# DCC++ EX User Manual

## What is DCC++ EX?

DCC++ EX is a Command Station and system for controlling model railroads. “DCC++ EX is designed to make this technology easily available an inexpensive. The goal of the DCC++ EX team is to make model railroading accessible to all.

The “DCC” stands for “Digital Command Control” and is a specification from the National Model Railroad Association (NMRA) for how to send signals to control model trains and accessories.

## What Does the “EX” Stand for?

In short, it stands for “EXtended”. We have EXtended and EXpanded on DCC++ and given you lots of EXtras. And we’re not EXaggerating ;)

## A Little History

The DCC++ EX team and the entire model railroad community owe a large debt of gratitude to a man named Gregg E. Berman. Greg had the idea to take a popular microcontroller board used in education and by hobbyists called an Arduino Uno and pair it with an Arduino Motor Controller Board and use it to create what he called a “Base Station” to control trains. Gregg called his creation DCC++. The “DCC++” name is a play on words since C++ is the computer programming language used to write the software that runs the command station. So putting DCC together with C++ became “DCC++”.

The genius of his idea was that he created a new use for things that were never designed to do what he made them do. The motor “shield” as it is called because it is one of many accessories that plug into the top of an Arduino board like shields, was designed to control the motors that drive the wheels of a robot. Thinking outside the box, he decided to try and see if instead, he make it generate a DCC signal that he could connect to his tracks and run his trains. The answer of course was yes, and the result of that experiment is what you are about to use to control your own model railroad.

Gregg stopped posting in the forums and updating the software and his excellent project became abandoned. That didn’t stop other industrious railroaders with programming skills from taking the open source (free to use and modify) program files and to try and fix issues add features to DCC++. However, there was no central group to organize these various efforts and what existed was a hodge podge of bits and pieces spread all over the internet.

A gentleman by the name of Fred Decker started working on the project himself with a dream of reviving it and making it into a standard that would continue to be developed and improved, and which would offer a complete system addressing the problems of cost and ease of use with other currently available systems. Part by his efforts and part by experiencing a lot of luck, he found other people who were interested in the project, and soon a team of more than twenty people from all over the world was put together. These volunteers built the DCC++ EX command station, invented the FireBox command station, the exWebThrottle and more. We hope to continue to develop solutions for the railroad community and pass the torch to other leaders who will, much like with a model railroad club, keep it going for the next generation.

If you are interested in this story and more about the history you hold in your hands, we encourage you to go to our website and read about it in the “A short history of DCC++”

## DCC++ EX Command Station

The DCC++ Base Station software is a C++ “sketch” (what Arduino calls a computer program) designed for the Ardunio Uno, Mega, Nano, Nano Every, and Teensy microcontrollers fitted with a Motor Controller like the Arduino Motor Shield. This sketch enables this combination to serve as a full-function DCC Command Station. With DCC++ EX, you can control DCC-equipped locomotives and DCC stationary decoders to run your turnouts and accessories.

For a complete overview of the system, as well as detailed instruction videos, please read our webpage: https://dcc-ex.github.io

DCC++ EX YouTube channel: <https://www.youtube.com/channel/UCJmvQx-fe0OMAIH-_g-_rZw>

## Command and Communication

Communication to and from the Command Station is through standard serial interface protocols via a USB cable connected to a Windows PC, Mac, or Linux system. A Bluetooth transceiver can optionally be connected to the Arduino’s serial pins to enable wireless serial communication with no modification to the sketch.

Commands and data sent to and from the DCC++ EX sketch are of the following form:

< C P1 P2 ... PN >

where C is a single alphanumeric command character and P1 through PN are optional alphanumeric parameters.

White space between the initial opening bracket < and the command character C is permitted but not required. White space between the command character C and the first parameter P1 (if there is one) is permitted but not required. White space between the last parameter PN (if there is one) and the closing bracket > is permitted but not required. White space between parameters (e.g. P1 and P2) IS required.

Characters sent to the sketch that are not enclosed in brackets are ignored. Only the first 20 characters after an opening bracket is received by the sketch are recorded and parsed. Any additional characters sent before the closing bracket is received by the sketch are ignored to prevent an inadvertent overflow of the parsing buffer. None of the current commands should require more than 20 total characters.

Users can manually send commands to, and receive data from the sketch using the Serial Monitor that is part of the Arduino IDE software. A generic Serial Window utility (ex: PUTTY) can also be used instead of the monitor that comes with the Arduino IDE. This sketch sets the baud rate of the Uno’s serial connection to its maximum value of 115200. You may change it to a lower value if desired but always ensure it matches the baud rate set for whatever program you use to communicate with the Uno.

Operating the sketch by manually typing commands into a serial window is a very good way of learning how the sketch works, understanding the data it sends back, testing new commands and parameters, and verifying proper connectivity to the tracks and trains. However, using the DCC++ Base Station in this fashion is not very practical for actually running a full model railroad.

Instead, the sketch should ideally be driven by a separate control/interface program that provides users with an intuitive GUI that constructs Base Station commands behind the scenes and transmits them to the Uno as needed. This same GUI should also process and act on any output sent back from the Base Station. DCC++ Controller, written in Java using the Processing IDE and graphics library (https://www.processing.org), is one such GUI. It is separately available under an open-source license as part of the overall DCC++ system.

## Installing and Using DCC++ Base Station

The entire DCC++ Base Station sketch is stored in single folder named DCCpp\_Uno, and consists of 6 C++ files, and 6 associated header (\*.h) file as follows:

DCCpp\_Uno.ino: Main entry point for the sketch. Declares global variables, configures timers and interrupts, and contains the required setup() and loop() subroutines.

SerialCommand.cpp: Parses and processes text commands received on the serial port.

CurrentMonitor.cpp: Measures the current drawn from both the main operations track and the programming track and monitors for short circuits.

Accessories.cpp: Allows for the optional creation of a list of turnouts controlled by stationary DCC decoders. The direction of turnouts specified in this list are updated and stored in the Uno’s EEPROM for retention after power is shut down.

Sensor.cpp: Allows for the optional creation of a list of train-detection sensors that are directly connected to unused pins on the Uno.

PacketRegister.cpp: Contains all the code that creates NMRA DCC bit packets based on the commands parsed by SerialCommand.cpp.

To open the sketch within the Arduino IDE, open the main entry point file DCCpp\_Uno.ino. The Arduino IDE will automatically open all the other files found in the same directory.

The DCC++ Base Station sketch has been fully tested with Arduino’s latest IDE version 1.6.5. Each of the 12 sketch files are heavily commented. In particular, SerialCommand.cpp contains complete descriptions of every DCC++ Base Station command, including all input parameters and return parameters. A summary of these commands (without listing the parameters) is as follows:

<t>: sets the throttle for a mobile engine decoder using 128-step speeds

<f>: controls mobile engine decoder functions F0-F28

<a>: controls stationary accessory decoders

<T>: controls turnouts connected to stationary accessory decoders

<w>: writes a configuration variable byte to an engine decoder on the main ops track

<b>: sets/clear a configuration variable bit in an engine decoder on the main operations track

<W>: writes a configuration variable byte to an engine decoder on the programming track

<B>: sets/clear a configuration variable bit in an engine decoder on the programming track

<R>: reads a configuration variable byte from an engine decoder on the programming track

<1>: turns on track power

<0>: turns off track power

<c>: reads current draw from main operations track

<s>: returns status messages, including power state, turnout states, and sketch version

## One Step at a Time

If you don’t have an Arduino Uno, you can still download the Arduino IDE for free from

<http://arduino.cc>

and open/explore the DCC++ Base Station sketch code and read through the comments. You can even compile the sketch, though without an Uno you cannot load it, and therefore cannot run it.

If you have an Uno but do not have an Arduino Motor Shield you can still load and run the software on the Uno, and test out the serial connectivity and command data flow, but you cannot connect the Uno directly to the tracks of a model railroad.

If you have an Uno AND a Motor Shield, but do not have a power supply, go buy a power supply, at which point you will then have a complete DCC++ Base Station ready to connect to a model railroad. As mentioned above, you’ll probably want to download and modify the DCC++ Controller GUI for your own use, since manually typing commands into the Serial Monitor of the Arduino IDE gets tiring very fast.

Enjoy!

Arduino DCC++

|  |
| --- |
| DCC++ BASE STATION is a C++ program written for the Arduino Uno and Arduino Mega |
|  | using the Arduino IDE 1.6.6. |
|  |  |
|  | It allows a standard Arduino Uno or Mega with an Arduino Motor Shield (as well as others) |
|  | to be used as a fully-functioning digital command and control (DCC) base station |
|  | for controlling model train layouts that conform to current National Model |
|  | Railroad Association (NMRA) DCC standards. |
|  |  |
|  | This version of DCC++ BASE STATION supports: |
|  |  |
|  | \* 2-byte and 4-byte locomotive addressing |
|  | \* Simultaneous control of multiple locomotives |
|  | \* 128-step speed throttling |
|  | \* Cab functions F0-F28 |
|  | \* Activate/de-activate accessory functions using 512 addresses, each with 4 sub-addresses |
|  | - includes optional functionailty to monitor and store of the direction of any connected turnouts |
|  | \* Programming on the Main Operations Track |
|  | - write configuration variable bytes |
|  | - set/clear specific configuration variable bits |
|  | \* Programming on the Programming Track |
|  | - write configuration variable bytes |
|  | - set/clear specific configuration variable bits |
|  | - read configuration variable bytes |
|  |  |
|  | DCC++ BASE STATION is controlled with simple text commands received via |
|  | the Arduino's serial interface. Users can type these commands directly |
|  | into the Arduino IDE Serial Monitor, or can send such commands from another |
|  | device or computer program. |
|  |  |
|  | When compiled for the Arduino Mega, an Ethernet Shield can be used for network |
|  | communications instead of using serial communications. |
|  |  |
|  | DCC++ CONTROLLER, available separately under a similar open-source |
|  | license, is a Java program written using the Processing library and Processing IDE |
|  | that provides a complete and configurable graphic interface to control model train layouts |
|  | via the DCC++ BASE STATION. |
|  |  |
|  | With the exception of a standard 15V power supply that can be purchased in |
|  | any electronics store, no additional hardware is required. |
|  |  |
|  | Neither DCC++ BASE STATION nor DCC++ CONTROLLER use any known proprietary or |
|  | commercial hardware, software, interfaces, specifications, or methods related |
|  | to the control of model trains using NMRA DCC standards. Both programs are wholly |
|  | original, developed by the author, and are not derived from any known commercial, |
|  | free, or open-source model railroad control packages by any other parties. |
|  |  |
|  | However, DCC++ BASE STATION and DCC++ CONTROLLER do heavily rely on the IDEs and |
|  | embedded libraries associated with Arduino and Processing. Tremendous thanks to those |
|  | responsible for these terrific open-source initiatives that enable programs like |
|  | DCC++ to be developed and distributed in the same fashion. |
|  |  |
|  | REFERENCES: |
|  |  |
|  | NMRA DCC Standards: http://www.nmra.org/index-nmra-standards-and-recommended-practices |
|  | Arduino: http://www.arduino.cc/ |
|  | Processing: http://processing.org/ |
|  | GNU General Public License: http://opensource.org/licenses/GPL-3.0 |
|  |  |
|  | BRIEF NOTES ON THE THEORY AND OPERATION OF DCC++ BASE STATION: |
|  |  |
|  | DCC++ BASE STATION for the Uno configures the OC0B interrupt pin associated with Timer 0, |
|  | and the OC1B interupt pin associated with Timer 1, to generate separate 0-5V |
|  | unipolar signals that each properly encode zero and one bits conforming with |
|  | DCC timing standards. When compiled for the Mega, DCC++ BASE STATION uses OC3B instead of OC0B. |
|  |  |
|  | Series of DCC bit streams are bundled into Packets that each form the basis of |
|  | a standard DCC instruction. Packets are stored in Packet Registers that contain |
|  | methods for updating and queuing according to text commands sent by the user |
|  | (or another program) over the serial interface. There is one set of registers that controls |
|  | the main operations track and one that controls the programming track. |
|  |  |
|  | For the main operations track, packets to store cab throttle settings are stored in |
|  | registers numbered 1 through MAX\_MAIN\_REGISTERS (as defined in DCCpp.h). |
|  | It is generally considered good practice to continuously send throttle control packets |
|  | to every cab so that if an engine should momentarily lose electrical connectivity with the tracks, |
|  | it will very quickly receive another throttle control signal as soon as connectivity is |
|  | restored (such as when a trin passes over rough portion of track or the frog of a turnout). |
|  |  |
|  | DCC++ Base Station therefore sequentially loops through each main operations track packet regsiter |
|  | that has been loaded with a throttle control setting for a given cab. For each register, it |
|  | transmits the appropriate DCC packet bits to the track, then moves onto the next register |
|  | without any pausing to ensure continuous bi-polar power is being provided to the tracks. |
|  | Updates to the throttle setting stored in any given packet register are done in a double-buffered |
|  | fashion and the sequencer is pointed to that register immediately after being changes so that updated DCC bits |
|  | can be transmitted to the appropriate cab without delay or any interruption in the bi-polar power signal. |
|  | The cabs identified in each stored throttle setting should be unique across registers. If two registers |
|  | contain throttle setting for the same cab, the throttle in the engine will oscillate between the two, |
|  | which is probably not a desireable outcome. |
|  |  |
|  | For both the main operations track and the programming track there is also a special packet register with id=0 |
|  | that is used to store all other DCC packets that do not require continious transmittal to the tracks. |
|  | This includes DCC packets to control decoder functions, set accessory decoders, and read and write Configuration Variables. |
|  | It is common practice that transmittal of these one-time packets is usually repeated a few times to ensure |
|  | proper receipt by the receiving decoder. DCC decoders are designed to listen for repeats of the same packet |
|  | and provided there are no other packets received in between the repeats, the DCC decoder will not repeat the action itself. |
|  | Some DCC decoders actually require receipt of sequential multiple identical one-time packets as a way of |
|  | verifying proper transmittal before acting on the instructions contained in those packets |
|  |  |
|  | An Arduino Motor Shield (or similar), powered by a standard 15V DC power supply and attached |
|  | on top of the Arduino Uno or Mega, is used to transform the 0-5V DCC logic signals |
|  | produced by the Uno's Timer interrupts into proper 0-15V bi-polar DCC signals. |
|  |  |
|  | This is accomplished on the Uno by using one small jumper wire to connect the Uno's OC1B output (pin 10) |
|  | to the Motor Shield's DIRECTION A input (pin 12), and another small jumper wire to connect |
|  | the Uno's OC0B output (pin 5) to the Motor Shield's DIRECTION B input (pin 13). |
|  |  |
|  | For the Mega, the OC1B output is produced directly on pin 12, so no jumper is needed to connect to the |
|  | Motor Shield's DIRECTION A input. However, one small jumper wire is needed to connect the Mega's OC3B output (pin 2) |
|  | to the Motor Shield's DIRECTION B input (pin 13). |
|  |  |
|  | Other Motor Shields may require different sets of jumper or configurations (see Config.h and DCCpp.h for details). |
|  |  |
|  | When configured as such, the CHANNEL A and CHANNEL B outputs of the Motor Shield may be |
|  | connected directly to the tracks. This software assumes CHANNEL A is connected |
|  | to the Main Operations Track, and CHANNEL B is connected to the Programming Track. |
|  |  |
|  | DCC++ BASE STATION in split into multiple modules, each with its own header file: |
|  |  |
|  | DCCpp: declares required global objects and contains initial Arduino setup() |
|  | and Arduino loop() functions, as well as interrput code for OC0B and OC1B. |
|  | Also includes declarations of optional array of Turn-Outs and optional array of Sensors |
|  |  |
|  | SerialCommand: contains methods to read and interpret text commands from the serial line, |
|  | process those instructions, and, if necessary call appropriate Packet RegisterList methods |
|  | to update either the Main Track or Programming Track Packet Registers |
|  |  |
|  | PacketRegister: contains methods to load, store, and update Packet Registers with DCC instructions |
|  |  |
|  | CurrentMonitor: contains methods to separately monitor and report the current drawn from CHANNEL A and |
|  | CHANNEL B of the Arduino Motor Shield's, and shut down power if a short-circuit overload |
|  | is detected |
|  |  |
|  | Accessories: contains methods to operate and store the status of any optionally-defined turnouts controlled |
|  | by a DCC stationary accessory decoder. |
|  |  |
|  | Sensor: contains methods to monitor and report on the status of optionally-defined infrared |
|  | sensors embedded in the Main Track and connected to various pins on the Arudino Uno |
|  |  |
|  | Outputs: contains methods to configure one or more Arduino pins as an output for your own custom use |
|  |  |
|  | EEStore: contains methods to store, update, and load various DCC settings and status |
|  | (e.g. the states of all defined turnouts) in the EEPROM for recall after power-up |
|  |  |
|  | DCC++ BASE STATION is configured through the Config.h file that contains all user-definable parameters |
|  |  |
|  | \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |

XXX (move this to a history doc) like Mike Dunston who was a valuable mentor, Chris XXX who had his own genius inspiration for a way to completely re-think how the code in the Command Station could work, Harald Baue, a top notch programmer and electronics technician who had been his own idea, and David XXX, a brilliant young man who

## WiFi

## Ethernet

From Haba: If you want to do ethernet you should dig out the id of the Arduino and use that as the last octetts of the MAC addr. Classic had #define MAC\_ADDRESS { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xEF }. The user needs to know that if they have more than one CS in their setup, they will have to give them each a unique MAC address.

because these cheapo chips do not seem to have a burned in mac addr.

## Engine Driver

## WiThrottle

No... the withrotyle to cs is there for people to run completely witout jmri.

Ed to withrottle to jmri to cs keeps everything coordinated... but cs is not there to coordinate withrottle and jmri.

.. but it will coordinate multiple withrottles that connect directly

The withrottle server in cs works fine with commands on the serial as well... but if you issue throttle or function commands cs updates the reminders... withttles poll to get this but JMRI doesnt.

We.. we coukd shove <t ...> responses out the serial but thres no function equivalent and it probably wouldnt work if jmri is ethernet or wifi... to me it seems dumb to have two separate systems driving the same loco... thats why withrottle drives THROUGH jmri rather than round it.

My wife tells me where to drive all the time but she doesnt have a separate steering wheel

## Short Circuit / Overlimit Protection

## CV Read, Write, and Ack Detection

This section will cover reading and writing CVs on the service (programming) track. For information on programming on the main track, see \*\*TODO: Add Link\*\*

NOTE: In order to use the service track, you must have a current sense signal from your motor controller connected to the CS, or have an external current sense board properly wired. The motor board configuration in your config.h file must also have the correct value for current sense factor.

### Quick Definitions

CV – Configuration Variable. This is simply a memory location in your decoder that holds your settings. Each CV can hold one or several pieces of information. You need to consult your decoder manual for a list of CVs it supports.

Read a CV – Polls the CV memory location and waits for an acknowledgement from the decoder to verify its setting

Write a CV – Writes and then verifies a value into a CV to store or change a setting

ACK – ACK stands for “acknowledgement” and is sent by the decoder in response to read and write command. An ACK from an NMRA compliant decoder is nominally 60 milliAmps for 6 milliseconds.

Current Sense – A circuit supplied by a motor shield or motor driver board or external current sense board that senses the output current to the track. This is required for service track operations to work.

Current Sense Factor – A value supplied in the motor board definition from your config.h file that is calibrated for the specification of your motor board type and tells the CS how to read the current it is reporting.

Current Limit – A limit you set at a point below the maximum current the board can deliver that will trigger the overlimit warning and shut down the track.

Maximum Current – The maximum current the board is capable of handling

Overlimit – The condition reported when the current to the tracks exceeds your pre-determined “Current Limit”. You may have a 5A board but never want it to go above 3A.

Short Circuit – If a piece of equipment fails or a piece of metal falls across the track and creates a “short”, this triggers an overlimit warning to the serial monitor and the optional display and turns off power to the track.

Packet – A set of bytes representing a command sent to the track.

### Overview

While on the service track, your controller (ie: JMRI, a throttle with CV writing capability, or even a serial monitor connected to the CS) can send commands that change settings in your decoder. You can change the address of your loco, speed settings, sounds on a sound decoder, and more.

NOTE: When your loco is on the service track, it will only respond to programming commands, it will not be able to respond to throttle or function commands. This is by design to protect your loco and part of the NMRA standard. If you want your loco to move, you have to place it on the Main Track. However we do have a special method to be able to test your programming commands on the same track. For information see the JOIN command.

### Things to know about the service track

* For safety, it is limited to 250mA of current no matter what the main track or your motor driver can deliver. For decoders that draw too much current, this can be increased. TODO, link to section.
* You loco won’t respond to throttle commands, but as a result of the way it generates an ACK (sending a pulse of current through the motor), the loco may jump or inch along a bit each time. This is normal.

DCC++ EX is much faster than most other Command Stations and an order of magnitude faster and more accurate than DCC++. Reading a page of CVs from DCC++ in our testing takes an average of XX seconds. The same page of CVs read in DCC++ EX is XX seconds.

### How it Works

First, it may help to know how reading and writing works. Each CV holds a “byte” of data, which is composed of 8 “bits”. Each of those bits are like a light switch and is either “on” or “off”. On is a 1 and off is a 0. Each of those 8 bits can hold a separate setting, like the special CV29 does, or it can represent a number from 0 to 255. This will be important in a minute.

Because DCC on the main track is a one-way communication system (we can only send commands to the track, the loco can’t send the same kind of data back), there needs to be a way to know if the loco got a write command or to be able to “read” the contents of a CV. The way this is done is through an “ACK” signal on an isolated service track. The decoder in the loco sends a pulse of current to the track, that we can read as a “yes”. If you have ever played the guessing game called “20 Questions”, that is how the system works. When reading from the decoder, we can only ask it yes or no questions; “Is CV20 set to 1?”, and then we must wait for a certain period of time for a response. “Not very efficient”, you say? Correct! But keep reading.

If we had to start at 0 and work our way up to 255 asking the CV, “are you a 0? Are you a 1? Are you a 2?...”, that would be a pretty slow process if a lot of CVs were set to 255. We could have 256 guesses before we got the ACK to let us know we got the correct answer. Luckily, we have a few tricks we can use.

First, DCC++EX reads each bit, not the numerical value. We can put the bits together with math to know the number value they represent. That means that instead of a maximum of 256 guesses, our worst case is only 8. Right there is a 32X speed increase.

Next, we don’t wait for the specified number of packets to be sent to the decoder (up to 16 packets comprising our command data packet, repetitions of that packet and a few other packets) before we look for the ACK. We look for an ACK in between each packet and immediately stop sending packets and report our results. This is new to DCC++EX.

DCC++EX uses its Track Sensetm to know what it is reading and guess the probable value of the contents of a CV before it reads it. It is much faster to ask “Is CV 115 set to 1” if we know that most of the time that is the value in that CV than it is to read each bit in the CV

More accurate

Current sensors read a current and report it as a voltage that is proportional to that current. So a motor driver that can output 5 Amps, might represent that as 0 to 5 Volts that we measure on an Analog pin of the Arduino. Just before reading or writing to your CVs, we take a baseline reading of what current is being reported with the power off. This gives us the value your Motor Shield or Current Sense Board is reporting with no current. This is important because some current sense boards report positive and negative current. So zero current is halfway between the minimum and maximum values. In the case of a 5V range, this kind of sensor would report 2.5V for 0 Amps of current. Taking the quick reading allows the CS to automatically adjust to both kinds of current sensors and to calibrate its readings for your particular conditions at that particular moment. You still need to have the correct value for current sense factor in the MOTOR\_BOARD\_TYPE set in your config.h file.

Diagnosing Decoder Issues

<D ACK ON>

Increasing the current limit on the programming track

### Overlimit and Short Circuits

Retries in a pattern. Speed of detection.

Maybe the code should spit out a warning if offset+limit > 1000 because that would mean no short detection is possible.

You can add LCD(n,F("XXX")); calls to mysetup.h, where n is 0 to 7 and XXX is your long string. It will be truncated unless you also #define MAX\_MSG\_SIZE 40 in config.h.

### Startup Calibration for Current Sense

Mention bi-directional current and floating pins that will give bad readings