas rodas pode ser nominalmente diferente. Para obter resultados mais precisos, considere medir sua roda para confirmar que o diâmetro é preciso.

Para recapitular, para o Classe Robô V2 as seguintes informações são verdadeiras:

Descrição	Valor
Ticks por revolução	28 ticks
Redução total	30.21
Circunferência da roda	90mm * π

Cada uma dessas informações será usada para encontrar o número de ticks do encoder (ou contagens) por milímetro que a roda se move. Em vez de se preocupar em calcular essas informações manualmente, esses valores podem ser adicionados ao código como variáveis constantes. Para fazer isso, crie três variáveis:

- COUNTS_PER_MOTOR_REV
- DRIVE_GEAR_REDUCTION
- WHEEL_CIRCUNFERENCE_MM

> A convenção de nomenclatura comum para variáveis constantes é conhecida como CONSTANT_CASE, onde o nome da variável está todo em maiúsculas e as palavras são separadas por um sublinhado.

Adicione as variáveis à classe OpMode, onde as variáveis de hardware são definidas. Definir as variáveis dentro dos limites da classe, mas fora do modo operacional (op mode), permite que elas sejam referenciadas em outros métodos ou funções dentro da classe. Para garantir que as variáveis sejam referenciáveis, elas são definidas como variáveis static final double. O modificador static permite referências às variáveis em qualquer lugar dentro da classe, e o modificador final indica que essas variáveis são constantes e não são alteradas em outros lugares no código. Como essas variáveis não são do tipo inteiro, são classificadas como variáveis double.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
```

```
    private DcMotor rightmotor;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.21;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * Math.PI;
```

Agora que essas três variáveis foram definidas, elas podem ser usadas para calcular outras duas variáveis: a quantidade de contagens do encoder por rotação da roda e o número de contagens por milímetro que a roda se move.

Para calcular as contagens por revolução da roda, multiplique COUNTS_PER_MOTOR_REV por DRIVE_GEAR_REDUCTION. Use a seguinte fórmula:

$$y = a * b$$

Onde:

- **a = COUNTS_PER_MOTOR_REV**
- **b = DRIVE_GEAR_REDUCTION**
- **y = COUNTS_PER_WHEEL_REV**

Crie a variável **COUNTS_PER_MOTOR_REV** pelo código. Ela também pode ser *static final double*.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION;
```

Uma vez que COUNTS_PER_WHEEL_REV é calculado, use-o para calcular os pulsos por milímetro que a roda se move. Para fazer isso, divida COUNTS_PER_WHEEL_REV pela CIRCUNFERÊNCIA_DA_RODA_MM. Utilize a seguinte fórmula.

$$x = (a * b)/c = y/c$$

Onde:

- **a = COUNTS_PER_MOTOR_REV**
- **b = DRIVE_GEAR_REDUCTION**

- **c = WHEEL_CIRCUMFERENCE_MM**
- **y = COUNTS_PER_WHEEL_REV**
- **x = COUNTS_PER_MM**

Crie a variável **COUNTS_PER_MM** pelo código. Ela também pode ser *static final double*.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION;
    static final double     COUNTS_PER_MM           = COUNTS_PER_WHEEL_REV / WHEEL_CIRCUMFERENCE_MM;
```

> COUNTS_PER_WHEEL_REV será criado como uma variável separada de COUNTS_PER_MM, pois é usado no cálculo de uma velocidade alvo.

Movendo para um distância alvo

Agora que você criou as variáveis constantes necessárias para calcular a quantidade de pulsos por milímetro movido, você pode usá-las para definir uma distância alvo. Por exemplo, se você quiser que o robô se mova para frente dois pés, converter de pés para milímetros e multiplicar pelo COUNTS_PER_MM fornecerá a quantidade de pulsos necessários para alcançar essa distância.

Crie mais duas variáveis chamadas leftTarget e rightTarget. Essas variáveis podem ser alteradas e editadas em seu código para indicar às rodas motrizes as posições para as quais devem ir. Em vez de colocá-las com as variáveis constantes, crie essas variáveis dentro do modo de operação (op mode) mas acima do comando waitForStart();.

> A função setTargetPosition(); aceita um tipo de dado inteiro (int) como parâmetro, em vez de um tipo duplo (double). Já que tanto leftTarget quanto rightTarget serão usados para definir a posição alvo, crie ambas as variáveis como variáveis int.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
```

```
rightmotor.setDirection(DcMotor.Direction.REVERSE);

leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

int leftTarget;
int rightTarget;

waitForStart();
```

Atualmente, o principal fator de distância é COUNTS_PER_MM; no entanto, você pode querer percorrer uma distância que está no sistema imperial, como 2 pés (ou 24 polegadas). A distância alvo, nesse caso, precisará ser convertida para milímetros. Para converter de pés para milímetros, utilize a seguinte fórmula:

d(mm) = d(ft) * 304.8

Se você converter 2 pés para milímetros, o resultado será 609,6 milímetros. Para fins deste guia, arredondemos isso para 610 milímetros. Multiplique 610 milímetros pela variável COUNTS_PER_MM para obter o número de pulsos necessários para mover o robô 2 pés. Como a intenção é fazer com que o robô se mova em linha reta, defina tanto leftTarget quanto rightTarget para serem iguais a 610 * COUNTS_PER_MM.

> Como mencionado anteriormente, a função setTargetPosition(); exige que seu parâmetro seja do tipo de dado inteiro. As variáveis leftTarget e rightTarget foram definidas como inteiros, no entanto, a variável COUNTS_PER_MM é do tipo double. Uma vez que esses são dois tipos de dados diferentes, é necessário fazer uma conversão de tipos de dados. Neste caso, COUNTS_PER_MM precisa ser convertido para um inteiro. Isso é tão simples quanto adicionar a linha (int) na frente da variável double. No entanto, é preciso ter cautela com possíveis erros de arredondamento. Como COUNTS_PER_MM faz parte de uma equação, é recomendável converter para um inteiro após encontrar o resultado da equação. O exemplo de como fazer isso é mostrado abaixo.

```
int leftTarget = (int)(610 * COUNTS_PER_MM);
int rightTarget = (int)(610 * COUNTS_PER_MM);
```

Edite as linhas setTargetPosition(); para que ambos os motores sejam configurados para a posição alvo apropriada. Para fazer isso, adicione as variáveis leftTarget e rightTarget aos seus motores respectivos.

```
leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);
```

> Tente executar o código e observar o comportamento do robô. Considere algumas das seguintes perguntas:
>
> - O robô está se movendo para frente por dois pés?
> - O robô parece estar se movendo em linha reta?
> - O código está sendo executado sem erros?

Definindo a velocidade

A velocidade é um controle em malha fechada dentro do SDK que utiliza as contagens do encoder para determinar a potência/aproximada velocidade necessária para que os motores alcancem a velocidade definida. Ao trabalhar com configurações de encoder, é recomendável definir uma velocidade em vez de um nível de potência, pois oferece um maior controle.

Para definir uma velocidade, é importante entender a velocidade máxima em RPM que o seu motor é capaz de atingir. No caso do Bot Classe V2, os motores são capazes de atingir no máximo 300 RPM. Com um conjunto de rodas motrizes, é provável que você obtenha um melhor controle configurando a velocidade para um valor inferior ao máximo. Neste caso, vamos definir a velocidade para 175 RPM.

> Lembre-se que a função setVelocity é medida em ticks por segundo

Crie uma nova variável *TPS*. Adicione TPS ao Op Mode abaixo de onde rightTarget está definido.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");


        rightmotor.setDirection(DcMotor.Direction.REVERSE);


        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);


        int leftTarget = (int)(610 * COUNTS_PER_MM);
        int rightTarget = (int)(610 * COUNTS_PER_MM);
        double TPS;


        waitForStart();
```

Como RPM representa a quantidade de rotações por minuto, é necessário fazer uma conversão de RPM para ticks por segundo. Para fazer isso, divida o RPM por 60 para obter a quantidade de rotações por segundo. Em seguida, multiplique as rotações por segundo por COUNTS_PER_WHEEL_REV para obter a quantidade de ticks por segundo.

TPS = 175/69 * CPW R

```
double TPS = (175/60) * COUNTS_PER_WHEEL_REV
```

Substítua as funções **setPower();** para **setVelocity();**. Adicione TPS como parâmetro **setVelocity();**

```
waitForStart();

leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(TPS);
rightmotor.setVelocity(TPS);

while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())){

    }
```

> Tente compilar o código. Você obteve erros?

Com o estado atual do código você vai conseguir os seguintes erros:

```
org/firstinspires/ftc/teamcode/HelloWorld_EncoderAuton.java line 55, column 18: ERROR: cannot find symbol
  symbol:   method setVelocity(double)
  location: variable leftmotor of type com.qualcomm.robotcore.hardware.DcMotor
org/firstinspires/ftc/teamcode/HelloWorld_EncoderAuton.java line 56, column 19: ERROR: cannot find symbol
  symbol:   method setVelocity(double)
  location: variable rightmotor of type com.qualcomm.robotcore.hardware.DcMotor
```

Isso ocorre porque a função setVelocity(); é uma função da interface DcMotorEx. A interface DcMotorEx é uma extensão da interface DcMotor, que fornece funcionalidades avançadas do motor, como acesso a funções de controle em malha fechada. Para utilizar setVelocity();, as variáveis do motor precisam ser alteradas para DcMotorEx. Para fazer isso, tanto a criação das

variáveis privadas dos motores quanto o mapeamento de hardware precisam ser alterados para DcMotorEx.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotorEx leftmotor;
    private DcMotorEx rightmotor;
```

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotorEx.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotorEx.class, "rightmotor");
```

> Como DcMotorEx é uma extensão de DcMotor, as funções específicas de DcMotor podem ser utilizadas por variáveis definidas como DcMotorEx.

Depois de fazer essas alterações, o código básico para mover dois pés está concluído! O código abaixo é a versão finalizado do código. Nesta versão, outros componentes de hardware e a telemetria foram adicionados.

```
@Autonomous

public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotorEx leftmotor;
    private DcMotorEx rightmotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION;
    static final double     COUNTS_PER_MM           = COUNTS_PER_WHEEL_REV / WHEEL_CIRCUMFERENCE_MM;

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotorEx.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotorEx.class, "rightmotor");
```

```
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);



        int leftTarget = (int)(610 * COUNTS_PER_MM);
        int rightTarget = (int)(610 * COUNTS_PER_MM);
        double TPS = (175/ 60) * COUNTS_PER_WHEEL_REV;

        waitForStart();



        leftmotor.setTargetPosition(leftTarget);
        rightmotor.setTargetPosition(rightTarget);

        leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
        rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

        leftmotor.setVelocity(TPS);
        rightmotor.setVelocity(TPS);

        while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {
            telemetry.addData("left", leftmotor.getCurrentPosition());
            telemetry.addData("right", rightmotor.getCurrentPosition());
            telemetry.update();
        }
    }
}
```

Melhorando a transmissão utilizando RUN_TO_POSITION

Na seção de Navegação do Robô - OnBot Java, foi discutido o mecanismo de setPower();. setPower(); dita em que direção e velocidade um motor se move. Em um drivetrain, isso determina se o robô se move para frente, para trás ou vira.

No modo RUN_TO_POSITION, os contadores do codificador (ou setTargetPosition();) são usados em vez de setPower(); para determinar a direção do motor. Se o valor da posição alvo for maior que a posição atual do codificador, o motor se move para frente. Se o valor da posição alvo for menor que a posição atual do codificador, o motor se move para trás.

Como a velocidade e a direção impactam como um robô vira, setTargetPosition(); e setVelocity(); precisam ser editados para fazer com que o robô vire. Considere o código a seguir:

```
int leftTarget = (int)(610 * COUNTS_PER_MM);
int rightTarget = (int)(-610 * COUNTS_PER_MM);
double TPS = (100/ 60) * COUNTS_PER_WHEEL_REV;



waitForStart();



leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(TPS);
rightmotor.setVelocity(TPS);
```

O rightTarget foi alterado para ser uma posição alvo negativa. Supondo que o codificador comece em zero devido a STOP_AND_RESET_ENCODER, isso faz com que o robô vire para a direita. A velocidade permanece a mesma para ambos os motores. Se você tentar executar este código, pode notar que o robô gira em torno de seu centro de rotação. Para obter uma curva mais ampla, altere a velocidade de modo que o motor direito esteja funcionando a uma velocidade menor do que o motor esquerdo. Ajuste a velocidade e a posição alvo conforme necessário para obter a curva desejada.

> Para obter mais informações sobre como a direção e a velocidade impactam o movimento de um robô, consulte a explicação de setPower(); na seção de Navegação do Robô.

O código a seguir mostra como adicionar uma curva ao programa após o robô se mover para frente por 2 pés. Após o robô atingir a meta de 2 pés, há uma chamada para STOP_AND_RESET_ENCODERS, o que reduzirá a necessidade de calcular qual posição alcançar

após atingir uma posição.

```
@Autonomous

public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotorEx leftmotor;
    private DcMotorEx rightmotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION;
    static final double     COUNTS_PER_MM           = COUNTS_PER_WHEEL_REV / WHEEL_CIRCUMFERENCE_MM;

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotorEx.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotorEx.class, "rightmotor");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

        // TPS variable split to change velocity for each motor when necessary

        int leftTarget = (int)(610 * COUNTS_PER_MM);
        int rightTarget = (int)(610 * COUNTS_PER_MM);
        double LTPS = (175/ 60) * COUNTS_PER_WHEEL_REV;
        double RTPS = (175/ 60) * COUNTS_PER_WHEEL_REV;

        waitForStart();
```

```
leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(LTPS);
rightmotor.setVelocity(RTPS);

//wait for motor to reach position before moving on
while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {
    telemetry.addData("left", leftmotor.getCurrentPosition());
    telemetry.addData("right", rightmotor.getCurrentPosition());
    telemetry.update();
}
// Reset encoders to zero
leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

// changing variables to match new needs

leftTarget = (int)(300 * COUNTS_PER_MM);
rightTarget = (int)( -300 * COUNTS_PER_MM);
LTPS = (100/ 60) * COUNTS_PER_WHEEL_REV;
RTPS = (70/ 60) * COUNTS_PER_WHEEL_REV;

leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(LTPS);
rightmotor.setVelocity(RTPS);

//wait for motor to reach position before moving on
while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {
    telemetry.addData("left", leftmotor.getCurrentPosition());
    telemetry.addData("right", rightmotor.getCurrentPosition());
```

```
            telemetry.update();
        }
    }
}
```

Revisão #3
Criado 19 dezembro 2023 12:13:03 por Enzo Coutinho
Atualizado 19 dezembro 2023 13:57:02 por Enzo Coutinho