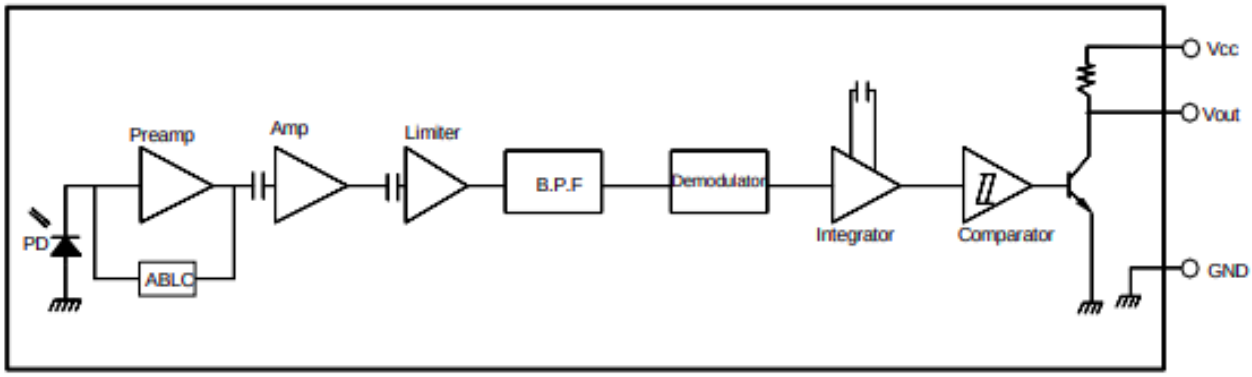
IRController – Desenvolvendo sua própria biblioteca do IRremote em C++

Autor: Lucas Campos Achcar

Há alguns meses atrás, eu pretendia aprender como a biblioteca de IR funcionava, então, procurei sobre referências de como o circuito de transmissão e recepção funcionava e acabei encontrei um artigo do site sbproject sobre o protocolo NEC que é basicamente um protocolo universal de IR: https://www.sbprojects.net/knowledge/ir/nec.php. Lendo o artigo, decidi escrever um código para aprender um pouco mais sobre o protocolo, na escrita do código, acabei utilizando o C++, pois acho bacana a utilização de classes para a separação das funções, e claro, da um ar de elegância ao código.

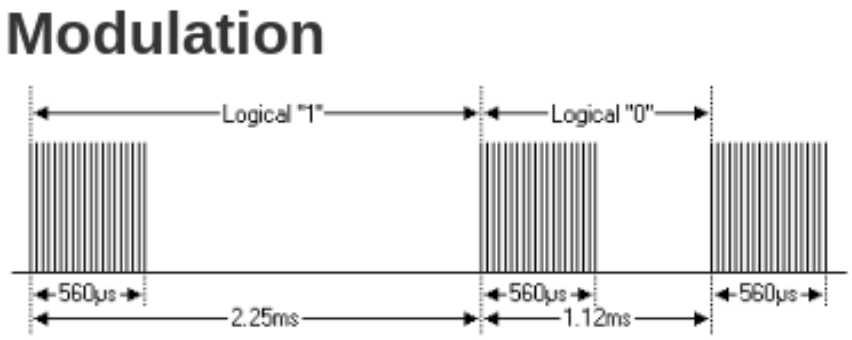
Antes de partir para o código, irei explicar um pouco sobre o protocolo. Tudo começa pela a modulação, você que conhece sobre onda portadora e onda modulante já deve ter noção sobre isso. A portadora da onda de IR é uma onda de frequência de 37.917 kHz, claro, isso no protocolo NEC, alguns protocolos proprietários utilizaram outras frequências. Deixando claro, não é a frequência da luz infravermelho que é modulada, o que ocorre é apenas a modulação por pulsos (liga/desliga) com essa frequência base. Na Figura 2 é possível notar essa alta frequência nos níveis altos, a remoção das altas frequências ocorre via hardware utilizando-se filtros passa baixas, veja a Figura 1, isso permite que apenas dados reais sejam recebidos pelo arduino. Ai o leitor me pergunta, “ah, se o hardware faz tudo, o que sobrou para o software ?” Bom, ainda temos que interpretar os dados, ou seja, a informação transmitida pelo transmissor, e é isso que nosso código irá fazer.

Figura 1 – Diagrama de blocos do receptor IR AX-1838HS



Fonte: http://dalincom.ru/datasheet/AX-1838HS.pdf

Figura 2 – Modulação e Bit Logical do protocolo NEC



Fonte: https://www.sbprojects.net/knowledge/ir/nec.php

A Figura 2 demostra a modulação que ocorre e os ciclos de tempos que representa cada bit, para o bit lógico alto, temos 560 microsegundos ligado em um tempo total de 2.25 milisegundos, vamos chamar de comprimento de tempo, esse nome irá ficar mais claro futuramente. Para o bit lógico baixo, temos os mesmo 560 microsegundos ligados, porém, com o tempo total de 1.12 milisegundos, ou seja, podemos identificar os bits a partir desses duty cycle. A transmissão de dados de uma forma resumida pode ser vista na Figura 3.

Figura 3 – Transmissão de dados do protocolo

Fonte: https://www.sbprojects.net/knowledge/ir/nec.php

A Figura 3 nos mostra como o protocolo irá funcionar. Primeiro, temos um tempo que indica o inicio da transmissão que corresponde a 13.5 milisegundos, sendo 9 milisegundos em high e 4.5 milisegundos em low, após isso, temos os bits de transmissão com o padrão informado na Figura 1. A transmissão total é de 32 bits divididos em 16 bits para address e 16 bits para command, os primeiros 8 bits do address é a mensagem real, os outros 8 bits do address é os primeiros 8 bits da mensagem real, porém, com inversão dos bits (para servir como uma verificação), e os últimos 16 bits é para indicar o command. No código, não foi utilizado o address de verificação ou mesmo a separação do command, apenas recebemos os 32 bits e salvamos em um buffer, como será apresentado mais a frente, porém, pode-se criar uma classe que receba esse buffer e faça todas as separações e verificações possíveis e, informar o usuário com segurança os dados, mas, como não era o objetivo quando o código foi escrito, essa parte não foi implementada. Outra função que não foi implementando é a repetição, ou seja, quando uma tecla é pressionada, mas o artigo do NEC possui todas as informações necessárias para a implementação.

Bom, temos todas as informações necessárias para a implementação do protocolo, na verdade, acho que faltou um. Como estamos utilizando um sistema digital para a

leitura dos dados em high e low, existe um tempo para fazer essa leitura, pois internamente dentro do arduino, temos um DAC (Conversor Analógico Digital), e como fazer essa leitura de forma segura ?, podendo-se afirmar que um bit da Figura 1, esteja em high ou low ? Bom, a ideia é sempre fazer a leitura na metade do tempo dos sinais, porém, nos testes finais, percebeu-se que existia uma perda e/ou confusão muito grande de dados se a amostragem fosse feita dessa maneira, portando, optou-se em amostrar o sinal sempre em 2/3 do tempo antes da transição, por exemplo, se o sinal inicial é de 9 ms, será feito a amostragem em 6 ms, o tempo que ocorrerá a mostragem de cada sinal pode ser visto na Figura 4 (linhas 11 até 15 do código).

Figura 4 – Tempo de amostragem dos cicles da transmissão

```
11  #define LEAD_CODE_CYCLE_TIME_HIGH 9 - 9 / 3
12  #define LEAD_CODE_CYCLE_TIME_LOW 4.5 - 4.5 / 3
13  #define RECEIVE_CODE_CYCLE_TIME 0.56 - 0.56 / 3
14  #define BIT_1_CYCLE_TIME 1.12 - 1.12 / 3
15  #define BIT_0_CYCLE_TIME 0.56 - 0.56 / 3
```

Fonte: Autor

Na linha 11 e 12 da Figura 4, temos o tempo da amostragem que corresponde o inicio da transmissão, na linha 13, temos o tempo de amostragem que indica quando ocorrerá a transmissão de um bit, na linha 14 e 15, temos os tempos de amostragens que indicam qual nível lógico está sendo transmitido, high (bit 1) ou low (bit 0).

Figura 5 – Classe IRController

```
19  class IRController {
20    protected:
21      unsigned PIN_INPUT;
22
23      double nextReadTime;
24      double lastTime;
25      double currentTime;
26
27      double initialRegisterLOW;
28      double lastRegisterLOW;
29
30    double initialRegisterHIGH;
31      double lastRegisterHIGH;
32
33      bool lastRead;
34
35      double freqPulse;
36
37      STATUS stats;
38
39      unsigned indexBuffer;
40      bool buffer[AMOUNT_BYTE];
41      int hex;
42
43    public:
44      IRController(unsigned, double);
45      void resetController();
46      void readSignal();
47      double convertMillisToMicros(double Millis);
48      long unsigned getCode();
49      void clearBuffer();
50  };
```

Fonte: Autor

Partindo para o código, na Figura 5 temos a classe principal que irá gerenciar o controle do IR, em protected, temos algumas váriaveis necessárias para o estado de controle do código, sendo:

- ***PIN_INPUT*** → Pino que será a entrada do receptor de IR (Vout da Figura 1);
- ***nextReadTime*** → O tempo de leitura do próximo bloco de recepção;
- ***lastTime*** → O tempo da última leitura;
- ***currentTime*** → O tempo corrente;
- ***initialRegisterLOW*** → O tempo inicial quando ocorre uma leitura em low;
- ***initialRegisterHIGH*** → O tempo inicial quando ocorre uma leitura de high;
- ***lastRegisterLOW*** → O tempo final da leitura em low;
- ***lastRegisterHIGH*** → O tempo final da leitura em high;

- ***lastRead*** → Registra o último estado (LOW ou HIGH) da última leitura;
- ***freqPulse*** → Indica a frequência de transmissão dos dados (37.917 kHz para o protocolo NEC como já informado);
- ***stats*** → É um enum do tipo **STATUS** que indica o estado em que se encontra a transmissão (veja Figura 6), o código se comporta como uma máquina de estado, sabendo se está iniciando a transmissão (***LEAD_CODE_HIGH*** e ***LEAD_CODE_LOW***), iniciando a transmissão de um bit (***RECEIVE_CODE***) ou se a transmissão é do bit em si (***BIT_RECEIVE***);
- ***IndexBuffer*** → Como a transmissão é feita em 32 bits de cada vez, o indexBuffer marca o index 0 no inicio da transmissão, e a cada bit irá incrementar 1 unidade no buffer até completar os 32 bits previstos da transmissão;
- ***buffer*** → É o buffer principal, ao final da transmissão, esse buffer irá guardar o resultado da transmissão dos 32 bits totais;
- ***hex*** → Ao final da transmissão, o buffer de 32 bits irá ser convertido para um inteiro , esse inteiro é o hex.

Figura 6 – Estado de transmissão

```
9   enum STATUS { LEAD_CODE_HIGH, LEAD_CODE_LOW, RECEIVE_CODE, BIT_RECEIVE };
```

Fonte: Autor

Portanto, essas variáveis irão fazer o controle da máquina de estado na hora da recepção dos bits.

Além das variáveis de estado, na Figura 5, também temos as funções e métodos de controle que irão gerenciar cada etapa dos estados, vamos ver com um pouco mais cada um detalhadamente.

Vamos iniciar pelo construtor (Figura 6), o construtor é um método que possui o mesmo nome da classe, esse método é chamado apenas uma vez quando a classe é instanciada, ou seja, criado um objeto.

Figura 6 – Construtor IRController

```
71  IRController::IRController(unsigned PIN_INPUT, double freqPulse) {
72    this->PIN_INPUT = PIN_INPUT;
73    this->freqPulse = freqPulse;
74
75    pinMode(PIN_INPUT, INPUT);
76
77    resetController();
78  }
```

Fonte: Autor

O construtor irá receber dois parâmetros (argumentos), o primeiro unsigned irá informar o pino de entrada do módulo IR, o segundo argumento é a frequência da portadora, na linha 72 e 73 atribuimos as variáveis de escopo local para o do escopo do objeto, na linha 75 iniciamos a porta como input e na linha 77 fazemos um reset nas variáveis bases.

Figura 7 – Método resetController

```
80  void IRController::resetController() {
81    /* 10^6 microSecond -> 1 second */
82    nextReadTime = (1.0f / freqPulse)*pow(10, 6);
83
84    stats = LEAD_CODE_HIGH;
85
86    indexBuffer = 0;
87  }
```

Fonte: Autor

Na Figura 7 temos a função ***resetController*** que é responsável por fazer a inicialização ou reinicialização das variáveis de controle, na linha 82 nos iniciamos a variável ***nextReadTime*** com o tempo inicial da frequência da portadora, na linha 84 iniciamos a variável stats como ***LEAD_CODE_HIGH***, fazendo com que a primeira leitura ocorra no nível lógico alto (lembrando que o start da transmissão é um bit em estado high de 9 ms), por último, na linha 86, iniciamos o index do buffer na posição 0 (inicio do buffer).

Figura 8 – Função convertMillisToMicrons

```
159  double IRController::convertMillisToMicros(double Millis) {
160    return Millis*pow(10, 3);
161  }
```

Fonte: Autor

A função da Figura 8 é bem simples e autoexplicativo, recebemos um tempo em milisegundos e a função retorna o valor convertido para microsegundos.

Figura 9 – Método clearBuffer

```
64  void IRController::clearBuffer() {
65    if(stats == LEAD_CODE_HIGH) {
66      for(unsigned i = 0; i < AMOUNT_BYTE; ++i)
67        buffer[i] = 0;
68    }
69  }
```

Fonte: Autor

O método ***clearBuffer*** da Figura 9 também é bem simples, na linha 65 é verificado se o estado atual da máquina de estado é um ***LEAD_CODE_HIGH*** (Ainda não ocorreu a transmissão), se isso é verdadeiro, apenas limpamos o buffer dos dados.

Figura 10 – Função getCode

```
52  // Convert buffer to hex
53  long unsigned IRController::getCode() {
54    long unsigned retn = 0;
55    if(stats == LEAD_CODE_HIGH) {
56      for(unsigned i = 8; i < AMOUNT_BYTE; ++i) {
57        retn = retn << 1;
58        retn |= buffer[i];
59      }
60    }
61    return retn;
62  }
```

Fonte: Autor

A função mostrada na Figura 10 é um método do tipo get que retorna o buffer em formato inteiro, isso apenas ocorre se o estado atual da máquina de estado for um ***LEAD_CODE_HIGH***, ou seja, não conseguimos acessar o ***getCode*** quando estamos recebendo dados, apenas quando estivermos no final da transmissão ou o emissor não transmitir nada. A conversão de um buffer de matriz de 32 posições para um long unsigned é feito na linha 57 e 58, utilizando um método de shift register (simbolo <<) e a lógica ‘ou’ que representa o simbolo ‘|’ (pipeline), com isso, podesse mover os bits à esquerda do registrador como shift register e com a lógica ‘ou’ adicionar bits ao primeiro bit à direita do registrador, ao final, teremos um inteiro sem sinal contendo os 32 bits do buffer.

Por último temos a cereja do bolo, a função ***readSignal***. Irei dividir o código em partes para facilitar sua compreensão.

Figura 11 – readSignal (Parte 1)

```
89  void IRController::readSignal() {
90    currentTime = micros();
91    if((currentTime - lastTime) > nextReadTime) {
92
93      bool valueRead = !digitalRead(PIN_INPUT);
94
95      // Register de LOW and HIGH last and current time to checking
96      if(valueRead == HIGH) {
97        lastRegisterHIGH = currentTime;
98
99        if(lastRead != valueRead) {
100         initialRegisterHIGH = currentTime;
101         lastRegisterLOW = currentTime;
102       }
103     } else  {
104       lastRegisterLOW = currentTime;
105
106       if(lastRead != valueRead) {
107         initialRegisterLOW = currentTime;
108         lastRegisterHIGH = currentTime;
109       }
110     }
111
112     double RegisterHigh = lastRegisterHIGH - initialRegisterHIGH;
113     double RegisterLow = lastRegisterLOW - initialRegisterLOW;
```

Fonte: Autor

Na Figura 11 temos uma fração da função ***readSignal***. Iniciamos na linha 90 com a obtenção do tempo inicial da execução da função, na linha 91 verificamos se o tempo inicial menos a última lida (se for a 1º vez, o ***lastTime*** será 0) é maior que a próxima leitura, se for verdadeiro, entramos no bloco da condição. Na linha 93 fazemos a leitura da porta digital do IR, o bloco de código da linha 96 até a linha 110 faz uma anotação do tempo inicial e final dos estados low e high em que foram lidos, por exemplo, se ocorrer uma leitura em high, anotamos em ***lastRegisterLow*** e em ***initialRegisterHIGH*** o tempo corrente, pois, o final do sinal low é também o inicio do sinal high, a mesma coisa acontece quando ocorre a inversão do sinal, ou seja, temos sempre o inicio e o fim de cada estado, isso é feito a cada troca de estado da porta digital (essas informações é o que chamamos de comprimento de tempo dos níveis lógicos), pois temos o controle dela pela variável ***lastRead*** que guarda o último estado da porta antes de ocorrer a troca, essas informações são armazenadas, pois, será útil para saber quanto tempo levou a transição dos estados. As linhas 112 e 113 nos informa quanto tempo o sinal está em low ou high a partir do tempo corrente.

Figura 12 – Continuação da Função readSignal

```
115     if(lastRead != valueRead) {
116       // Lead Code
117       if((RegisterLow) >= convertMillisToMicros(LEAD_CODE_CYCLE_TIME_LOW) && stats == LEAD_CODE_LOW) {
118         // Waiting Pulse Low Finish to Register Bit
119         stats = RECEIVE_CODE;
120       } else if((RegisterHigh) >= convertMillisToMicros(LEAD_CODE_CYCLE_TIME_HIGH) && stats == LEAD_CODE_HIGH){
121         // Waiting Next Pulse Low (LEAD_CODE)
122         stats = LEAD_CODE_LOW;
123       } else if(stats == RECEIVE_CODE && (RegisterHigh) >= convertMillisToMicros(RECEIVE_CODE_CYCLE_TIME)) {
124         stats = BIT_RECEIVE;
125       } else if(stats == BIT_RECEIVE) {
126         // Receive Code
127         if((RegisterLow) >= convertMillisToMicros(BIT_1_CYCLE_TIME)) {
128           // Bit 1
129           buffer[indexBuffer++] = 1;
130           stats = RECEIVE_CODE;
131         } else if((RegisterLow) >= convertMillisToMicros(BIT_0_CYCLE_TIME)) {
132           // Bit 0
133           buffer[indexBuffer++] = 0;
134           stats = RECEIVE_CODE;
135         }
136       }
```

Fonte: Autor

A Figura 12 é a continuação do código da Figura 11. O bloco de código da condição 115 só é executada se ocorrer uma troca de estado da porta digital do IR, ou seja, vamos sempre verificar os estados quando ocorrer uma transição do sinal de low para high ou de high para low. Antes de ir para a linha 117, vamos falar da linha 120, nessa linha, verificamos se o comprimento do tempo do estado em high é maior que ***LEAD_CODE_CYCLE_TIME_HIGH***, que no protocolo NEC, corresponde a 2/3 de 9 milisegundos, além disso, temos que estar no estado ***LEAD_CODE_HIGH***, que é o estado inicial da transmissão, dentro do bloco '*else if*', mudamos o estado para ***LEAD_CODE_LOW***. Agora, na linha 117, verificamos se o comprimento do sinal low foi maior que ***LEAD_CODE_CYCLE_TIME_LOW*** que corresponde a 2/3 de 4.5 ms , verificamos também se o estado que esperamos é um ***LEAD_CODE_LOW*** (estado atribuido anteriormente em high no inicio da transmissão, ou seja, é necessário o estado de ***LEAD_CODE_HIGH*** para ocorrer o estado de ***LEAD_CODE_LOW***), se verdadeiro, o estado irá mudar para ***RECEIVE_CODE***, estado que define o inicio da transmissão dos dados em si. Na linha 123, verificamos se o estado é um ***RECEIVE_CODE*** e se o comprimento do tempo do sinal high corresponde a 2/3 de 0.56 ms (***RECEIVE_CODE_CYCLE_TIME*)**, ou seja, o tempo correspondente ao inicio da transmissão de um bit. Na linha 125, se o estado atual for de receber um bit, entramos no bloco, verificamos o comprimento de tempo que ficou no estado de low, se o comprimento do tempo for 2/3 de 1.12 ms (***BIT_1_CYCLE_TIME***), o bit corresponde é o 1, se for 2/3 de 0.56 ms (***BIT_0_CYCLE_TIME***), corresponde ao bit 0, assim, salvamos no buffer de 32 bits e incrementamos o index do buffer (linhas 129 e 133), e voltamos ao estado de ***RECEIVE_CODE*** para a leitura do próximo bit.

Figura 13 – Código final do signalRead

```
138         // Test -> Print Buffer and Reset IRRemote
139         if(indexBuffer == AMOUNT_BYTE) {
140           if(DEBUG_MODE) {
141             for(unsigned i = 0; i < AMOUNT_BYTE; i++)
142               Serial.print(buffer[i]);
143             Serial.println();
144             Serial.print("HEX ");
145             Serial.print(getCode(), HEX);
146             Serial.println();
147           }
148           resetController();
149         }
150       }
151
152       lastTime = micros();
153
154       // Change the last value read
155       lastRead = valueRead;
156     }
157   }
```

Fonte: Autor

A Figura 13 mostra o código final do ***signalRead***. Na linha 139, verificamos se o ***indexBuffer*** é igual a ***AMOUNT_BYTE*** (32 bits) se verdadeiro, entramos no bloco. O bloco em si é simples, exibimos no terminal do serial todos os bits e inclusíve seu hex decimal, após isso, fazemos uma chamada de função do ***resetController*** para resetar as váriaveis de controle e aguardar uma próxima transmissão.

Figura 14 – Código de exemplo - utilizando nossa classe IRController

```
163  IRController *IR;
164
165  /* Test */
166
167  unsigned LED[] = {8, 9, 10};
168  bool LEDstats[] = {false, false, false};
169
170  void setup() {
171    Serial.begin(9600);
172
173    pinMode(LED[0], OUTPUT);
174    pinMode(LED[1], OUTPUT);
175    pinMode(LED[2], OUTPUT);
176
177    // Port 03 and freqPulse 37.917Khz
178    IR = new IRController(3, 37.917*pow(10, 3));
179    Serial.println("IRRemote Controller");
180  }
181
182  void loop() {
183    IR->readSignal();
184
185    unsigned long codeValue = IR->getCode();
186    if(codeValue) {
187      IR->clearBuffer();
188
189      Serial.print("HEX ");
190      Serial.print(codeValue, HEX);
191      Serial.println();
192
193      /* Test */
194      if(codeValue == 0x7B807F) {
195        LEDstats[0] = !LEDstats[0];
196        digitalWrite(LED[0], LEDstats[0] ? HIGH : LOW);
197      } else if(codeValue == 0x7BC03F) {
198        LEDstats[1] = !LEDstats[1];
199        digitalWrite(LED[1], LEDstats[1] ? HIGH : LOW);
200      } else if(codeValue == 0x7BA05F) {
201        LEDstats[2] = !LEDstats[2];
202        digitalWrite(LED[2], LEDstats[2] ? HIGH : LOW);
203      }
204    }
205  }
```

Fonte: Autor

Para finalizar, a Figura 14 nos mostra o bloco final do código onde colocamos um exemplo do funcionamento da nossa classe/biblioteca. Iniciamos com a linha 163, nessa linha, criamos um ponteiro do tipo ***IRController***, o ponteiro irá receber um endereço do objeto que será instanciado na linha 178, a palavra reservada ***new*** cria uma instância da classe ***IRController***, e os parâmetros são aquelas informadas na Figura 6, sendo o 1º argumento, o pino de dados do IR e o segundo argumento, a frequência do dispositivo. As linhas 167 e 168 são matrizes de estados para o controle dos nossos leds de exemplo, e no método de ***setup*****,** as linhas 173, 174 e 175, apenas faz a inicialização das portas como output, e, para finalizar, a linha 171 inicializa o nosso serial.

Ainda na Figura 14, no nosso método de ***loop***, iniciamos na linha 183 lendo o sinal da transmissão com a função ***readSignal***, na linha 185, capturamos o que tem no buffer

da transmissão com a função ***getCode***, e na linha 186 verificamos se é diferente de 0x00, se verdadeiro, apenas exibimos no monitor serial o valor hex (linhas 189 até 191) e acionamos cada led individualmente. Observe que aqueles valores de hex decimal foram obtidos de modo empirico, executei o código e simplesmente apertei os botões do controle para obter cada hex no monitor serial. Seguindo o controle da Figura 15, o primeiro hex (**0x7B807F**) da linha 194 corresponde ao botão POWER (Vermelho) e acende o led do pino 8, o segundo hex (**0x7BC03F**) corresponde ao botão VOL+ e acende o led do pino 9 e o terceiro hex (**0xFD40BF**) corresponde ao botão FUNC/STOP e acende o led do pino 10.

Figura 15 – Controle Emissor IR

Fonte: Tinkercad

O link para o código fonte e o projeto no tinkercad pode ser encontrado ao final desse artigo.

Código Fonte
https://github.com/AchcarLucas/Engenharia-F-sica-USP-/blob/master/Ardu%C3%ADno/IRRemote%20DIY/remote/remote.ino

Projeto Tinkercad
https://www.tinkercad.com/things/3kyn4YPRWlY