Servos, DCC & Arduinos

Mark Riddoch

For some time now I have been thinking about ways to make things other than the locomotives move on my layouts. Radio Control servos have become an accepted way to control the points and semaphore signals, but we could use them for more if we had a good way to integrate the proportional control of them into our control systems. Currently MERG have with the servo4 and CANServo8 that allow us to have two position actuators. The Sema4 allows bounce to be simulated but if you want to animate a crane we probably want more than just the ability to set two points and have the crane move between them. We can use custom devices like the simple servo testers to do better with positioning, but it means a separate set of controls for your animated devices, is that what we really want?

Proportional Servo Movement

The first thing I thought about was to have a dockside crane that I could swing the jib and wind the hook up and down. I embarked on modifying a CANServo8 to take position information messages rather than just binary ON/OFF events. I also started to convert a CANACE8 to be a CAN A/D convertor so that I could use a set of pots to send CBUS events that would move the servo to a particular position. This all seemed great, but I still needed separate controls, although it used the same bus wiring. What if I wanted to do something that was rail mounted, CBUS would not help me. Why not put something together that would work for both static items, like the dock side crane, and also mobile devices, e.g. a break down crane. Could I make use of existing controls to do this? My thoughts turned to using DCC for controlling the servos.

Attempt number one was simply to use the setting of the speed knob to control the absolute position: setting the speed to 50% would cause the servo to move to 50%. That works fine, but has the disadvantage that when you switch to another device, e.g. your locomotive, you have to turn the speed down. Switching back to your servo then ends up with a “jump”, since it is very hard to set the speed position accurately to what it was previously. Also I wanted to be able to have several servos in one device, so having just a single speed setting set the servo would mean I would need a decoder per servo - this is starting to get large, especially as I model in 4mm.

The second attempt, and the one which I will describe in more detail here, was to use the speed control still, instead of the speed control setting an absolute position, I would use it to control the speed of the servo movement. I built a decoder that has 3 CV values per decoder: a start position, an end position and a time to move between those two positions. Setting the direction to be forward and turning up the speed control would cause the servo to start moving between the start position and the end position. The speed at which it moved was determined by the speed setting on the controller, setting a speed of 50% would result in the servo moving at half the defined speed, i.e. it would take twice the time programmed in the CV to move all the way from the start to end position. Setting the speed to zero would cause the servo to stop at whatever the current position was. Changing to the backwards direction would cause the servo to move from whatever the current position is towards the start position.

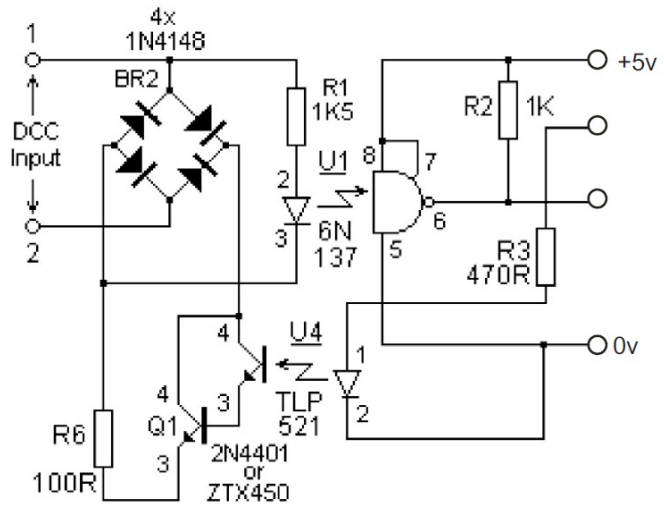
This gave me control to set the servo to any position between the programmed start and end position and have it stop. It also allowed the speed at which the servo moved to be varied from zero to the pre-defined maximum. The issue of multiple servos was addressed by using the function keys to define which servo to move. Turn on function 0 and the first servo would move, turn it off and that servo would stop, regardless of the current speed setting. Switching on function 1 would mean that the speed and direction setting was applied to the second servo and so on. Multiple functions could be on at once, meaning that multiple servos could be moved at once.

I built a prototype using an Arduino, since the DCC library and servo library are freely available and it is a very simple and quick platform for doing this sort of prototyping. I took my prototype along to the Thames Valley Area meeting at Grazeley and showed my attempts to the rest of the members. There was a reasonable amount of interest. We had already done an Arduino tutorial evening on driving servos, and it was agreed that it would be good to do something similar for this DCC

application of servos. We timetabled two evenings as workshop evenings so we could all build a version of this and hopefully share some experiences and knowledge.

DCC Interface Board

The first of these evenings was about building the hardware component; not much was required for the Arduino, just the interface to enable it to receive the DCC packets and to send acknowledgements back to the command station. Here I cheated and just extracted the front end opto-isolator circuit from the MERG accessory decoders. We did not worry about powering the Arduino and servos from the DCC bus. In this prototype the Arduino would be powered from the USB connector that was used to program it. The power for the servos would come from the Arduino onboard regulator. Fortunately one of our members, Dave Ingoldby, was adept at etching PCBs and agreed to etch enough PCBs and make up kits so that everybody who was interested could build an interface.



A month later we have 20 kits and a room full of people ready to start assembling a DCC interface board for the Arduino. Assembly went well and by the tea break just about everybody had assembled a kit and we were ready to start testing them. Amazingly all but 1 interface board worked first time. With that we called it a successful evening and retired ready for part 2 of the workshop. This time we had a room full of people with laptops, various DCC systems, Arduino boards and the interface boards we had previously made. Rather than simply go straight in with the final code I had come up with I decided it would probably be more beneficial for everybody, including me, to go through the stages I had with the Arduino DCC library - known as NmraDcc. If nothing else it was good for me since it reinforced what I had learnt on the way and hopefully it would help others in the group as well.

DCC “hello word”

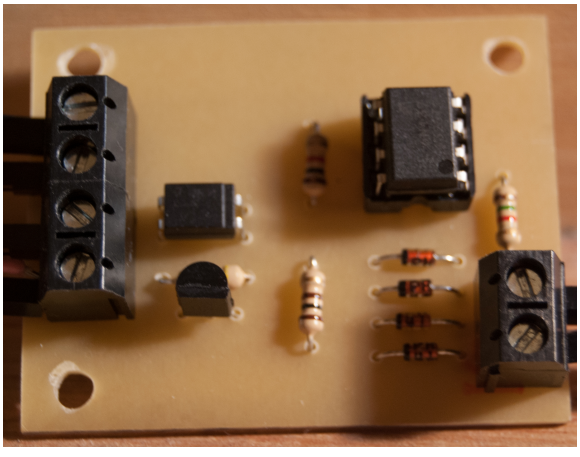
Once everybody had the Arduino IDE installed and copies of the required libraries we started with the DCC library equivalent of “hello world”, a simple Arduino Sketch that would print the details of the speed control setting DCC packet. The first step was the initialization of the DCC Library. This consisted of defining the two Arduino pins that were connected to the interface boards we had previously made and calling the init method in the NmraDcc library. In our case the connection from pin 6 of U1 was connected to the digital pin 2 on the Arduino and the R3 connection was attached to pin 3 of the Arduino. The 5 volts and 0 Volts of the interface circuit where connected to the 5V and GND pins on the Arduino.

```
 #include <NmraDcc.h>

/*
 * DCC Example 1
 *
 * Print all speed control packets for DCC address 3
 */
NmraDcc  Dcc;
DCC_MSG  Packet;
const int DccAckPin = 3;
const int DccInPin = 2;

void setup()
{
  Serial.begin(9600);

  // Configure the DCC CV Programing ACK pin as an output
  pinMode(DccAckPin, OUTPUT);
  digitalWrite(DccAckPin, LOW);

  // Setup which External Interrupt, the Pin it's associated with that we're
  // using and enable the Pull-Up
  Dcc.pin(0, DccInPin, 1);

  // Call the main DCC Init function to enable the DCC Receiver
  Dcc.init(MAN_ID_DIY, 10, FLAGS_OUTPUT_ADDRESS_MODE, 0);
}
```

The NmraDcc library is written so that it has a “loop” routine that must be called periodically in order for the DCC packets to be processed. This is achieved by simply calling the Dcc.process method within the Arduino loop function.

```
void loop()
{
  // You MUST call the NmraDcc.process() method frequently from
  // the Arduino loop() function for correct library operation
  Dcc.process();
}
```

This gets everything setup to process DCC packets but it doesn’t give a way to see what those packets are. The NmraDcc library uses template functions to effectively provide callbacks to the user code when DCC events are received. In the case of the DCC speed packet defining a function with the name notifyDccSpeed will be called whenever a DCC speed packet is received. Our “hello world” example merely printed the information we received to the serial port, the serial port monitor function of the Arduino IDE allows us to view the information printed.



Left: One of the interface boards we built in the first session.

```
/*
* Called with the speed, direction and number of speed steps whenever
* a DCC speed packet is received
 */
void notifyDccSpeed( uint16_t Addr, uint8_t Speed, uint8_t ForwardDir, uint8_t
SpeedSteps )
{
  // If the address is not 3 then simply return
  // Comment this out to have all addresses printed
  if (Addr != 3)
    return;
  Serial.print("DCC Speed ");
  Serial.print(Speed,DEC);
  Serial.print(" Addr ");
  Serial.print(Addr, DEC);
  Serial.print(" ForwardDir ");
  Serial.print(ForwardDir, HEX);
  Serial.print(" SpeedSteps ");
  Serial.println(SpeedSteps, DEC);
}
```

Although a simple example, it gave insight into the way speed is represented in DCC and how DCC systems deal with sending data, i.e. the periodic sending of speed packets even when the speed is not changed. We also had the first building block for our servo decoder.

DCC Functions

The second exercise was to look at the way the function buttons were translated into DCC packets. The basic setup and loop were identical to the first example. A new routine, notifyDccFunc, was added to receive the DCC function packet data.

```
 /*
 * Called to report the state of the function keys
 */
void notifyDccFunc( uint16_t Addr, uint8_t FuncNum, uint8_t FuncState)
{
  // If the address is not 3 then simply return
  // Comment this out to have all addresses printed
  if (Addr != 3)
    return;
  Serial.print(" DCC Func Addr ");
  Serial.print(Addr, DEC);
  Serial.print(" FuncNum ");
  Serial.print(FuncNum, HEX);
  Serial.print(" FuncState ");
  Serial.println(FuncState, HEX);
}
```

This example was much more revealing of some aspects of the history of DCC, the way in which the numbers of functions available have evolved over time. There are two interesting parameters to this call other than the obvious one of address: FuncNum and FuncState. Most people expected the FuncNum to the the selected function and the FuncState to be on or off. This is not the case. The FuncNum parameter is used to select a “bank” of functions, and the FuncState is a bitmap that represents the state of the functions within that bank.

A value of 1 in FuncNum represents F0, F1, F2, F3 and F4, with bit 4 representing the state of F0. If bit 4 is set then F0 is on and if bit 4 is clear then F0 is off. F1 is represented by bit 0, F2 by bit 1, F3 by bit 2 and F4 by bit 3. The reason for the slightly strange bit encoding dates back to before the function packets where introduced and there was merely a lighting function bit in the loco speed packet. A value of 2 in FuncNum represents F5 to F12.

A First Servo Decoder

This gave us the building blocks for the DCC side of the program, so we progressed to the servo side. I had already taken the standard Arduino servo library and wrapped it into another library that provided an interface that had a similar init and loop philosophy to the NmraDcc library.

Arduino libraries are basically C++ classes. I created a class with a constructor that took the Arduino pin to which the servo was connected, a start and end angle and time in seconds to move between those angles at maximum speed. As with the NmraDcc library, the loop method of this class should be called regularly to allow the servo to move smoothly.

```
class DCCServo {
 private:
   ...
 public:
   DCCServo(int, int, int, unsigned int);
   void loop();
   void setSpeed(int, boolean);
   void setActive(boolean);
   void setStart(int);
   void setEnd(int);
   void setTravelTime(int);
};
```

The setSpeed method is called to set the speed at which the servo should move, as a percentage of the maximum speed defined in the constructor and also the direction of travel.

The setActive method is used to connect and disconnect the servo, which allows the servo to be stopped and the control signal removed from the servo to prevent any servo jitter or twitching.

The setStart, setEnd and setTravelTime methods are used to update the values passed to the constructor for the angles and travel time. These come into play later. The next example was the first time we actually started to get the servos to move under DCC control. We updated our setup routine to create 3 instances of the DCCServo class.

```
DCCServo  *servo1, *servo2, *servo3;
const int servoPin = 9;

void setup()
{
  Serial.begin(9600);

  // Configure the DCC CV Programing ACK pin for an output
  pinMode(DccAckPin, OUTPUT);
  digitalWrite(DccAckPin, LOW);

  // Setup which External Interrupt, the Pin it's associated with that
  // we're using and enable the Pull-Up
  Dcc.pin(0, DccInPin, 1);

  // Call the main DCC Init function to enable the DCC Receiver
  Dcc.init(MAN_ID_DIY, 10, FLAGS_OUTPUT_ADDRESS_MODE, 0);

  // Create the 3 instances of the DCCServo class that represents
  // each of the servos we control. The arguments are the pin number,
  // two limits of travel and the time in seconds to move between them
  // at 100% velocity
  servo1 = new DCCServo(servoPin,     10, 130, 30);
  servo2 = new DCCServo(servoPin + 1, 20,  80, 10);
  servo3 = new DCCServo(servoPin + 2,  0,  90, 60);
}
```

In this example the three servos were preset to move between 10 and 130 degrees in 30 seconds, 20 and 80 degrees in 10 seconds and 0 to 90 degrees in 60 seconds. The servos are wired with

the 5V and 0V lines (red and black on some servos) connected to the Arduino 5V and GND connections. The white control lines of each servo are then connected to digital pins 9, 10 and 11 of the Arduino board.

The notifyDCCSpeed routine was replaced with one that calculated the current speed as a percentage and called the setSpeed method of each of the servos with this speed percentage and the forward/reverse flag. A value of 0 in the ForwardDir argument represents reverse direction and 0x80 is used to indicate forwards movement.

```
/*
 * Work out the current speed percentage and direction and update each of the
 * servos with this data
 */
void notifyDccSpeed( uint16_t Addr, uint8_t Speed, uint8_t ForwardDir, uint8_t
SpeedSteps )
{
  /* Only respond to address 3 */
  if (Addr != 3)
    return;

  int percentage = ((Speed - 1) * 100) / SpeedSteps;
  servo1->setSpeed(percentage, ForwardDir != 0);
  servo2->setSpeed(percentage, ForwardDir != 0);
  servo3->setSpeed(percentage, ForwardDir != 0);
}
```

The notifyDccFunc routine was also replaced with one that would set servo1, servo2 and servo3 active or inactive based on the current setting of F0, F1 and F2 functions.

```
/*
 * Update the active status of each of the servos based on the functions
 * that are enabled.
 */
void notifyDccFunc( uint16_t Addr, uint8_t FuncNum, uint8_t FuncState)
{
  /* Respond only to address 3 */
  if (Addr != 3)
    return;

  if (FuncNum != 1)
    return;
  if (FuncState & 0x10)
    servo1->setActive(true);
  else
    servo1->setActive(false);
  if (FuncState & 0x01)
    servo2->setActive(true);
  else
    servo2->setActive(false);
  if (FuncState & 0x02)
    servo3->setActive(true);
  else
    servo3->setActive(false);
}
```

Finally we updated the loop function so that each of the servo object's loop methods are called in addition to the NmraDcc.process method.

```
void loop()
{
  Dcc.process();

  // Call the loop functions for each of the servos
  servo1->loop();
  servo2->loop();
  servo3->loop();
}
```

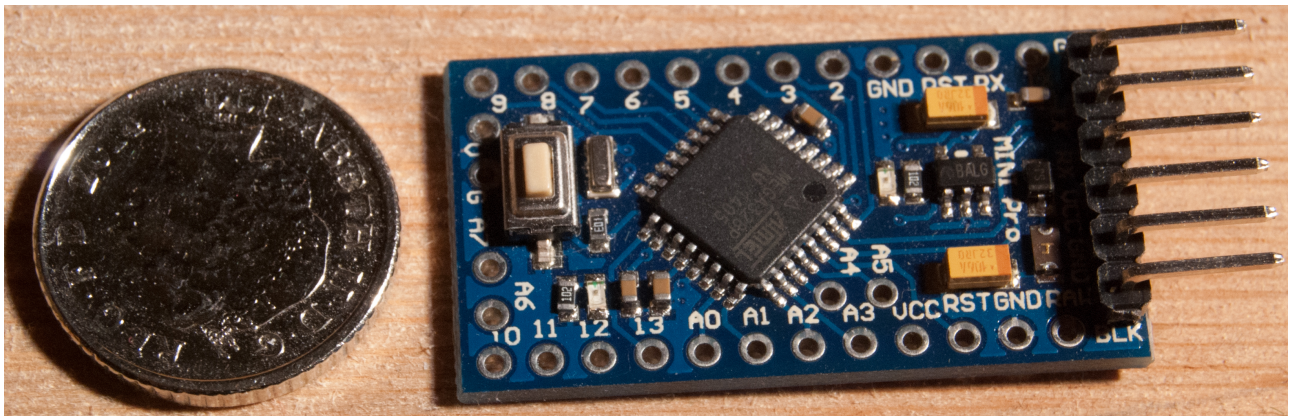
Putting all the pieces together, we had a DCC decoder that could drive the three servos between the compiled in angles, stop the servos at any angle between the end points for that servo and control which servos are moved when. What is missing to make this a usable servo decoder is the ability to set the end points, speed and address via the DCC system, i.e. programing via CV's.

Programming CVs

A number of steps are involved in making the decoder work with CV values. Firstly an internal storage structure is created and #define's written for all of the CV numbers that apply to this decoder.

```
 // CV Storage structure
struct CVPair
{
  uint16_t  CV;
  uint8_t   Value;
};

/*
 * The CVs that are used for servos
 */
#define CV_S0LIMIT0    10
#define CV_S0LIMIT1    11
#define CV_S0TRAVEL    12
#define CV_S1LIMIT0    13
#define CV_S1LIMIT1    14
#define CV_S1TRAVEL    15
#define CV_S2LIMIT0    20
#define CV_S2LIMIT1    21
#define CV_S2TRAVEL    22
```

The next step was to create a set of "factory defaults" for the CV, in this example we used the values we had previously hardwired into the setup routine.



Above: Arduino development boards do not have to be big. Here is an Arduino Mini clone pictured next to a 5 pence coin.

```
/*
 * The factory default CV values
 */
CVPair FactoryDefaultCVs [] =
{
  {CV_MULTIFUNCTION_EXTENDED_ADDRESS_LSB, 3},
  {CV_MULTIFUNCTION_EXTENDED_ADDRESS_MSB, 0},
  {CV_MULTIFUNCTION_PRIMARY_ADDRESS, 3},
  {CV_VERSION_ID, DCCSERVO_VERSION_ID},
  {CV_MANUFACTURER_ID, MAN_ID_DIY},
  {CV_S0LIMIT0, 10},
  {CV_S0LIMIT1, 80},
  {CV_S0TRAVEL, 20},
  {CV_S1LIMIT0, 30},
  {CV_S1LIMIT1, 110},
  {CV_S1TRAVEL, 10},
  {CV_S2LIMIT0, 30},
  {CV_S2LIMIT1, 110},
  {CV_S2TRAVEL, 10},
  {CV_29_CONFIG, 0},
};
```

The setup routine was modified so that the servo instances are created with the values in the CV's rather than the hardwired values. Also the flags passed in the Dcc init routine are changed to include FLAGS_MY_ADDRESS_ONLY, which means that the notify functions will now only be called for the DCC address set in our CV, either in CV1 or in the extended address in CV17 and CV18, the setting of CV29 defines if extended addressing is in use.

```
/*
 * Setup routine called on startup of the decoder
 */
void setup()
{
  Serial.begin(9600);

  // Configure the DCC CV Programing ACK pin for an output
  pinMode(DccAckPin, OUTPUT);
  digitalWrite(DccAckPin, LOW);

  // Setup which External Interrupt, the Pin it's associated with that
  // we're using and enable the Pull-Up
  Dcc.pin(0, DccInPin, 1);

  // Call the main DCC Init function to enable the DCC Receiver
  Dcc.init(MAN_ID_DIY, 10,
        FLAGS_MY_ADDRESS_ONLY|FLAGS_OUTPUT_ADDRESS_MODE, 0);


  // Create the instances of the servos, initialise the limits and travel
  // times from the CV values
  servo1 = new DCCServo(servoPin, Dcc.getCV(CV_S0LIMIT0),
                      Dcc.getCV(CV_S0LIMIT1), Dcc.getCV(CV_S0TRAVEL));
  servo2 = new DCCServo(servoPin + 1, Dcc.getCV(CV_S1LIMIT0),
                      Dcc.getCV(CV_S1LIMIT1), Dcc.getCV(CV_S1TRAVEL));
  servo3 = new DCCServo(servoPin + 2, Dcc.getCV(CV_S2LIMIT0),
                      Dcc.getCV(CV_S2LIMIT1), Dcc.getCV(CV_S2TRAVEL));
}
```

To add the mechanism to accept new CV values, there are two things required: a method to send the acknowledge to the command station and a routine that actions the changes to the CV values. These are both done using notify functions.

```
// This function is called by the NmraDcc library
// when a DCC ACK needs to be sent
// Calling this function should cause an increased 60ma current drain
// on the power supply for 6ms to ACK a CV Read
void notifyCVAck(void)
{
  digitalWrite( DccAckPin, HIGH );
  delay( 6 );
  digitalWrite( DccAckPin, LOW );
}
```

DCC is essentially a unidirectional system, with data being sent from the command station to the decoders, or at least it has been until recently with the advent of Railcom for transmitting data in the opposite direction. Decoder programing does not use Railcom, it uses the much more primitive mechanism by which the command station monitors a change in the current draw from the decoder to receive data from the decoder. The command station looks for an increase of at least 60mA for a period of 6ms to represent a single acknowledge bit.

Using the interface board that we built in the first session, this is achieved by turning on the DccAckPin that causes the transistor Q1 to turn on and current to flow through the 100ohm resistor. You may have wondered why when you program a loco decoder that the motor gets pulsed, this is simply the decoder sending acknowledge bits back for the programming sequence and the CV read back. Normal locomotive decoders can use the motor as a way to alter the current draw of the decoder. In our case, since the Arduino and servos are not powered from the DCC bus, we have to use the opto-isolator to drive track voltage through the resistor in order to increase the current draw on the DCC side of the isolator circuitry.

The NmraDcc library notifies us of changes to the values of CVs using the same notify function mechanism. The function notifyCVChange is called whenever the decoder is sent a new CV value by the command station. In our case we then call the appropriate function in the servo instance that is being updated. Changing CV12 will change the time it takes for the first servo to move between the two end points, hence it calls the setTravelTime for the servo1 object.

```
/*
 * Called to notify a CV value has been changed
 */
void notifyCVChange( uint16_t CV, uint8_t value)
{
  Dcc.setCV(CV, value);
  switch (CV)
  {
    case CV_S0LIMIT0:
      servo1->setStart(value);
      break;
    case CV_S0LIMIT1:
      servo1->setEnd(value);
      break;
    case CV_S0TRAVEL:
      servo1->setTravelTime(value);
      break;
    case CV_S1LIMIT0:
      servo2->setStart(value);
      break;
    case CV_S1LIMIT1:
      servo2->setEnd(value);
      break;
    case CV_S1TRAVEL:
      servo2->setTravelTime(value);
      break;
    case CV_S2LIMIT0:
      servo3->setStart(value);
      break;
    case CV_S2LIMIT1:
      servo3->setEnd(value);
      break;
    case CV_S2TRAVEL:
      servo3->setTravelTime(value);
      break;
    case CV_MULTIFUNCTION_PRIMARY_ADDRESS:
      MyAddress = value;
      break;
  }
}
```

When the NmraDcc library sees a factory reset event for the decoder, a write to CV8, it calls the special notify function, notifyCVResetFactoryDefault. This notify call cannot directly set the values for all the CVs, this must be done one CV at a time with the DCC objects process method being called.

```
/*
 * A factory reset is required. Called on first run if the NVRAM does not
 * contain valid CV values
 */
void notifyCVResetFactoryDefault()
{
  // Make FactoryDefaultCVIndex non-zero and equal to num CV's to be reset
  // to flag to the loop() function that a reset to Factory Defaults needs to be done
  FactoryDefaultCVIndex = sizeof(FactoryDefaultCVs)/sizeof(CVPair);
}
```

The variable FactoryDefaultCVIndex is set to the number of CVs that need to be set for the factory to complete. The loop routine is modified to call setCV on each of the CVs that need to be set and decrementing FactoryDefaultCVIndex on each call.

```
/*
 * Loop function, this is the main body of the code and is called repeatedly
 * in order to do whatever processing is required.
 */
void loop()
{
  // Execute the DCC process frequently in order to ensure
  // the DCC signal processing occurs
  Dcc.process();

  /* Check to see if the default CV values are required */
  if( FactoryDefaultCVIndex && Dcc.isSetCVReady())
  {
    FactoryDefaultCVIndex--; // Decrement first as initially it is the
                             // size of the array
    Dcc.setCV( FactoryDefaultCVs[FactoryDefaultCVIndex].CV,
          FactoryDefaultCVs[FactoryDefaultCVIndex].Value);
  }

  // Now call the loop method for every DCCServo instance
  servo1->loop();
  servo2->loop();
  servo3->loop();
}
```

With all these components in place we now had a DCC decoder that could not only drive the servos as described but which also let us set the decoder address, servo end points and speed via conventional CV programing, either on a programming track or using “programming on the main”.

Other Devices

As an extension to the workshop and because a number of people had purchased Arduino starter kits from ebay that included a stepper motor and shield to control it, I wrote another library, with a similar interface to the DCCServo library that would allow a stepper motor to be driven rather than a servo.

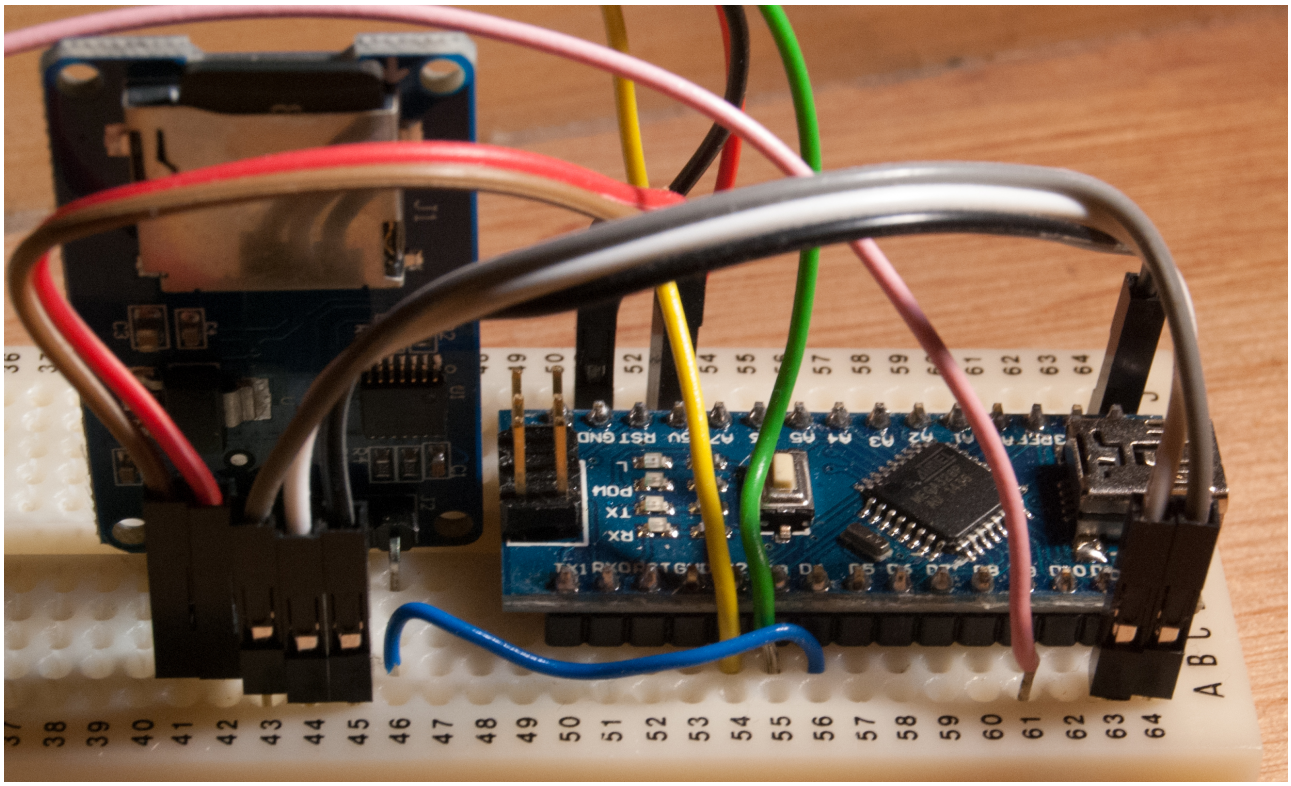
The public interface to the stepper motor library is shown below and is slightly simpler than that required for the servo. There is no start and end position, just a maximum speed defined in revolutions per minute.

```
class DCCStepper {
  private:
    ...
  public:
    DCCStepper(int, int, int, int, int, int);
    void loop();
    void setSpeed(int, boolean);
    void setActive(boolean);
    void setRPM(int);
};
```

In this case the constructor is passed the four pins to which the stepper motor shield was attached and a default value for the maximum RPM value of the stepper motor. To use a stepper motor rather than a servo, the setup routine needs to be altered to call the correct constructor, the notify routines for the speed and function buttons call the setSpeed in this new object and the stepper loop function is called rather than the servo loop function. It is actually a lot easier than is suggested. One member of the group very quickly had a variant of the decoder that could drive two servos and one stepper motor. Proving the versatility of the approach and the ease with which custom decoders could be created.

As an exercise to illustrate how versatile the approach is and how cheaply solutions can be put together I rounded the evening off by showing a prototype sound decoder based on the same

interface card, libraries and Arduinos. I found a library that could play WAV files from an SD card and a cheap microSD card adapter for use with the Arduino. In all I put together a very simple, low quality sound decoder that could be controlled via the DCC system with one evenings work and £10 worth of parts. Although not usable as it stood, with some more work and a few improvements on the audio side I think it could become a viable option for creating DCC controlled sound effects.



Above: A DCC Sound Decoder on a breadboard, on the left is a micro-SD card read, an Arduino Micro clone on the right. The speaker, amplifier and DCC interface are not shown.

All the code shown here is available freely online, my example code and libraries can be found on GitHub, https://github.com/M1118. The NmraDcc library is also available online, on the http://mrrwa.org website. I found this a useful exercise in understanding more about how DCC works, but more than that I can see this being a practical solution to controlling the likes of cranes and other animated objects on the layout.

Several different Arduino models have been used by the group, including a number of cheap clones purchased from ebay. Arduino mini and nano clones can be purchased for £2 or less and are small enough to consider installing within our models, at least for 4mm and larger scales, if you don't want to consider buying Arduino processors and making your own PCBs. I have been considering the possibility of creating a "DCC decoder core" with the interface for the DCC bus, power supplies for the Arduino and the Arduino chip itself on a single, probably surface mount board that could be used as a flexible platform for DIY decoder building.