Capítulo 5

Un primer programa

1. La interrupción necesaria

1.1 Una librería

Los datos DCC se envían a las vías mediante la modulación de señales eléctricas cada 58 µs. Un bit 1 corresponde a una señal positiva de 58 µs, seguida de una señal negativa de la misma duración. Un bit `0` es una señal positiva de 116 µs, esto es, dos señales de 58 µs, seguidas de una señal negativa de la misma duración. Así, todo es un múltiplo de 58 µs. Basta con una interrupción ajustada a esta duración para que la señal DCC se genere con la precisión necesaria.

En el mundo Arduino, programar interrupciones no es complicado. Existen varias bibliotecas para gestionarlas. La biblioteca msTimer, que se propone como predefinida, utiliza una base de tiempo de 1 ms, que no es lo suficientemente precisa para generar una señal DCC. Por lo tanto, se utiliza la biblioteca FlexiTimer2, ya que ofrece una resolución de 1 µs.

Esta biblioteca no está disponible en el gestor de Arduino, por lo que es necesario descargar el archivo por separado para luego utilizar la opción **Añadir biblioteca ZIP**. El archivo está disponible en la dirección: https://playground.arduino.cc/Main/FlexiTimer2/

Esta biblioteca da acceso a tres funciones: `set`, `start` y `stop`, que permiten respectivamente configurar, iniciar y detener una interrupción. Solo las dos primeras son necesarias para crear una señal.

FlexiTimer2::set(units, resolution, intFunction)

- `units`: número de incidencias en la base de tiempo antes de llamar a la interrupción.
- `resolution`: base de tiempo (expresada en segundos).
- `intFunction`: función llamada cuando se dispara la interrupción.

Una vez definida la base de tiempo, se llama a la función `intFunction` a una cadencia correspondiente a `units` x `resolution`. Se trata de una función callback (un principio ya presente en la biblioteca Wire que gestiona el bus I²C). La función en cuestión no toma ningún parámetro como entrada.

FlexiTimer2::start()

Esta función inicia realmente la interrupción, no necesita ningún parámetro.

FlexiTimer2::stop()

Detiene el procesamiento de las interrupciones.

1.2 Ejemplo concreto

Este código es un ejemplo de cómo implementar una interrupción. No puede ser más sencillo.

```
#include <FlexiTimer2.h>

void dccInterrupt()
{
  static int counter=0;
  digitalWrite(LED_BUILTIN,!(counter++&0x1000));
}

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
  FlexiTimer2::set(1, 0.000058, dccInterrupt);
  FlexiTimer2::start();
}

void loop()
{
  delay(10000); // Nada más a hacer aquí
}
```

En la función `setup`, se activa la salida que controla el led integrado; este va a utilizarse en la interrupción. Enseguida se inicializa la interrupción; la configuración provoca una interrupción cada 0,000058 segundos, es decir, 58 µs, la cadencia necesaria para la generación de una señal DCC. Entonces, se inicia la interrupción.

© Editions ENI - All rights reserved

La función `loop` no hace nada. Esto es lógico, ya que el propósito de este ejemplo es hacerlo todo en la interrupción. Así que esperamos un segundo antes de devolver el control. Siempre es preferible poner una espera si el bucle principal no hace nada; si no, el Arduino se pasa el tiempo en bucle, lo que no resulta útil en absoluto.

Queda la función `inter`, a la que se llama automáticamente cada 58 μs, ya que es el procesamiento de la interrupción. Se declara un contador, la palabra clave `static` permite mantener el valor de esta variable local entre dos llamadas a la interrupción. A continuación, el contador se incrementa y el led se enciende según el estado del bit 12 del contador. Está, por lo tanto, apagado durante 4096 llamadas y luego encendido el mismo número de llamadas. Hay un parpadeo cada 8192 llamadas, esto es, 8192 x 58 μs = 0,475 segundos, o algo más de dos destellos por segundo, que es lo que se visualiza. Ahora que la interrupción funciona, es posible usarla para generar la señal DCC y ¡esto es un poco más complicado!

2. La creación de la señal DCC

2.1 La teoría

Entre el botón que se gira para ajustar la velocidad del tren y la señal que se envía a las vías, se necesitan varias transformaciones. Esto comienza con el formateo de los datos que se leen en primera instancia. Se puede tratar de una entrada analógica a la que se conecta un potenciómetro, un teclado o interruptores. Estos datos se formatean y luego se convierten a la forma de un paquete DCC, constituido de un número variable de bytes:

- 1 o 2 bytes para el direccionamiento
- 1 a 3 bytes para el comando
- 1 byte de suma de verificación

Por tanto, este paquete DCC puede tener entre 3 y 6 bytes, según el caso. Los paquetes solo se crean y almacenan en caso de cambio de velocidad de las teclas de función; es inútil hacerlo sistemáticamente si nada ha cambiado.

Este paquete se almacena, a continuación, en la memoria, ya que los datos DCC deben reenviarse regularmente, aunque no cambien, pues de lo contrario las locomotoras acabarán parándose solas. Se trata de una cuestión de seguridad. Este almacenamiento adopta la forma de una tabla que lista todos los paquetes que se van a transmitir. Cuando se debe almacenar un paquete, primero hay que comprobar si va a sustituir a uno ya existente o si es uno nuevo.

Finalmente, el envío se realiza mediante una rutina en interrupción llamada cada 56 µs. Se necesitan dos llamadas para enviar un bit 1 y cuatro llamadas para enviar un bit 0. Además de los datos que componen el paquete en sí, también hay que enviar la cabecera y añadir los bits de separación y de STOP.

2.2 La creación de los bits

Cada vez que se llama a una interrupción, cada 58 µs, solo se transmite una señal. Esto corresponde, según el caso, a medio bit (en el caso de un bit 1) o a un cuarto de bit (en el caso de un bit 0). Por lo tanto, es necesario guardar el estado actual para saber a qué estado pasar durante la siguiente llamada. Esto es posible gracias a las variables de estado. Hablar de variables de estado es hablar de máquina de estados, que generalmente se conoce como autómata finito.

El generador de bits es el autómata de más bajo nivel. Se basa en dos variables:

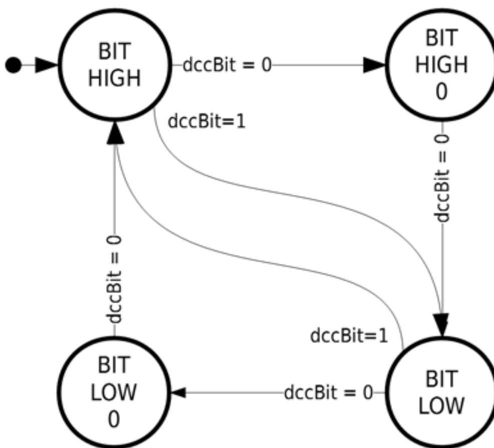
- `dccBit` (variable de entrada): indica si se debe emitir un bit 1 o un bit 0.
- `dccSubBit` (variable de estado): indica en qué punto se encuentra el envío del bit.

La salida DCC debe tomar estados sucesivos en función del valor del bit que se ha de enviar:

Para un bit 0, los estados son: 1 – 1 – 0 – 0.

Para un bit 1, los estados son: 1 – 0.

Esto puede traducirse en el siguiente autómata finito (máquina de estados):



© Editions ENI - All rights reserved

- dccBit puede tomar dos estados: 0 o 1.
- dccSubBit puede tomar cuatro estados: BIT_HIGH, BIT_HIGH_0, BIT_LOW y BIT_LOW0.

Esto nos lleva al siguiente programa. Se comienza por definir en qué salida se va a entregar la señal DCC. A continuación, se crean las constantes y variables utilizadas por el autómata.

```
#define DCC_OUT1 2

#define DCC_BIT_HIGH 0
#define DCC_BIT_HIGH0 1
#define DCC_BIT_LOW 2
#define DCC_BIT_LOW0 3

byte DccBit; // Bit que se está enviando
byte DccSubBit; // Parte del bit que se está enviando
```

La implementación práctica de un autómata suele realizarse mediante una estructura switch/case. Así, cada uno de los casos tratados corresponde a una de las etapas del gráfico de transición. El código puede parecer denso, pero en realidad es solo la representación del gráfico en forma de programa. Todo ello está integrado en una función dccInterrupt, que será llamada por una interrupción.

```
void dccInterrupt(void)
{
  switch(DccSubBit)
  {
  case DCC_BIT_HIGH :

    // Aquí es donde debe colocarse el código
    // presentado en la siguiente sección
    digitalWrite(DCC_OUT1,HIGH);
    if(DccBit)
      DccSubBit=DCC_BIT_LOW;
    else
      DccSubBit=DCC_BIT_HIGH0;
    break;
  case DCC_BIT_HIGH0 :
    digitalWrite(DCC_OUT1,HIGH);
    DccSubBit=DCC_BIT_LOW;
    break;
  case DCC_BIT_LOW :
    digitalWrite(DCC_OUT1,LOW);
    if(DccBit)
      DccSubBit=DCC_BIT_HIGH;
    else
      DccSubBit=DCC_BIT_LOW0;
    break;
  case DCC_BIT_LOW0 :
    digitalWrite(DCC_OUT1,LOW);
    DccSubBit=DCC_BIT_HIGH;
    break;
  }
}
```

Tal como está, este código no es muy útil. Aún se requiere inicializar la variable de entrada `DccBit` con los bits que se deben enviar. Por el momento, el código necesario se sustituye por un comentario. La definición del valor de este bit la realiza otro autómata.

2.3 La creación de las tramas

Este segundo autómata está un nivel por encima. Envía todo el contenido de una trama DCC; es decir, la cabecera, el bit START y, a continuación, todos los bytes de la trama (incluido el código de verificación) seguidos del bit STOP.

Para su funcionamiento se utilizan las siguientes variables:

- `dccDataMode` (variable de estado principal): permite saber dónde estamos.
- `dccStackUsed` (variable de entrada): indica el número de paquetes que deben enviarse.
- `dccHeaderCount` (variable de estado secundario): cuenta los bits de la cabecera.
- `dccByteCount` (variable de estado secundario): indica el número de byte que hay que enviar.
- `dccBitShift` (variable de estado secundario): indica el número de bit que hay que enviar.

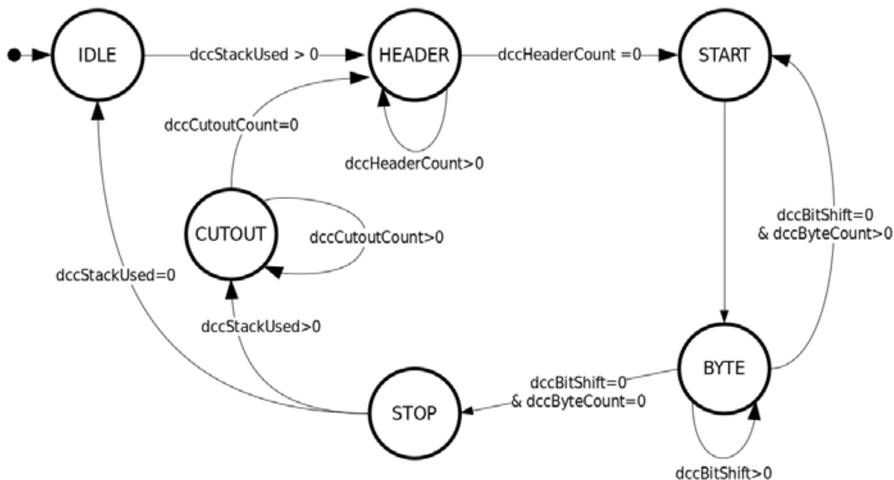
La variable de estado `dccDataMode` puede tomar los siguientes valores:

- `IDLE`: en espera (solo se utiliza si no hay ninguna locomotora activa).
- `HEADER`: envío de la cabecera (20 bits 1).
- `START`: envío de un 0 para marcar el inicio de un byte de datos.
- `BYTE`: envío del byte de datos propiamente dicho.
- `STOP`: envío de un 1 para marcar el final de los datos.
- `CUTOUT`: corte de la señal para la retroinformación.

Las tres variables de estado secundarias solo están activas en una parte del autómata:

- `dccHeaderCount`: permite ejecutar un bucle sobre el estado `HEADER` para contar los bits de la cabecera.
- `dccByteCount`: permite ejecutar un bucle entre los estados `BYTE` y `START`.
- `dccBitShift`: permite ejecutar un bucle sobre el estado `BYTE` cambiando de bit cada vez.
- `dccCutoutCount`: permite contar la duración del intervalo de corte.

© Editions ENI - All rights reserved



Se comienza por definir las constantes y variables utilizadas por el autómata.

```
#define DCC_PACKET_SIZE 6 // Tamaño máximo de un paquete DCC

byte DccDataMode;    // Variable de estado
byte DccStackUsed=1; // Número de paquetes que deben enviarse
byte DccHeaderCount; // Recuento de bits 1 de la cabecera
byte DccByteCount;   // Índice del byte que se está enviando
byte DccCutoutCount; // Recuento del intervalo de corte
byte DccBitShift;    // Recuento de los bits del byte que se están enviando

byte packetData[DCC_PACKET_SIZE]; // Paquete de datos que se han de enviar
byte packetSize; // Tamaño del paquete que se tiene que enviar
```

Puesto que el autómata de creación de bits cambia el valor del bit que se debe enviar cuando está en el estado `BIT_HIGH`, es después de la línea `case DCC_BIT_HIGH` cuando se implementa el código. Retoma la estructura del autómata tal y como se acaba de definir.

```
    switch(DccDataMode)
    {
    case DCC_PACKET_IDLE :
      if(DccStackUsed)
      {
        DccDataMode=DCC_PACKET_HEADER;
        DccHeaderCount=DCC_HEADER_SIZE;
      }
      break;
    case DCC_PACKET_HEADER :
      DccBit=1;
      if(!--DccHeaderCount)
      {
        DccDataMode=DCC_PACKET_START;
        DccByteCount=0;
      }
      break;
```