PROGRAMMING THE MIDI SOLUTIONS™

EVENT PROCESSOR
AND
EVENT PROCESSOR PLUS

A GUIDE TO ALL THE THINGS MIDI DIDN'T PLAN ON

by Bruce Wahler

Ashby Solutions™

v1.8

TABLE OF CONTENTS

INTRODUCTION 4

WHAT CAN THE EVENT PROCESSOR DO? 4

HOW THE GUIDE IS ARRANGED 5

BEFORE WE MOVE ON … 5

PROGRAMMING DEFINITIONS AND BASIC CONCEPTS 5

MIDI DATA TYPES 6

SYSTEM EXCLUSIVE (SYSEX) 7

REGISTERED PARAMETER NUMBER (RPN) 7

NON-REGISTERED PARAMETER NUMBER (NRPN) 8

CHAPTER 1 – BASIC PROGRAMMING 9

FILTERING 9

MAPPING 10

TRIGGERING EVENTS 11

DEFINING SEQUENCES 13

TURNING A SETTING ON AND OFF 15

CHAPTER 2 – ADVANCED PROGRAMMING 17

VARIABLES 17

ADDING FEATURES BY ENABLING AND DISABLING SETTINGS – PART II 20

THE 'ALL' PARAMETER 22

REVERSE FILTERING 22

SELECTING RANGES AND SCALING VALUES 24

INCREASING THE CAPABILITIES OF YOUR GEAR 25

THE SUPER-FOOT-CONTROLLER 26

SUPER-CONTROLLER #2 29

VELOCITY SENSITIVITY FROM AN ORGAN 30

DELAYS AND OTHER 'HARMLESS' MIDI STRINGS 32

THE SELECTIVE SWITCH 33

CHAPTER 3 – LIMITATIONS AND PITFALLS 36

MIDI BANDWIDTH 36

MIDI OVERFLOW 37

DUPLICATE SETTINGS 37

SERIAL VS. PARALLEL PROCESSING 37

EVENT PROCESSING ORDER 38

MISMATCHED VARIABLES AND VALUES 39

'LOSING' DATA IN THE TRANSLATION 39
NOTE NUMBERING VS. NOTE NAMING 39
SCALING AND ROUND-OFF ERRORS 40
BUTTONS THAT DON'T SEND MIDI 41
POWERING THE EVENT PROCESSOR 41
RECOVERING THE PROGRAMMING DATA 41

CLOSING THOUGHTS 42

INTRODUCTION

The Musical Instrument Digital Interface (MIDI) specification opened up a new world of flexibility in electronic instruments, especially in keyboards. MIDI allows a collection of musical devices to communicate with each other, and eliminates the need to drag along a large collection of bulky keyboards to gigs – a player only has two hands; why bring more than two keyboards?

Well, that's how MIDI is *supposed* to work. As in most things, there are differences between the concept and practice of MIDI, and things don't always work out as expected. Different musical products have varying degrees of MIDI support, and some manufacturers have interpreted the specifications differently that others. Even if two MIDI devices claim to support a feature (ex: keyboard velocity), there may be problems linking the two devices together.

MIDI Solutions' little "black boxes" have always provided easy answers to tricky MIDI problems, but the Event Processor™, and its big brother, the Event Processor Plus™ take this ability to a new level. With this new ability comes new complexity, and some users may find the programming capabilities of these boxes a little overwhelming at first. This guide attempts to remove some of the mystery of the Event Processor and Event Processor Plus, allowing users to program these boxes to their full capabilities.

NOTE: In the remaining passages of the guide, we'll refer to both the Event Processor and Event Processor Plus as simply the "Event Processor." In cases where the Event Processor Plus provides additional capabilities, we'll refer to it as the "Plus."

WHAT CAN THE EVENT PROCESSOR DO?

The Event Processor has ten (10) Settings, while the Event Processor Plus has 32 Settings. Each Setting can perform a number of tasks on the MIDI data:

- It can ***filter*** specific types of MIDI events to prevent other devices from seeing them;
- It can ***map*** (convert) one MIDI event into a different event;
- It can ***trigger*** a new event (or series of events) based on an incoming MIDI message;
- It can ***sequence*** through a series of events each time an incoming message is received;
- It can ***enable or disable*** some of its settings based on the events it sees;
- It can ***store data*** values and operating settings for later use.

Each of these features can be used in both simple and complex ways, as we will see in later sections.

HOW THE GUIDE IS ARRANGED

This programming guide contains a series of examples that show how the Event Processor can be used to solve real-world problems on MIDI equipment. Generally, the examples are practical ones based on real MIDI products, and some readers may find that their problems are directly solved by one or more of the examples.

Chapter 1 describes using the features of the Event Processor to solve simple, but often annoying, problems in MIDI gear. All owners of the Event Processor will want to read through this section, regardless of their level of MIDI knowledge.

Chapter 2 moves on to more complicated MIDI setups, and how the Event Processor can use several settings together to solve more complicated problems. Advanced users and MIDI "experts" will want to read through this section, too, as it opens up a new level of capabilities in the Event Processor. Users who are less skilled at the finer points of MIDI may want to browse this section to understand what the devices can do, and return at a later time to attempt complex programming, or arrange for programming help through MIDI Solutions or your dealer.

Chapter 3 will discuss problems and limitations caused by the MIDI data itself, or by the way the Event Processor handles the data. These are important points to keep in mind, especially when programming complex tasks in the Event Processor. The Event Processor may seem to be a "MIDI magician," but it has to obey a set of rules, too.

BEFORE WE MOVE ON …

In order to program the Event Processor, the user must have a general technical understanding of how MIDI works. A complete explanation of the MIDI protocol is beyond the scope of this guide. Users who are new to the world of MIDI are encouraged to read one or more of the available books on MIDI before attempting to program the Event Processor. Some of these books include:

- *MIDI for Musicians* by Craig Anderton; Music Sales Corporation, ISBN: 0825610508
- *Basic MIDI* by Paul White; Sanctuary Publishing, Ltd, ISBN: 1860742629
- *MIDI for the Professional* by Paul Lehrman, Tim Tully; Music Sales Corporation, ISBN: 0825613744
- *MIDI for the Technophobe* by Paul White; Sanctuary Press (UK), ISBN: 1860744443
- *MIDI Power!* by Robert Guérin; Muska & Lipman/Premier-Trade, ISBN: 1929685661
- *The MIDI Manual, Second Edition* by David Miles Huber; Focal Press, ISBN: 0240803302

PROGRAMMING DEFINITIONS AND BASIC CONCEPTS

The MIDI Solutions Event Processor Programming Tools use terminology designed around the needs of the MIDI protocol, and the Event Processor itself. Some of the most important ones to become familiar with are:

- **Events** – a MIDI command being sent is considered a MIDI 'event.' Events include notes being played, knobs being turned, and clock information being sent, as well as many other items. Think of an event as being the sending of a single type of MIDI information that we want to act upon.

- **Setting** – Each 'Setting' is a single task that the Event Processor can perform. In many cases, a Setting can also be thought of as a feature or function that we want the Event Processor to provide, like mapping a controller value or changing the MIDI channel of a message. Sometimes, though, it takes two or more Settings to produce the desired results. (This will become clearer as we continue in the guide.)

- **Value X** – a temporary value used to represent a range of values that a data byte in a MIDI command might send. Many times, while we don't know what the exact value of the data will be in a specific message, we want to pass the data on, either directly or after modifying the message. The term 'Value' allows us to keep track of the data and instruct the Event Processor to use the data in its Settings.

  Most MIDI commands are made up of two or more bytes; typically, either two or three bytes. The first byte is the 'Command Byte,' which tells what type of information is being sent (Note-On, Controller Change, etc.). The remaining bytes are the 'data' bytes – except in the case of System Exclusive (SysEx) data, which we'll discuss briefly in the next section. In the Event Processor we'll call the two typical data bytes 'Value X' and 'Value Y'. Generally, Value X refers a range of values in the first of the two data bytes, or in the only data byte in the command.

  The actual information in Value X changes between MIDI commands. In a Note-On command, for example, Value X tells us which note is being played; in a Control Change command, it tells us which controller is being adjusted. Thus, we need to call them something to keep track of the information that they contain.

  [Please note: The concept of Values is a necessary part of the flexibility of the device, and something that users will either grasp immediately, or will just have to accept as part of the programming process. Like your mother told you, you shouldn't be afraid to try new things!]

- **Value Y** – another temporary value used to represent a range of values in the MIDI command. Generally, Value Y refers to the second of two data bytes. Like Value X, Value Y information varies between MIDI commands; in our Note-On example, Value Y is the velocity of the note being played; in the Control Change example, Value Y is the value (knob position, etc.) of the control.

MIDI DATA TYPES

The Event Processor can work on all of the various types of MIDI data. The basic data types include:

- **Note On** – As expected, this is the command that tells a sound module to play a note. The first byte, called the 'Command Byte,' tells us that this is a note message, and the MIDI channel being used. The next two bytes tell the note number (0-127) and the velocity (0-127). Devices which do not send velocity are free to send any value in the third byte except zero. In the Event Processor, the note number is treated as Value X, and the velocity is treated as Value Y.

- **Note Off** – The counterpart to Note On, this command is rarely used; instead, a Note On with velocity of zero is used to show keys being released. Devices which support Note Off send the note number (0-127) and 'release velocity' (0-127). In the Event Processor, the note number is Value X, and the release velocity is Value Y.

- **Polyphonic Pressure** – This rarely-used feature of MIDI provides individual key pressure (Aftertouch) information for every key in a MIDI keyboards. Devices which support Polyphonic Pressure send the note number (0-127) and pressure amount (0-127). In the Event Processor, the note number is Value X, and the pressure amount is Value Y.

- **Continuous Controller (CC)** – This command describes the movement of the knobs, switches, pedals, etc. that a MIDI device uses to make adjustments during performance. This is one of the most common MIDI commands that we will want to use in the Event Processor, and despite its name, often represents controls (like switches) that don't provide a continuous range of settings; instead, they use 0 for off and 127 for on. Its two data bytes tell us the controller number (0-127) and the controller position (0-127). In the Event Processor, the controller number is Value X and the position is Value Y.

- **Program Change** – This command calls up a stored program on the device. It has only one data byte which provides the program number (0-127). In the Event Processor, the program number is Value X.

- **Channel Pressure or Aftertouch** – Most MIDI keyboards that support Aftertouch use this method, which sends one key pressure value for the entire keyboard. It also has only one data byte which provides the pressure amount (0-127). In the Event Processor, the pressure is Value X.

- **Pitch Bend** – This message defines the position of the Pitch Bender, which is usually a wheel, ribbon, or lever that automatically returns to its middle position. The command is somewhat unusual in that it both allows for a greater range of values than 0-127, and positive (up) and negative (down) bends. It has two data bytes which are used together to provide the bend value. In the Event Processor, the first data byte is Value X and the second data byte is Value Y.

- **System Real Time** – These messages include things like Clock, Start, Stop, System Reset, and Active Sensing. All of these messages have no MIDI channel, but are sent to all devices at once.

There are also some special data types that we need to address. Because these data types are more complicated, we'll spend a little extra time discussing them:

System Exclusive (SysEx)

This data format was developed by the originators of the MIDI specification to allow manufacturers to send model-specific data that wasn't planned for in the basic MIDI architecture. For the most part, these messages are used for less common MIDI tasks like transferring data to and from devices, or selecting special features that cannot be controlled through normal MIDI commands. SysEx commands have no associated MIDI channel; any device on the line is supposed to monitor for SysEx messages, and act on them as necessary. If the device doesn't act on the message, it's supposed to sit quietly until the message ends.

SysEx messages have a unique format, and are usually expressed in Hexadecimal (base 16), rather than Decimal (base 10): The first byte is always F0 (240), followed by several bytes to show the device's manufacturer or brand, and usually, the model of the device itself. The last byte is called the 'EOF' byte, and is always F7 (247). The following is an example of a SysEx message:

F0	00 00 3F 22	01 77	00 10 33 41 5A 00 05 54 32 01 22 5F	F7
Start	Manufacturer ID	Device ID	Message	EOF

In between the starting byte/ID and the EOF, all of the other bytes are undefined by MIDI, and can be whatever the manufacturer chooses, *but each byte cannot have a value greater than 7Fh (127).* SysEx messages can be very long, often thousands of bytes in length. SysEx isn't really intended for device-to-device communications, as much as for device-to-programmer or device-to-manufacturer needs. SysEx is also intended for infrequent communications, because it talks to every device, on every channel.

Registered Parameter Number (RPN)

As the use of MIDI expanded, some manufacturers started to work around the limitations of MIDI by sending all their information, even simple messages like knob and switch changes, through SysEx. This defeats the purpose of MIDI, which is to provide a simple, universal communications standard for *all* musical devices to use. It also slows down general MIDI communications by forcing everyone to pay attention to the messages. Two new data standards, RPN and NRPN, were developed to enhance MIDI control abilities without resorting to SysEx. Both standards start by using Control Change (CC) messages to tell the MIDI 'world" that they want to communicate outside the limits of standard MIDI.

RPN is the more structured of the two standards. If I were to decide that toasting bagels was a feature that MIDI instruments needed to address, and the MIDI Manufacturers Association agreed that my this was in the best interest of the *general* MIDI community (not likely!), they would 'register' one of the roughly 16,000 RPN

locations as being used for – and only for – toasting bagels. Certain limits would also be imposed on the data being sent by this new RPN, so that everyone would understand the messages. Some RPN examples are Master Tuning and Pitch Bend Range. For obvious reasons, there have been few RPN controls established: most MIDI needs can be met by other controllers of MIDI, and most of the remaining items tend to be very specific to one or two devices.

In theory, at least, no one is supposed to use any of the unregistered RPN locations – ever.

RPN uses the data byte of two MIDI controller locations, CC#100 and CC#101, to call out a specific RPN. At that point, changes to the RPN use the Data Slider (CC#6). In shorthand, a typical RPN command might look like this:

CC#100 = 40h (64)

CC#101 = 22h (34)

CC#6 = 127

This set of three CC messages would set the value of the RPN defined at 2022 in Hexadecimal – 7-bit math is in use here, because the highest bit of a data byte cannot be a '1' – as having a value of 127. Once an RPN is called out, additional messages can be sent by changing CC#6 only, since control of the Data Slider is now in the hands of the RPN. However, if someone else calls up a different RPN, the three-byte message must be sent again to reclaim the Data Slider.

To be safe and simple, it's best to send the three-byte version of the RPN each time it's used; however, this method triples the amount of data being sent with each RPN change. When programming the Event Processor to work with RPN controls, the user will have to consider tradeoffs between simplicity and data size. If the setup only uses one or RPN controls, it may be sufficient to only send the full three-command sequence at the beginning of use, or after the Event Processor detects a different RPN message header.

NON-REGISTERED PARAMETER NUMBER (NRPN)

NRPN is the 'back door' approach for those of us bagel toasters can't convince the mainstream MIDI community to adopt our ways and decide to implement the function, anyway. Each manufacturer can use NRPN to define commands that are unique to a particular product, and my bagel toasting NRPN is actually free to conflict with your NRPN to put toothpaste on a toothbrush. Since there are roughly 16,000 NRPN locations, it's possible to stay out of the way of others most of the time, but NRPN conflicts can and do happen. To prevent this problem, devices often send an initialization message, or a series of them, at power-up to establish their NRPN "turf."

Like RPN, NRPN uses the data byte of two MIDI controller locations, CC#98 and CC#99, to call out a specific NRPN. At that point, changes to the NRPN use the Data Slider. A typical RPN command might look like this:

CC#98 = 20h (32)

CC#99 = 00h (0)

CC#6 = 100

This set of three CC messages would set the value of the RPN defined at 1000 in Hexadecimal as having a value of 100. Once an NRPN is called out, additional messages can be sent by changing CC#6 only, but again, this is only true if no other NRPN message has been sent in between. Like RPN, there is a tradeoff between safety/simplicity and generating extra data that must be made.

Please also note that both RPN and NRPN controls use the Data Slider, so an RPN or NRPN controller must check for both kinds of messages before assuming that the Data Slider is still 'owned' by that RPN or NRPN.

CHAPTER 1 – BASIC PROGRAMMING

The concepts and examples used in this section show the basic capabilities of the Event Processor. Many of these concepts will be new to novice MIDI users, and will be explained in detail as we go along.

FILTERING

Filtering is a very basic feature of many MIDI processors. Sometimes, a certain MIDI event or group of events can cause a MIDI device to do unwanted or erratic things, and the only solution is to prevent the event from reaching the device by removing it. A good analogy is the oil filter in an automobile engine: the filter allows certain things to pass through (the oil), but prevents others (dirt, debris). The Event Processor can be programmed to remove certain events from the data it receives.

A very useful example of this is MIDI *channel pressure* or *aftertouch.* This feature is provided on many keyboards to make filter, vibrato, or other changes without the need for pedals or a second hand. However, aftertouch creates a lot of MIDI data, and if the receiving device doesn't respond to aftertouch messages, it might be desirable to remove them from the data stream. Figure 1 shows the programming screen settings required to filter all levels of aftertouch on one MIDI channel.

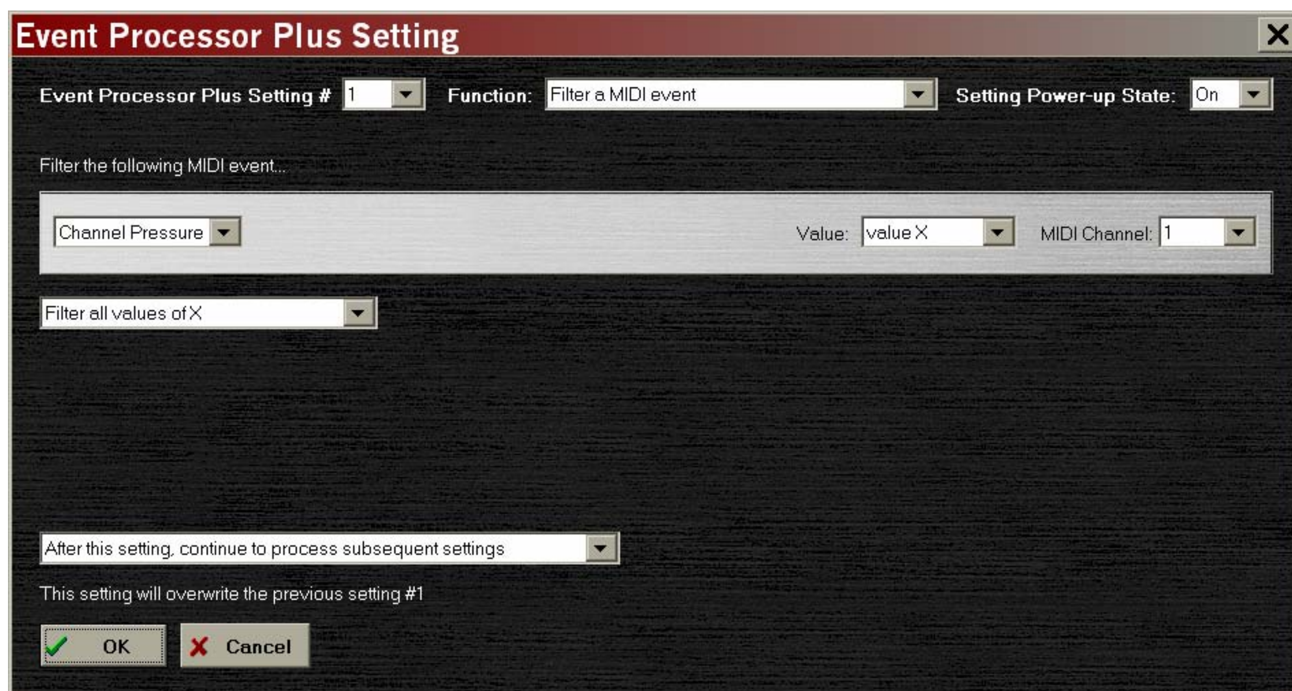


Figure 1 – Filtering Channel Pressure (Aftertouch)

Please note that we've selected to filter all values of pressure in this example. We could instead only filter out some values, leaving a reduced range of data to send to MIDI devices. Please also note that we've left the Setting Power-up State and Continue Processing boxes at their default selections. We'll discuss these settings in the Advanced Programming section.

MAPPING

Mapping is one of the most common reasons for needing a device like the Event Processor. A MIDI device may be able to respond to a certain control, but the sending device can't send the necessary information. The Event Processor can be programmed to change one event type into a different event; i.e., to 'map' the data into another data type.

Let's consider an example where we would like to change the Mod Wheel into an 'expression pedal.' Perhaps we have a piano module that can brighten or darken the tone of the sound through an expression pedal, but our controlling keyboard has no such pedal. Figure 2 shows the programming settings required to convert the Mod Wheel messages to Expression messages.

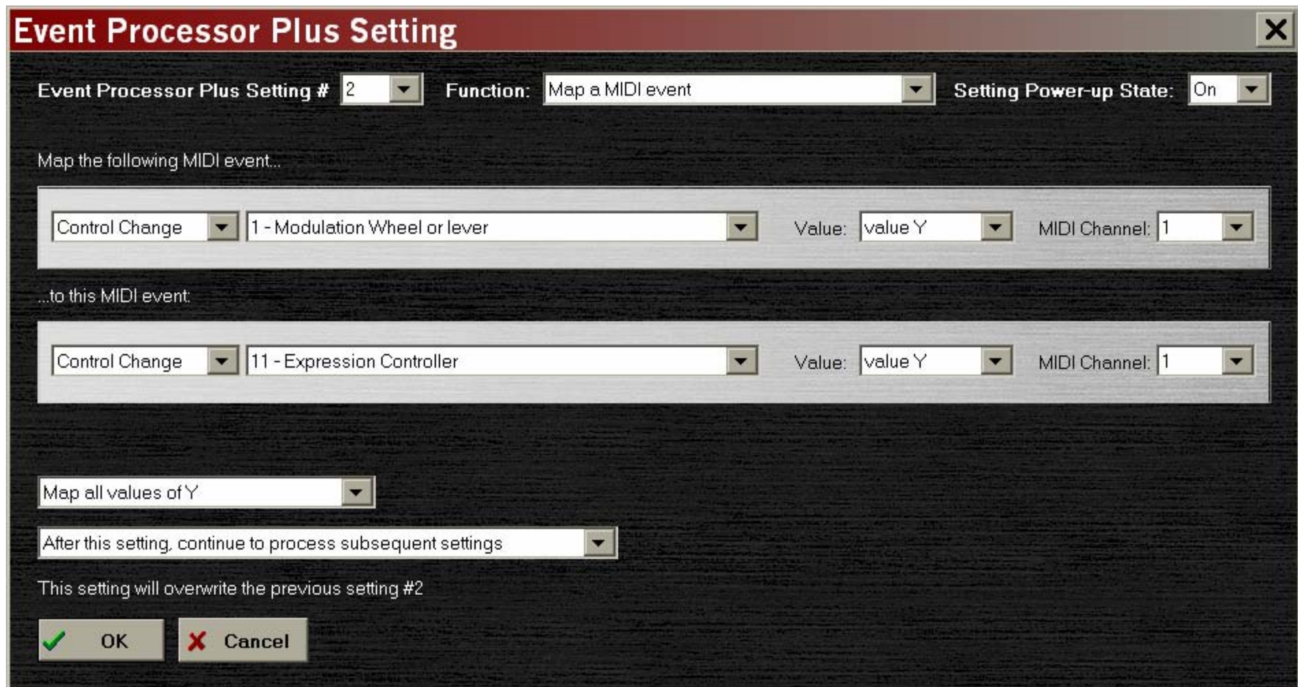


Figure 2 – Mapping the Mod Wheel to Expression

An important point to remember is that the Event Processor can map *one* MIDI event into *many* events. Sometimes, what's *not* already in the MIDI data is more important than what is. We could program the device to send Mod Wheel data on Channel 1 to Expression on Channel 1, Foot Pedal data on Channel 2, and Volume data on Channels 3 and 4. Remember, though, that each mapping uses up one Setting in the Event Processor.

Mapping isn't limited to changing one MIDI channel to another, or one control to another. The Event Processor can map notes to SysEx messages, standard controllers to NRPN – just about any MIDI event available into another event. In some cases, not all of the information in the original command can be transferred, however, because the two data types use a different number of data bytes. We'll discuss this again later.

It's also important to remember that mapping an event *replaces* the original data, and thus removes the original data from being seen at the output. If the original event needs to remain in place, because we want to *add* a message rather than replace it, we must also map the original event to it, as shown in Figure 3.

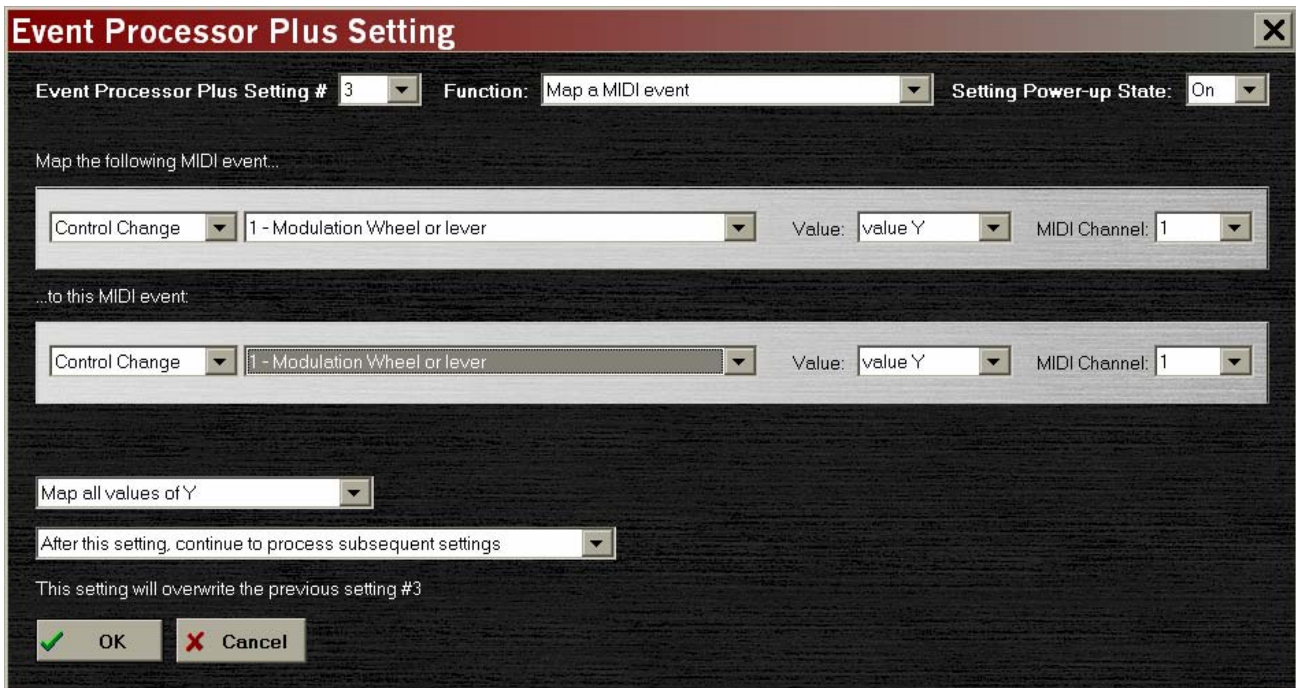


Figure 3 – Mapping an Event to Itself

TRIGGERING EVENTS

Sometimes we wish to change how a MIDI device works when a certain event crosses a certain threshold. Maybe we want to call up a Middle C on the piano when we push the Expression pedal down, or cue in a bank of lights by hitting a crescendo. The Event Processor can selectively process MIDI data based on a 'trip point.' For our next example, let's say that we have a spare volume/expression pedal lying around, and we want to control the speed of our MIDI organ module's rotating speaker (you know the name) simulator. Using the Trigger function, the Event Processor can be programmed to send one event when the pedal crosses a certain threshold, and a different event when the pedal crosses back. This turns the Expression pedal into a speed switch, and it doesn't require hitting an exact value; as long as the pedal sends a value above or below the threshold, the events will be triggered.

Figures 4 and 5 show how this can be done. In Setting #4, we trigger slow speed on incoming values of Expression (CC#11) 63 and below, and in Setting #5 we trigger fast speed on values 64 and above. Now we have a pedal that calls up slow speed when we pull back on it, and changes to fast speed when we push it down.

Note that unlike mapping, triggering does *not* filter out the original event: Expression data is still being sent to the rest of the MIDI chain.

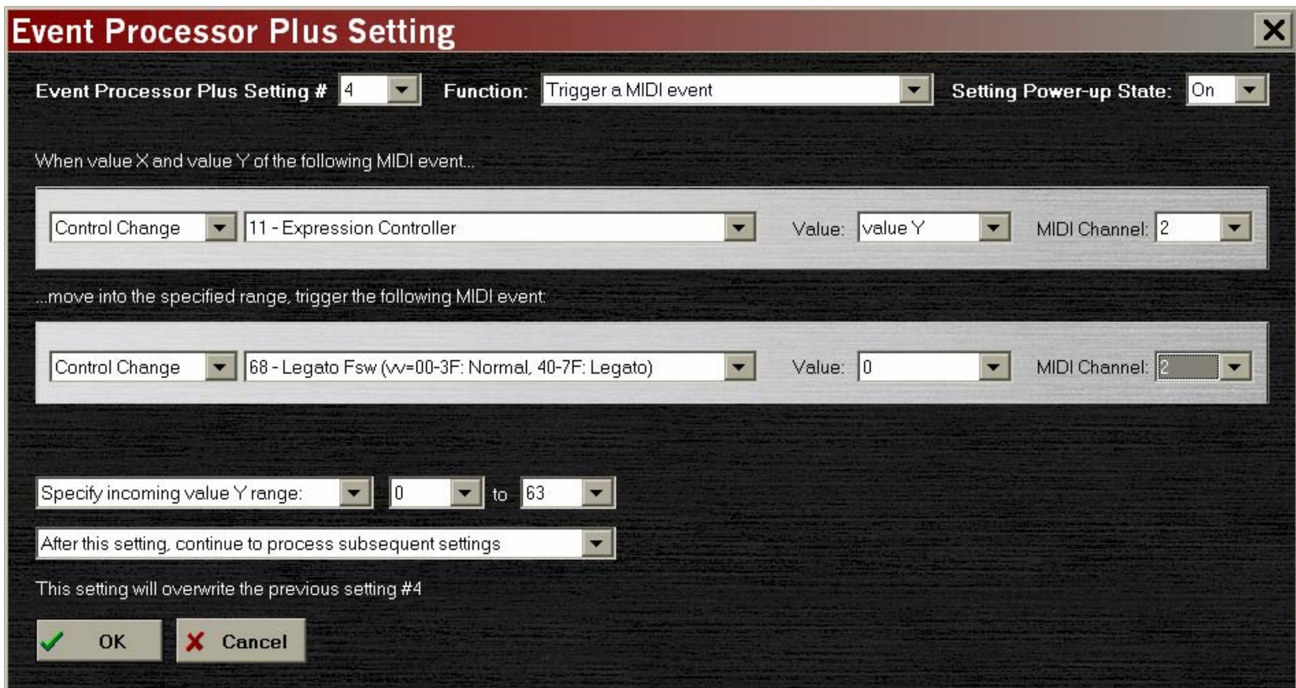


Figure 4 – Triggering Slow Speed

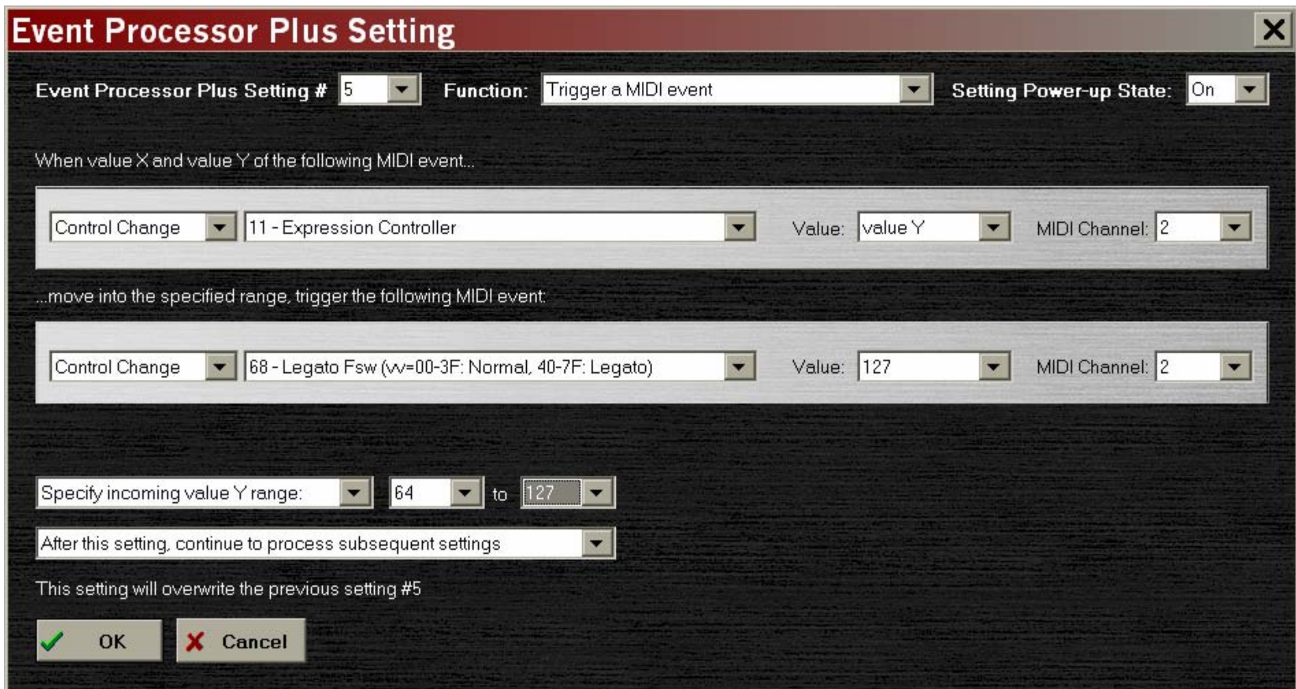


Figure 5 – Triggering Fast Speed

Another very important point to remember is that once a MIDI event is seen that is inside the trigger range of the Setting, the event will cause the Setting to “fire”; in order to be able to trigger again, another MIDI event outside the Setting range must be seen to “reload” the Setting. If Setting #4 used the range 0-127, the Setting would trigger the first time that an Expression event was seen, *and then never trigger again.*

One additional thing that we might want to consider is that the above example creates a very 'tight' speed control switch. If we were to sit with our foot on the pedal very close to the mid-point, tiny movements would send a slew of speed control changes. This is probably not a good thing! To prevent this problem, we could change the trigger ranges to say, 0-40 for slow speed and 90-127 for fast speed. This would result in a pedal that must be deliberately pushed toward fast or slow speed to react.

DEFINING SEQUENCES

Sometimes, we might want to alternate through two (or more) patterns with our trigger event. Let's modify our previous example to use the Sustain pedal to change the rotating speed. Unfortunately, the Sustain pedal is a 'momentary' switch, which sends one CC value when pressed and a different value when released. This results in a speed control that favors one speed over the other; the player must keep a foot on the pedal at all times to choose the "other" speed. Some players might like this method, but most of us grew up playing organs that changed rotating speed only on command. We'd rather tap the pedal once to change speeds, and then have the speed remain steady until the pedal is tapped again. Using the Define Sequence function, we can turn the sustain pedal from a momentary switch to an on/off switch.

To create this pattern, we need three Settings: the first one sets up the sequence; the remaining two Settings define the actions that occur in each step of the sequence. Figures 6-8 show these Settings.

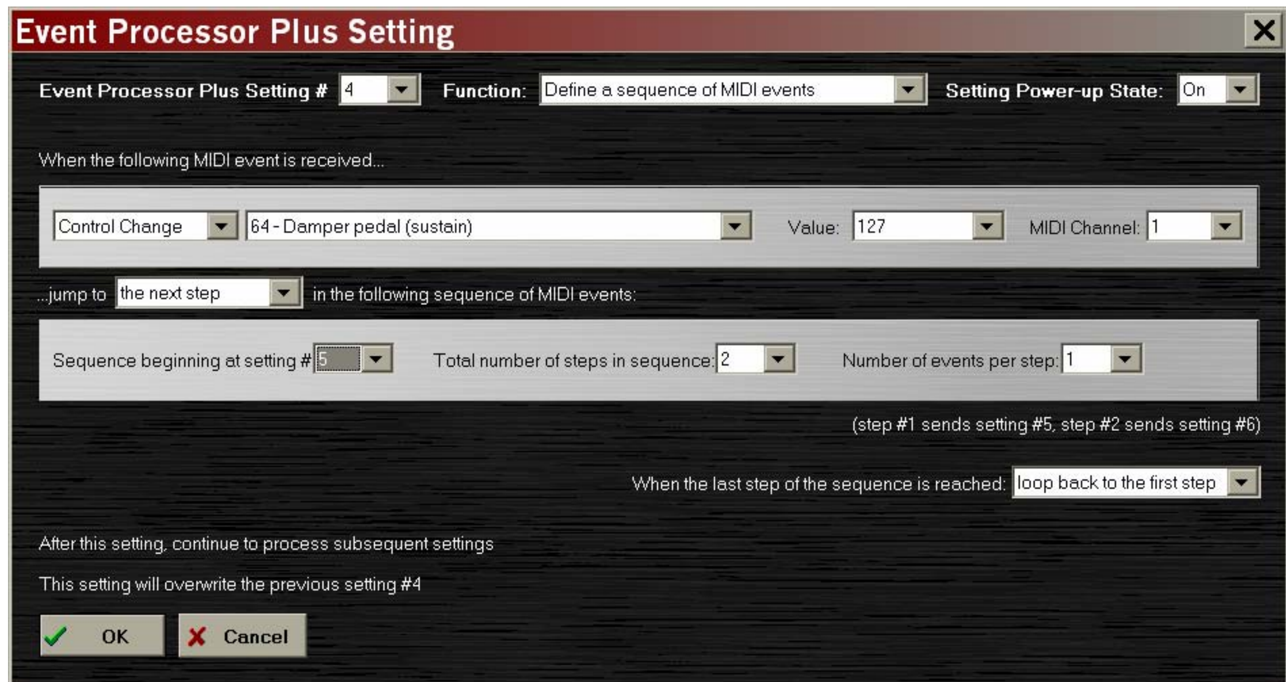


Figure 6 – Defining the Sequence

Please note that there are many variations to sequencing not demonstrated in this simple example. We can define a sequence with more than two steps; each step can have several Settings; and a sequence can stop after running once, or loop forever. The only limitation is that *each step of the sequence must be the same length* (one Setting, two Settings, etc.). If a need arises to send unequal steps, the short step must be padded with empty Settings so that all of the steps are equal.

Note: At the start of a program, all Settings are empty. Any Settings that aren't defined in the Programming Tools remain empty when the Event Processor is programmed.

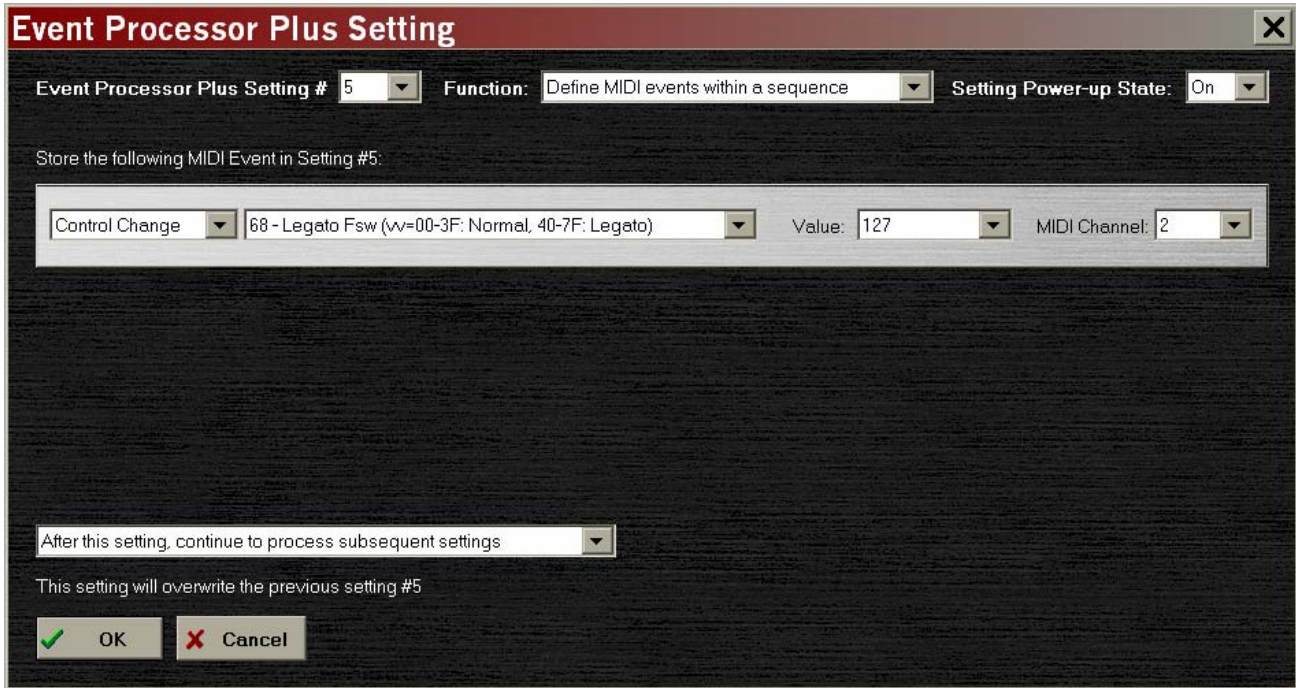


Figure 7 – First Step (Fast Speed)

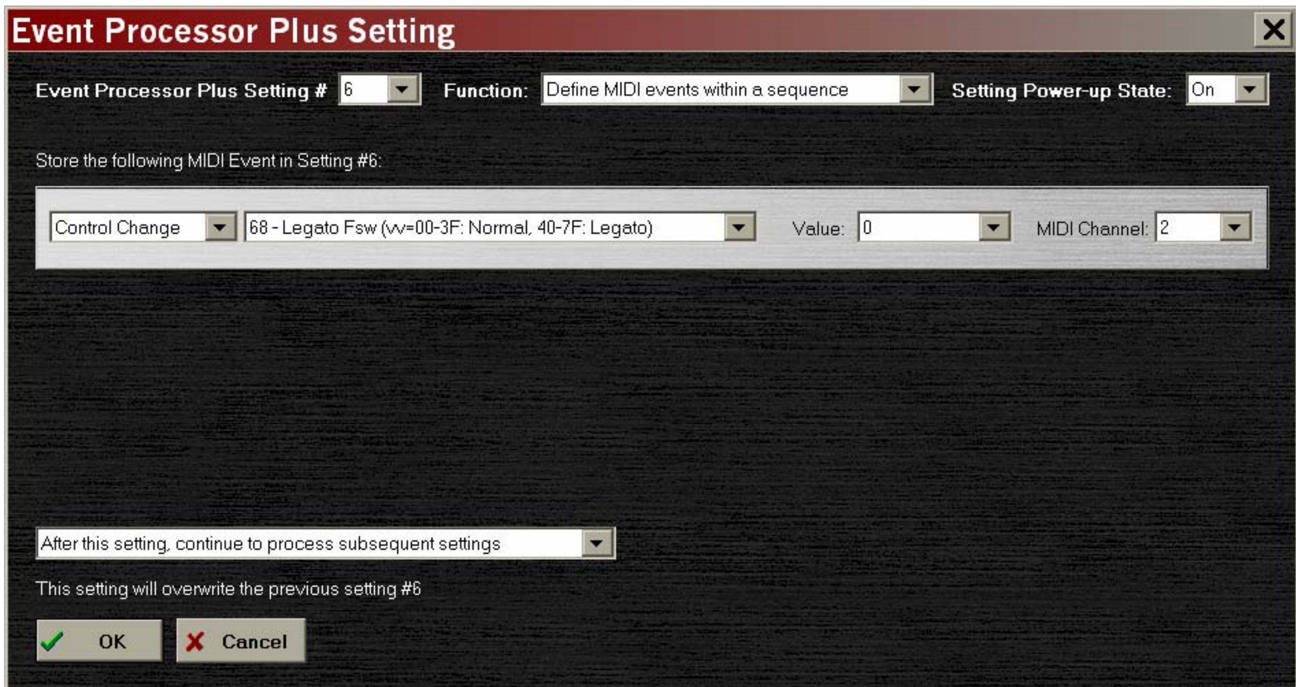


Figure 8 – Second Step (Slow Speed)

TURNING A SETTING ON AND OFF

Sometimes, it's useful to enable and disable some of the Event Processor's abilities based on other Events. For example, let's say that we want to map a portion of our digital piano keyboard as a zone to control an external module for left-hand bass – except that we don't want to *always* give up this section, because having 88 keys to play piano is useful, too. If our piano were a MIDI master keyboard, we would just press a couple of Zone buttons and split the keyboard on demand. But, our piano isn't that complex.

Fortunately, our digital piano has Pitch and Mod wheels. We could map the Mod Wheel to control the zones: When the wheel is less than halfway on, the piano controls the entire keyboard; when it's more than halfway on, the bass line takes over some of the keys. Figures 9 and 10 show how this can be accomplished: Setting #9 does the actual zone mapping, while Setting #6 turns the other Setting on or off based on the position of the Mod Wheel.

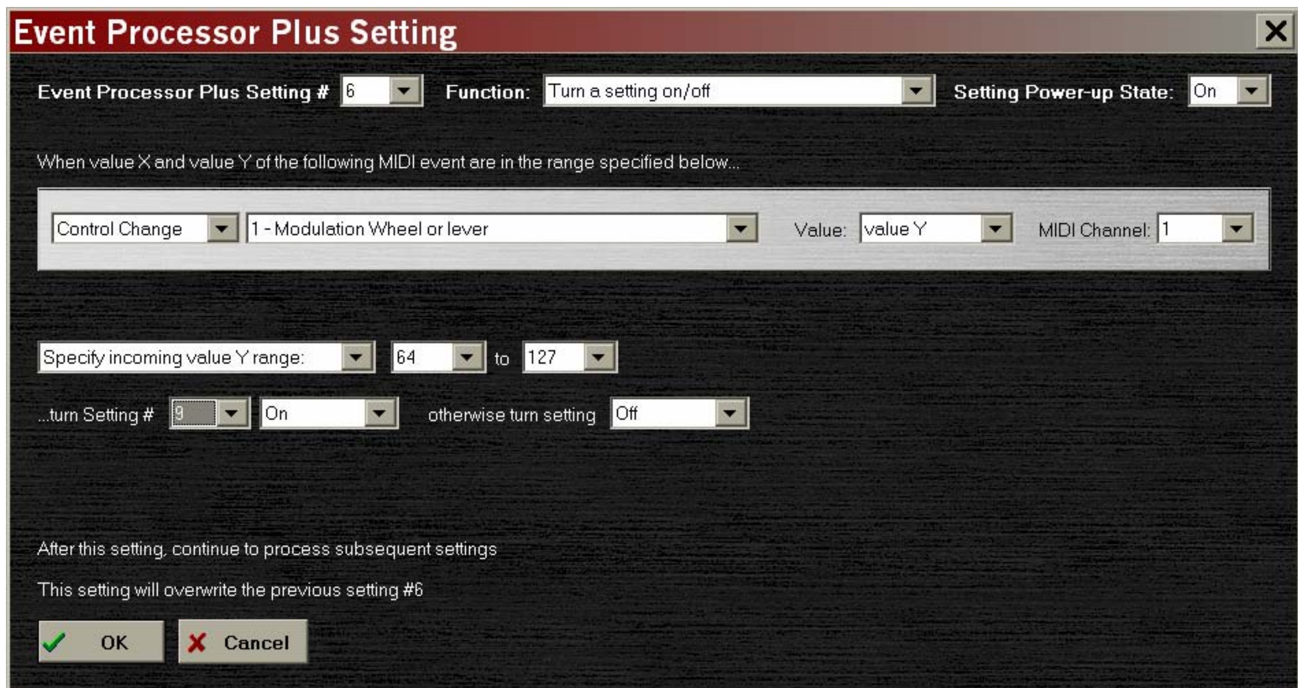


Figure 9 – Turning a Setting On/Off

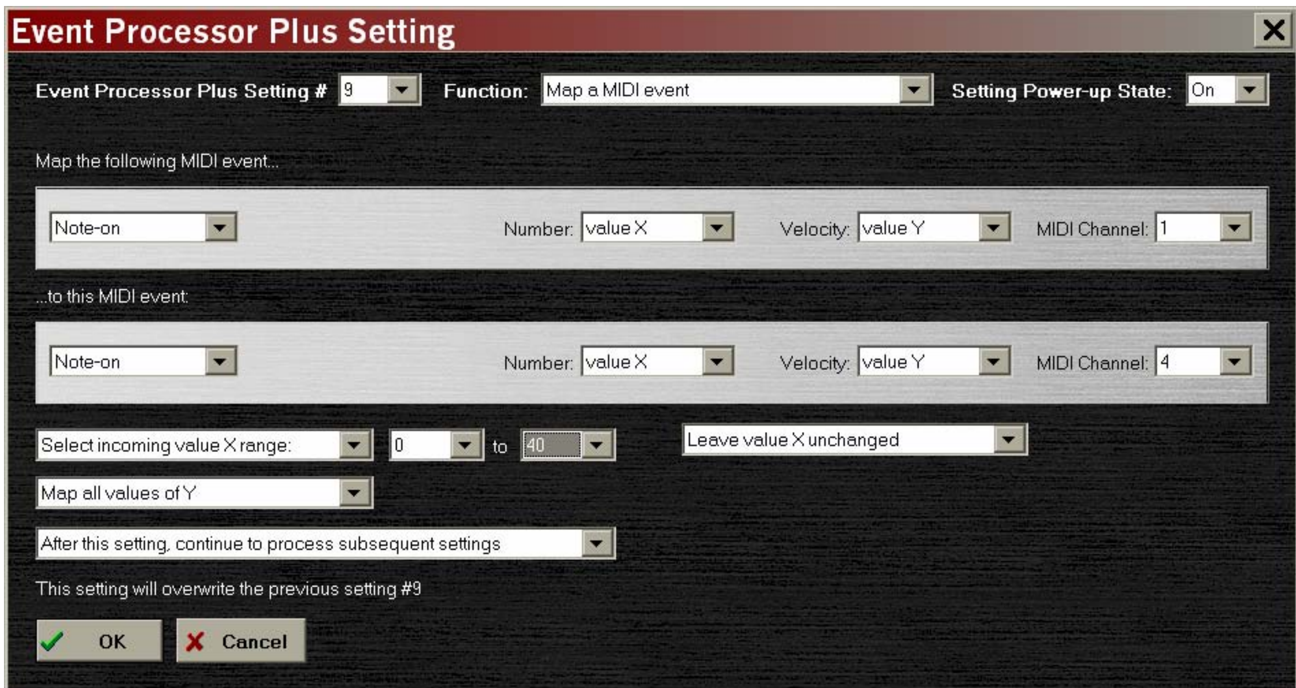


Figure 10 – Mapping the Bass Keys through the Mod Wheel

There is one problem here: If we play a note, move the Mod Wheel, then release the note, the note off (technically, Note On with velocity=0) will be sent on the new channel, rather than the old one. Unless we're looking for a drone effect, this is a bad situation. We need two more Settings which trigger All Notes Off messages on Channel 1 and 4 when the Mod Wheel moves in or out of the range that changes the channel. Figure 11 shows the first of these two Settings.

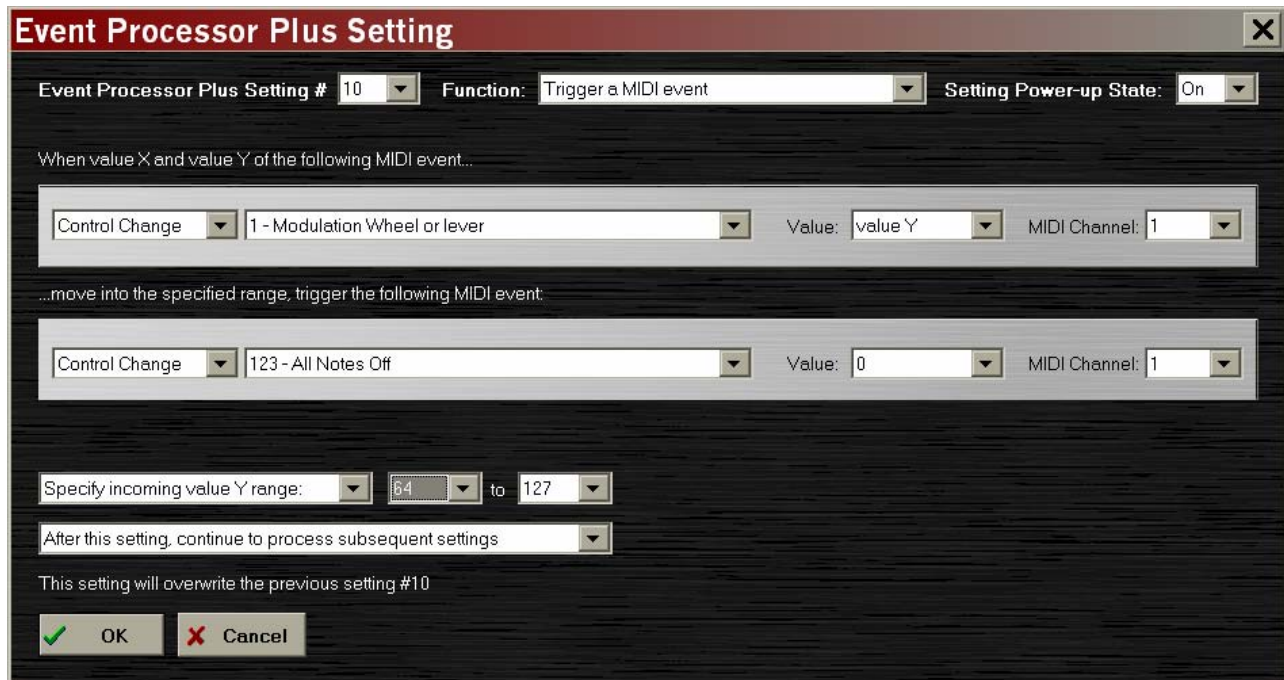


Figure 11 – Sending an All Notes Off Command to Prevent Stuck Notes

---

OK, we've now gone through all of the simple functions of the Event Processor. For many users, this is the end of the line – for now at least – and you can move off to programming the device with the Settings needed for your particular setup. Some MIDI Solutions customers use their devices to answer a single problem.

Some of you, however, may want the Event Processor to help in a more complicated setup, or change your MIDI setup frequently, or just want to understand what your device will be able to do if your setup becomes more complicated over time. The Event Processor's settings can be used to provide more complex changes in the function of MIDI gear than would appear and first glance. This brings us to …

CHAPTER 2 – ADVANCED PROGRAMMING

Now we get into the real "magic" of the Event Processor. This section shows how several of the basic features of the device can be used together to solve very complicated MIDI snags. It also discusses several options seen on the features discussed in earlier sections, which were deliberately left out of the Basic Programming discussions.

VARIABLES

The Event Processor has two Variable locations to save off MIDI data, while the Event Processor Plus has eight of these locations. We've intentionally left out this 'basic' tool, the setting of Variables, in our earlier discussions. We did this because while *setting* a variable is an easy task, the variable has no purpose until we *use it* to affect the way the Event Processor looks at data.

Sometimes, in order to solve a problem with MIDI gear, we need to know something about the data that was already sent. MIDI doesn't provide any way to look at past data; it's strictly a real-time communications link. A MIDI device could, of course, store up a record of past MIDI communications, but this takes hardware and software resources and few if any devices bother to record MIDI for later use. Of the few devices that do, almost none of them allow the user to access the data. Fortunately, the Event Processor can remember previous data for later use.

Let's say that we have a synthesizer that reinitializes a bunch of settings – volume, filter, Mod Wheel position, etc. – every time we change programs. That's a useful in some applications (like sequencer recording), but when playing live, it's very strange to hear the reverb knob jump from zero to a large value when touched after a Program Change. Yes, we can try to remember to always wiggle the reverb knob after a Program Change, but we'll probably forget sometimes. The Event Processor can make things *much* better! We can save off the setting of the reverb knob, Mod Wheel, and other settings every time they move, and then reapply the latest settings after each Program Change. We've effectively changed the way the synthesizer handles Program Changes, without having to touch the synthesizer. Figures 12 and 13 show how this can be done, using the Mod Wheel for our example.

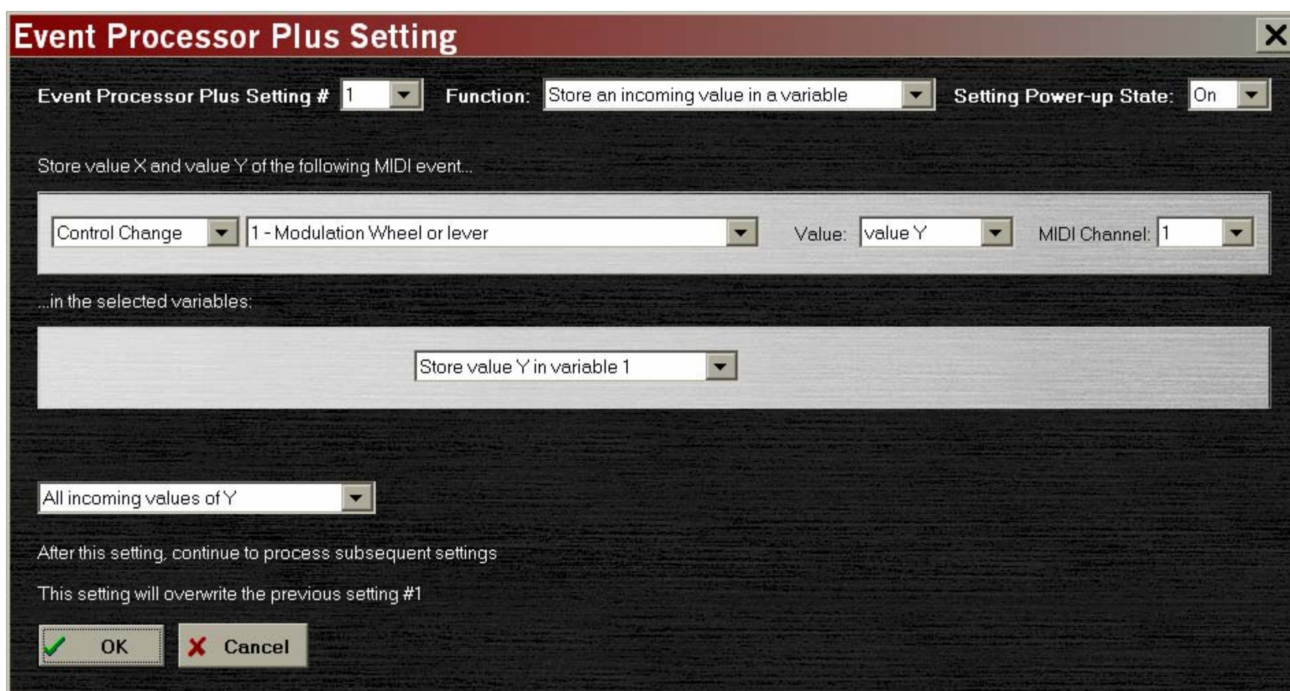


Figure 12 – Saving the Mod Wheel Position as a Variable

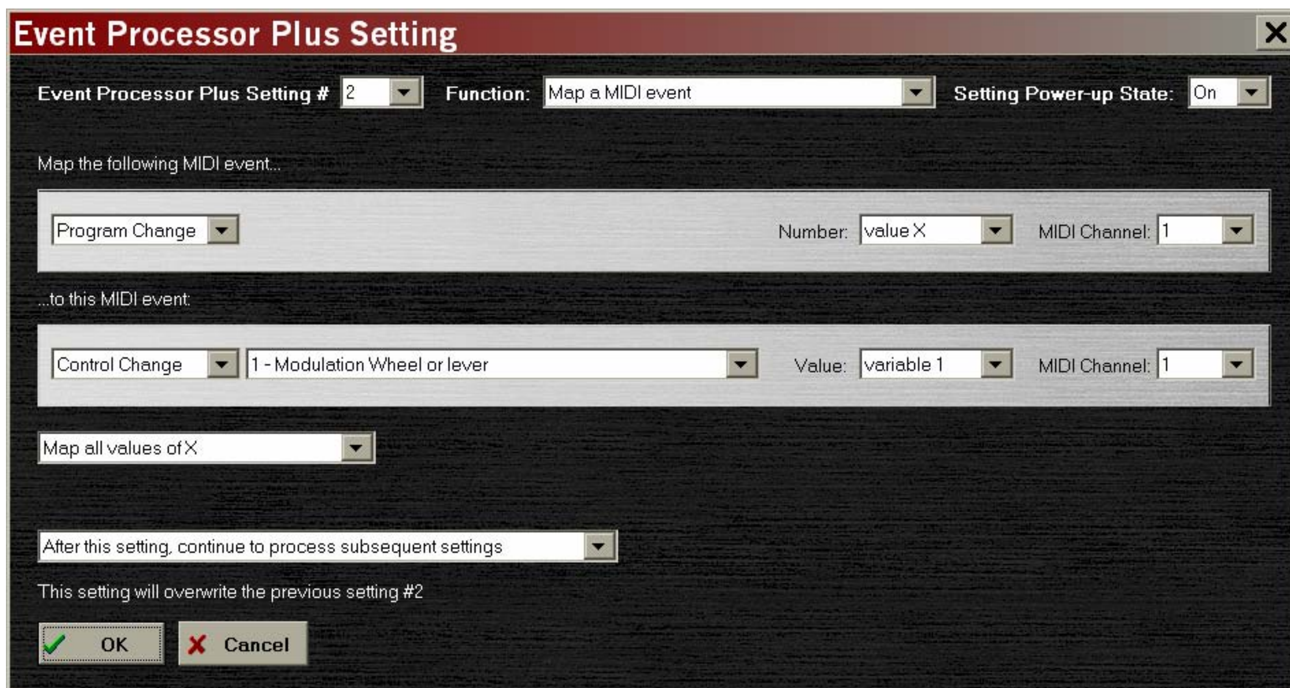


Figure 13 – Sending a Previous Mod Wheel Value

Remember that mapping the Program Change to the Mod Wheel setting automatically takes the Program Change out of the MIDI data. Although we haven't shown it here, we probably also want to map the Program Change to itself with a third Setting.

Another use for Variables is selective filtering of new MIDI events. Let's say that we can't eliminate all of the "hiccups" when a Program Change happens, the way we removed the Mod Wheel problem. Perhaps some of them are necessary changes to make our live show work. The changes are still bothersome, though, because they prevent playing the keyboard for a couple of seconds while all of the changes are processed. We really wish they wouldn't happen; at least not all the time. The changes are necessary, but maybe we can reduce the number of times that they happen. If we change the synthesizer to a classic Keith Emerson Moog patch, and our sequencer comes along and calls up the same patch change, why go through the whole mess twice? We're already set up correctly, right?

Figures 14 and 15 show the settings necessary to remove duplicate Program Change events. Once we receive a certain Program Change, all future Program Change events of the same value will be removed, but Program Changes with a *different* value will pass through normally.

This example demonstrates that the order of the Settings in a multi-Setting process is very important. If we were to reverse the order of the Settings, Setting #3 would store the Program Change in the Variable first, then Setting #4 would filter the Program Change based on this Variable, and the filtering would happen on every pass, *including the first time a Program Change is made.* Since we want the first Program Change to pass through, we must store the data in the variable after the filter Setting.

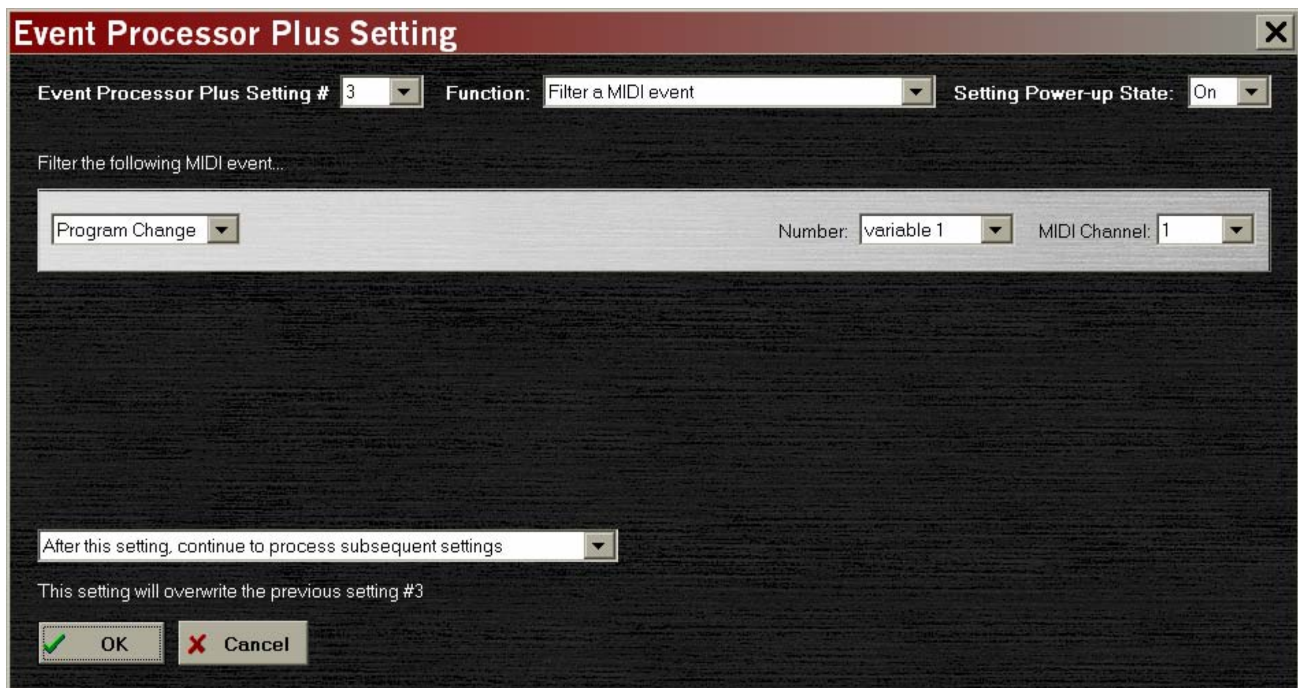


Figure 14 – Selective Filtering Based on a Variable

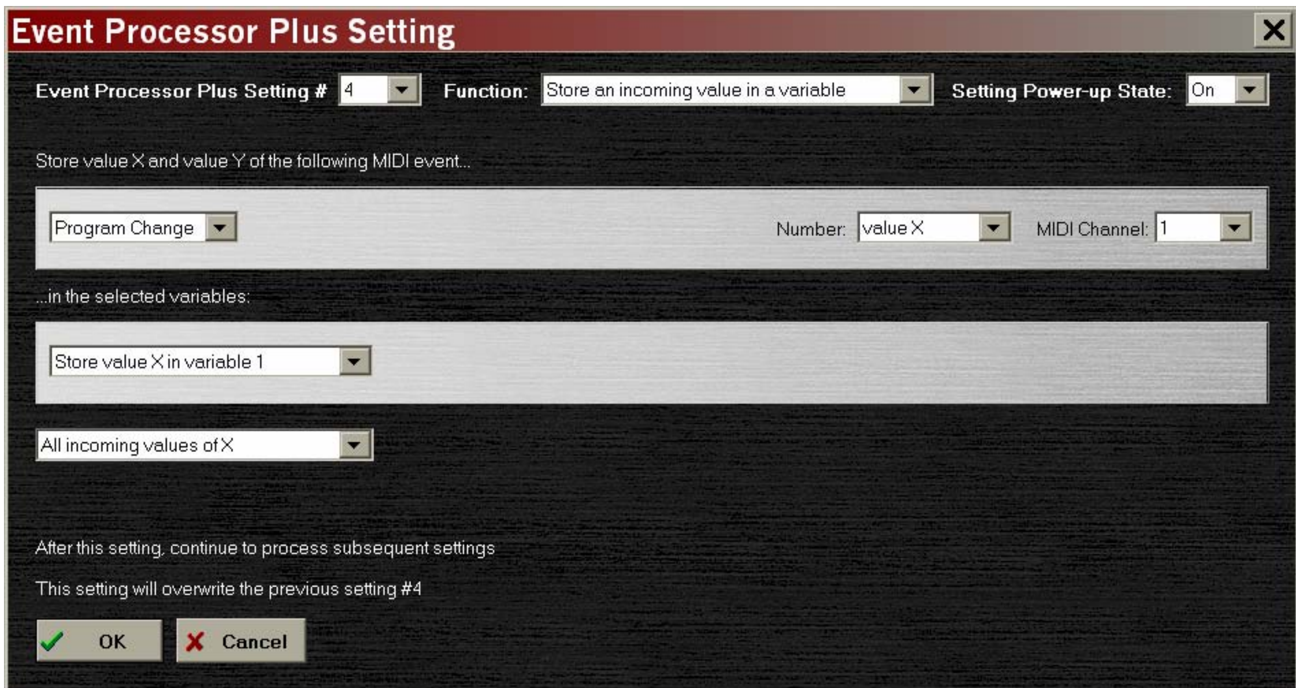


Figure 15 – Saving the Program Change Number

ADDING FEATURES BY ENABLING AND DISABLING SETTINGS – PART II

No matter how complex and well thought out a MIDI keyboard is, it seems like there's always at least *one* feature that the designers left out, right? And on the other side of the coin, there always seems to be at least one knob or button that has no purpose in your playing style. Maybe it's the volume knob, because you always like to run on "11," or the split button, because you never use splits. If that "useless" control sends out a MIDI message when it's used, the Event Processor can probably convert the message to something else!

Let's look at a MIDI master keyboard that has two zones. We're going to call up a rack of sound modules, however and wherever we want, on Channel 1 and 2. But, we also play left-hand bass once in a while – not often enough to dedicate one of these zones to bass, but we need this function at times. The bass is on Channel 9, and sometimes it's used with different combinations of other keyboards, so we want our new "Bass Button" to work with any combination of zones that are in place.

It just so happens that our MIDI master keyboard has a dedicated Celeste On/Off button – OK, a little weird, but Celeste seemed important back in 1988! We can set things up so that the bottom two octaves of the keyboard do what they normally do when "Celeste" is off, but call in the left-hand bass when it's turned on. Figures 16 and 17 show how this is accomplished: We create one Setting to re-map Channel 1 and 2 (and any other channel) to Channel 9, and another Setting to enable or disable the first Setting, based on the Celeste Button.

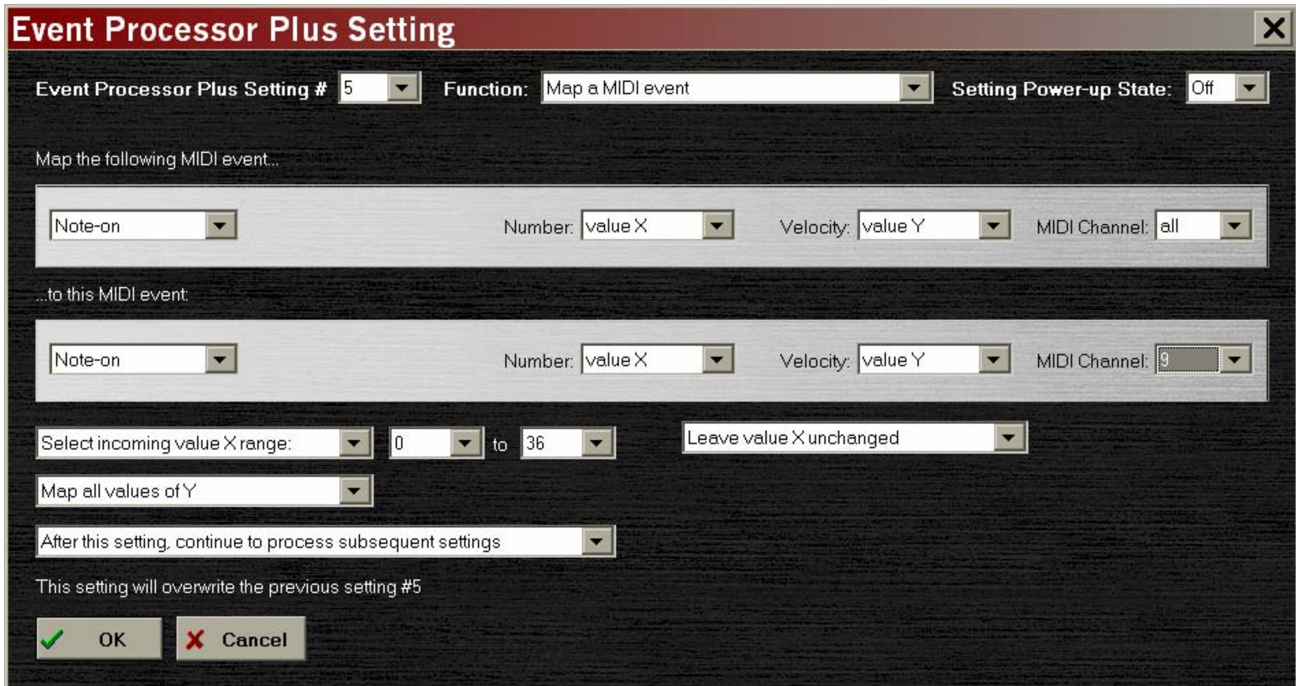


Figure 16 – Re-mapping the Bass Notes to Channel 9

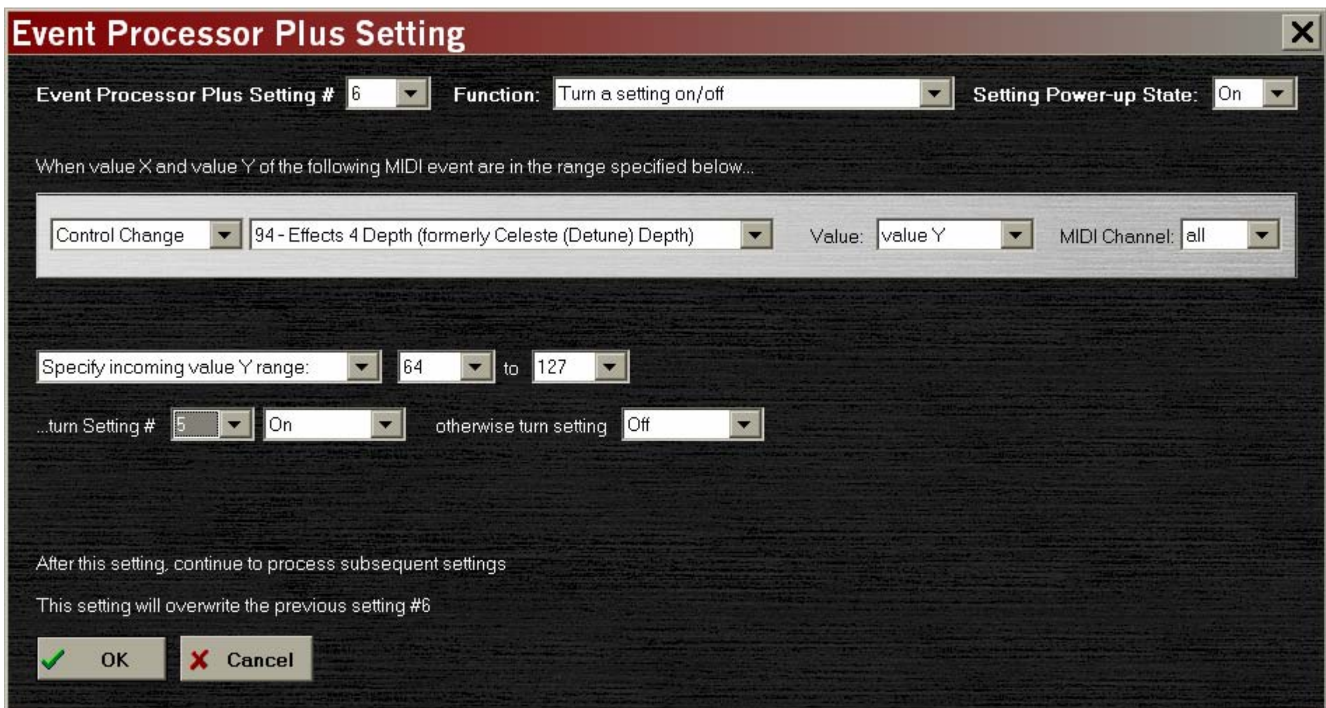


Figure 17 – Enabling Setting #5 through the Celeste Button

The reader should note that all Settings can be programmed to start enabled or disabled, to work better with other gear. In this case, the Celeste Button starts up disabled, so choosing to disable Setting 1 at startup eliminates the need to hit the Celeste Button a couple of times at the start of the show to synchronize the Event Processor with the button.

As in our earlier Mod Wheel example, there is a problem if we hold down the note while changing the channel: the Note On and Note Off messages will be on different channels. The solution to this problem is to again send All Notes Off messages each time the Celeste Button is pressed.

Similar control could, of course, be accomplished through Program Change 64, the Mod Wheel, a SysEx string, or even hitting the lowest note on the keyboard!

THE 'ALL' PARAMETER

In the preceding example, we used a dummy channel called 'All' to map the Note On messages to Channel 9, regardless of their origin. This only eliminates one Setting in our simple example, but if the keyboard had four or five zones, the savings would really add up. It's handy for global filtering Settings, too, such as eliminating Channel Pressure (Aftertouch) on all channels. Without an All variable, eliminating Aftertouch could require as many as 16 Settings, instead of just one.

REVERSE FILTERING

The All variable is also useful for "reverse filtering" or "programming by addition," where we start with a Setting that filters all the data, and then add Settings to let certain events back through to the output. Let's say that we want to use an old piano module from the '80s in our setup. The tone is killer, but the module overloads so easily that we can't connect it in line with our modern sequencer gear. Aftertouch sends it off, MIDI Clock sends it off – just about *everything* an acoustic piano can't do sends it off. Before we retire it, though, let's let the Event Processor take a crack at the problems. Figures 18-20 show how just three Settings can limit the MIDI data to the two elements the piano module likes to see: notes and the Sustain Pedal. Everything else is filtered out, regardless of the channel it arrives on.

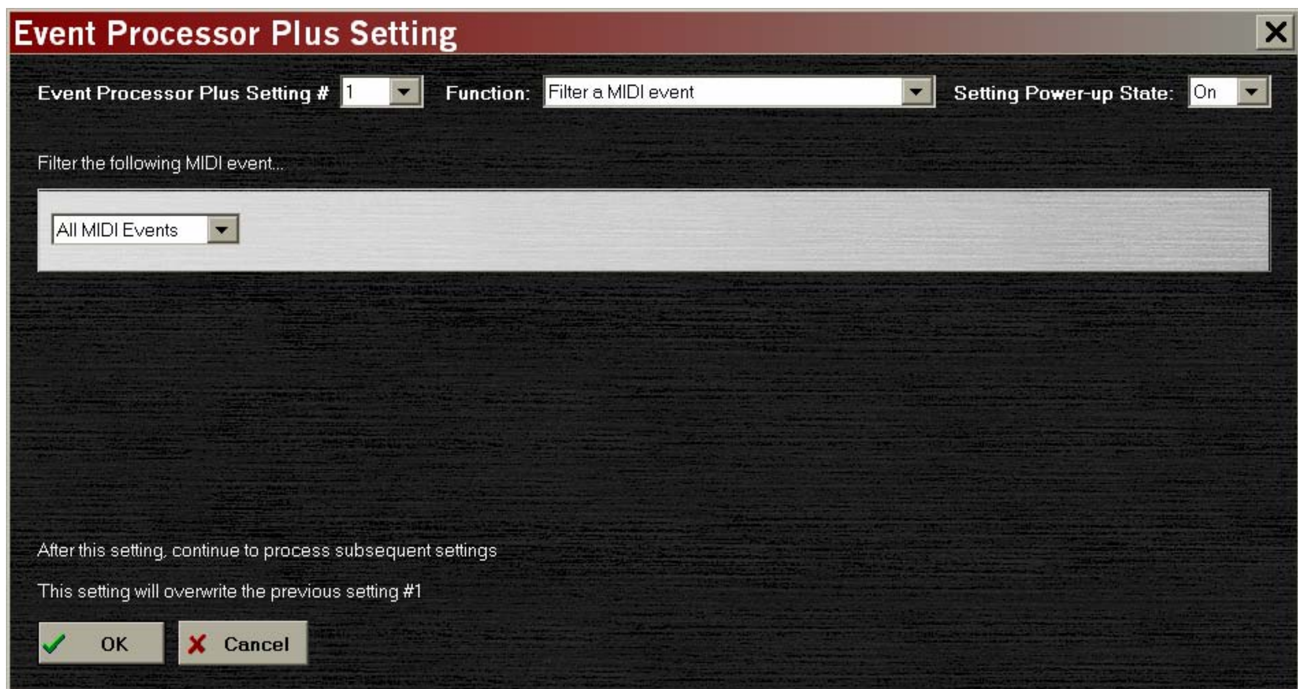


Figure 18 – Filtering Everything

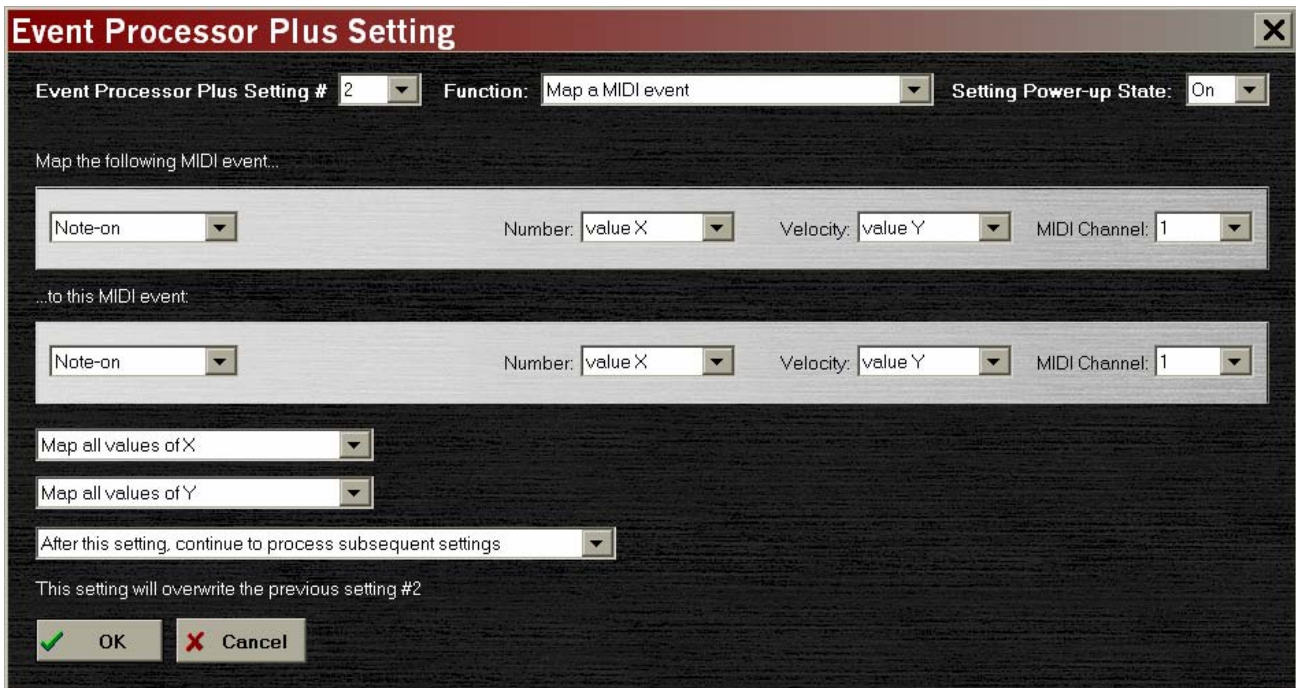


Figure 19 – Pass the Notes

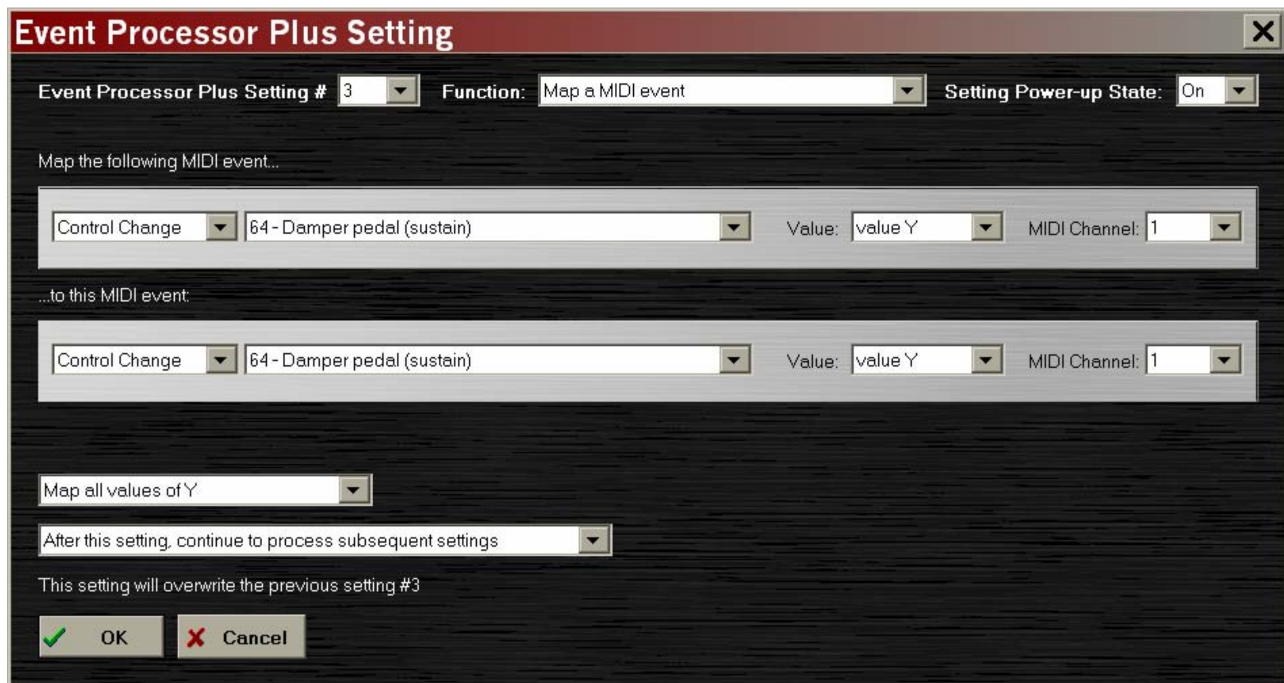


Figure 20 – Pass the Sustain Pedal

SELECTING RANGES AND SCALING VALUES

Sometimes, it's not necessary (or even desirable) to send the full range of 128 values that a controller can provide. The Event Processor can be programmed to only send a limited range of values from each Setting, and to scale a range of values from one data type to another. Beyond simple Mapping and Filtering, the Event Processor can selectively process MIDI data. For our next example, let's consider the control of 'drawbars' – those funny-looking slider-like controls – on a tonewheel organ. Most manufacturers dedicate nine CC locations, one per drawbar, for controlling this feature. Most manufacturers did so, but not all of them. One of the major organ makers figured that since a real drawbar is actually a nine-position switch, the 0-127 range of nine CCs is wasted in this situation. So, they mapped all nine drawbars to one CC – 9 drawbars x 9 positions equals 81 values, right? – saving eight CC locations. It's a neat trick, except that none of the other manufacturers followed suit, rendering this particular brand of organ pretty much incompatible with the rest of the industry.

Fortunately, the Event Processor can selectively convert the data into the 'standard' format so that we can use a different brand of master keyboard as a second manual for the organ, complete with its own set of drawbars. Figures 21 and 22 show two of nine Settings used to convert nine separate CC locations into the mapped single CC that the organ likes to see. Each Setting converts its 0-127 range into a very specific, limited portion of the 'Master CC'.

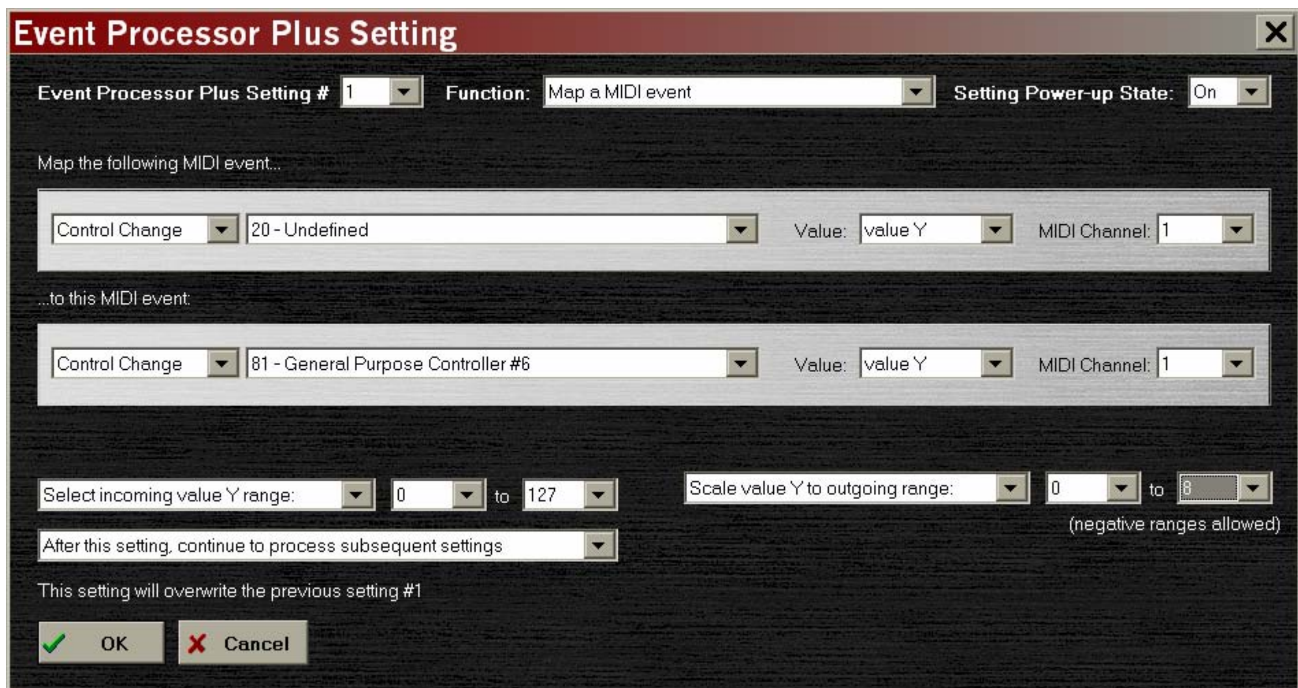


Figure 21 – 16' Drawbar Scaling

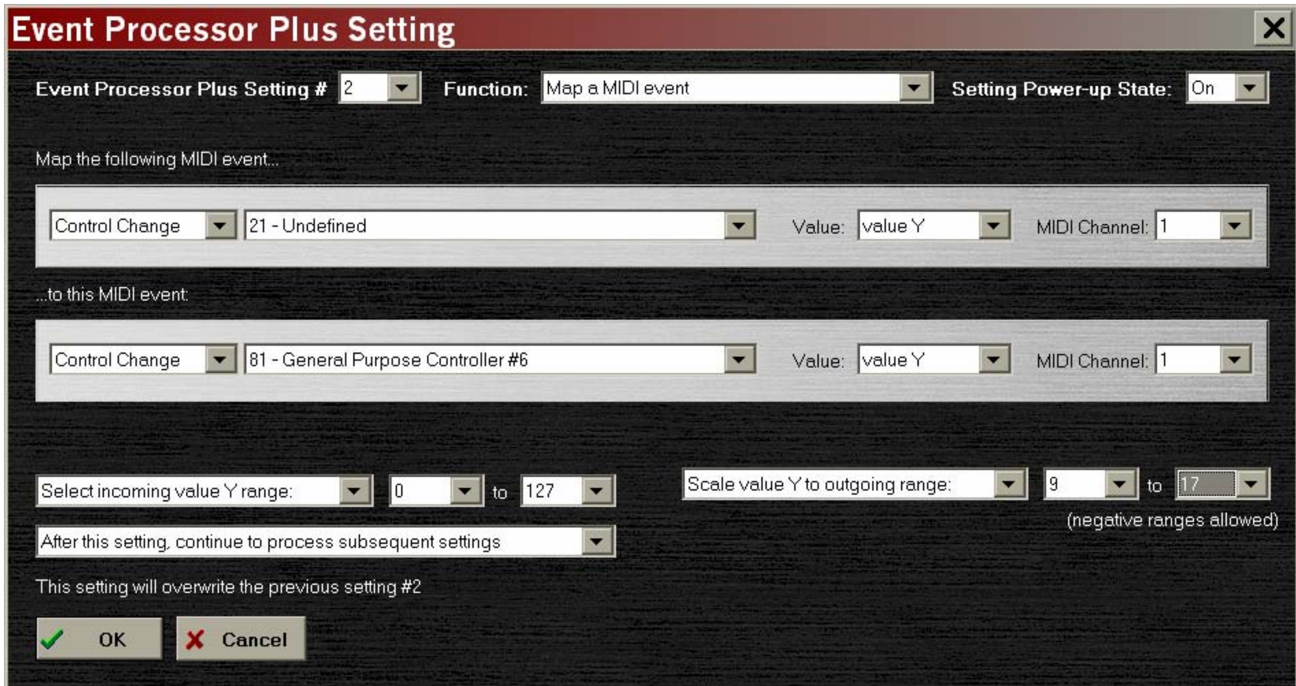


Figure 22 – 5-1/3' Drawbar Scaling

The scaling does not have to be a reduction, as in our example. A small range of MIDI values can be expanded into a larger one, too. Even a *reverse* mapping – i.e., making the knob or pedal work 'backwards' – can be programmed. This might be useful in our example if the controller keyboard sends standard slider messages where an 'up' push results in a larger control value; simply scale the output from 8 to 0, etc. Just remember that the Event Processor cannot magically add in missing values that never existed in the original data: If a 4-position switch is mapped to the volume pedal, the output values will be 0, 42, 84, and 127, and changing the switch positions will result in audible jumps in the volume.

INCREASING THE CAPABILITIES OF YOUR GEAR

As a general rule, the Event Processor's capability to add features to your setup is limited only by the imagination – although sometimes, we have to get "creative" to reach the right result. I have a very nice MIDI master keyboard, whose manufacturer and trade name shall remain nameless, that does just about anything that I could imagine – well, *almost* anything. The designers allowed each programmable button to send two different values when used for MIDI CC messages; I'm not limited to just 0 and 127. For whatever reason, though, it only works with CC data. I can't toggle between two different Program Changes, which would be really useful to save off a collection of often-used stored patches in my gear. (I know, how many Program buttons can a guy need? Well, it's a master keyboard, and it controls several modules, so the answer is, "more than I have.") How did I get around this?

MIDI defines 128 controllers for CC operations, and most of them are effectively undefined. In fact, the CCs above 100 are practically unused. I dedicated a bank of buttons to call up programs, but rather than send Program Change messages, I programmed each button to send out two different CC#118 values. (All things being equal, CC#118 is pretty harmless.) Then I use the Event Processor to convert CC#118 into Program Change messages, based on the data value of the CC#118, and I've tricked the master keyboard into sending two different – and completely selectable – Program Changes with each button! Figure 23 shows the details.

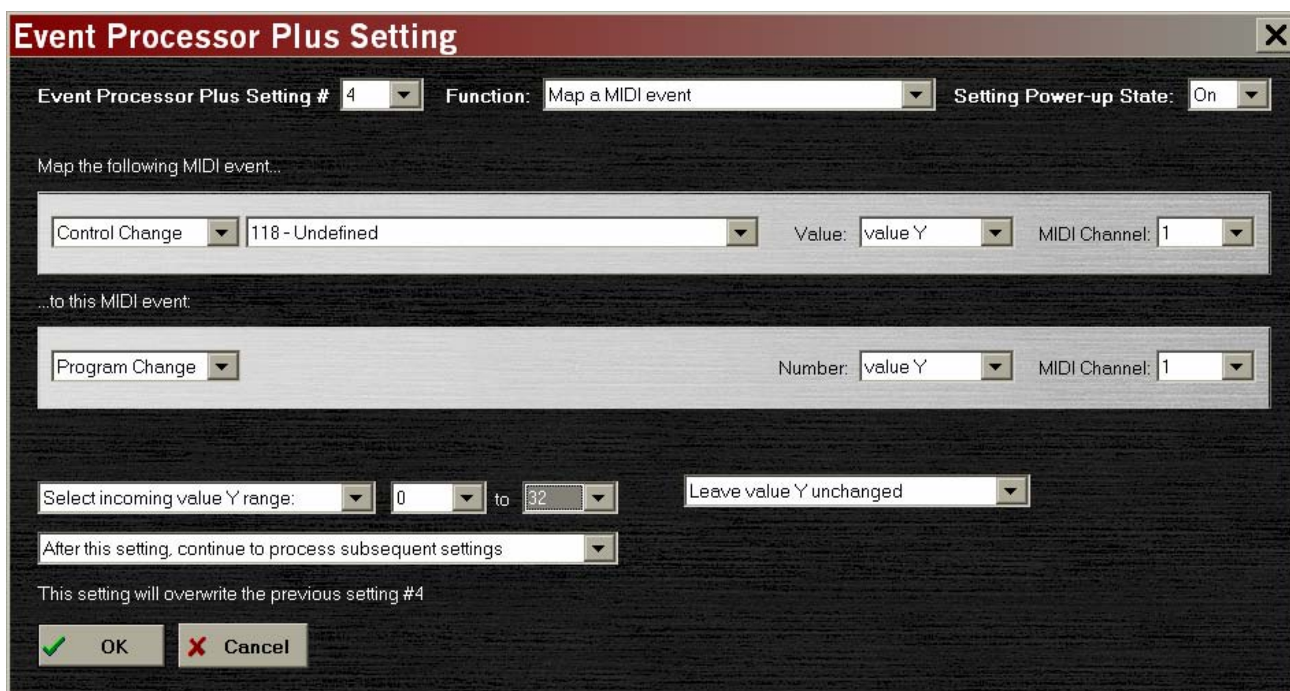


Figure 23 – Converting CC#118 into Program Changes

Please note that while I could have mapped the entire 128 Program Changes in this Setting, I am only dedicating the first 33 Program Changes to the module on Channel 1. This is because I might want to create a similar Setting for another device on a different channel. Thus, I can split up my Program Change 'method' and dole out patches to several devices, all based on the Program Change number – by using additional Settings, of course. If each of the devices can only accept say, 16 programs, I can 'recycle' the Program Change numbers by scaling CC118 values of 0-15 to PG#0 – PG#15 on Channel 1 in this Setting; CC118 values of 16-31 into PG#0 – PG#15 on Channel 2 in another Setting; etc.

CC#118 of value 0 on CH1	→ PG#0 on CH1	CC#118 of value 16 on CH1	→ PG#0 on CH2
CC#118 of value 1 on CH1	→ PG#1 on CH1	CC#118 of value 17 on CH1	→ PG#1 on CH2
CC#118 of value 2 on CH1	→ PG#2 on CH1	CC#118 of value 18 on CH1	→ PG#2 on CH2
…		…	
CC#118 of value 15 on CH1	→ PG#15 on CH1	CC#118 of value 31 on CH1	→ PG#15 on CH2

THE SUPER-FOOT-CONTROLLER

One of the more common examples of a rugged, easy-to-use MIDI control for the stage is the "foot controller" option that guitarists add to multi-effects units, especially the rack-mounted type. These controllers usually provide an expression pedal, plus a set of foot switches to call up user programs. (They sometimes have foot switches for CC use, too, which we'll ignore.) The expression pedal takes care of itself; if not, we can easily change the destination with another event. The program switches, however, aren't that useful to a keyboard player as-is. First, keyboards usually have other means of calling up programs; and the foot pedals are under the keyboards, and much harder to see. Second, I've always felt that the bank-and-four-switches method was a painful way to scroll through 40 or 50 programs on stage, because it's easy to lose track of which bank is "on deck." Unfortunately, the switches usually can't be programmed from their factory settings.

Adding an Event Processor to the foot controller can remove this restriction. We can re-map the foot switches to send CC, NRPN, SysEx – and yes, even Program Change – messages. For simplicity, let's use a foot controller that has one Bank switch and four Program switches, with eight banks total. We only want to define two banks of settings for our own use. The first bank will be (surprise!) Program Change messages, while the

second bank will send CC commands to turn on reverb and echo effects. The mapping scheme of foot controllers is straightforward: the foot switches send a short series of program messages, while the Bank switch adds an offset to continue to call up more series of PG settings:

Bank = 0	PG#0	PG#1	PG#2	PG#3
Bank = 1	PG#4	PG#5	PG#6	PG#7
Bank = 2	PG#8	PG#9	PG#10	PG#11
…				
Bank = 7	PG#28	PG#29	PG#30	PG#31

Since we only want two banks, but our controller sends eight, Banks 2, 4, and 6 will simply mirror the information sent in Bank 0; and Banks 3, 5, and 7 will mirror the information in Bank 1. This means that we can never be more than one bank away from where we want to be, even if we lose track of the bank number. Figure 24 shows the first of the three 'mirror' events, where we take PG#8-PG#11 and re-map them to PG#0-PG#3. We need only three of these events, because it's not necessary to re-map Bank 0; it already does what we want.

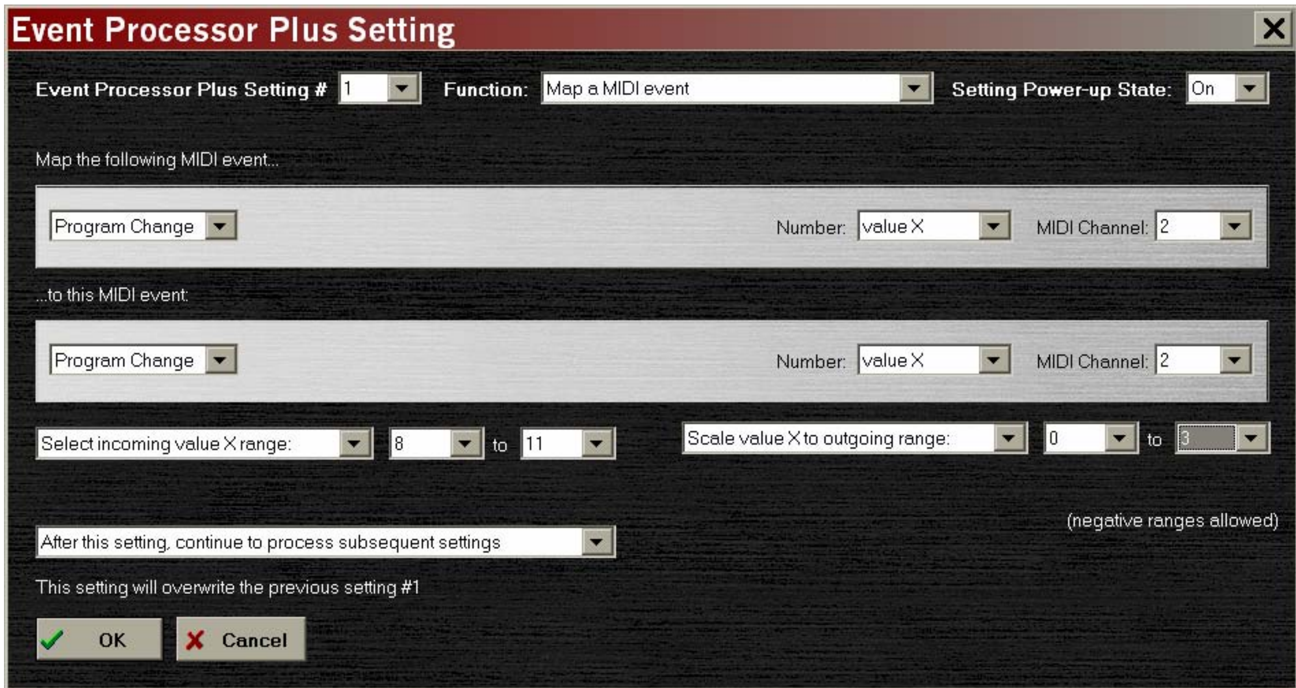


Figure 24 – Mirroring PG Bank 2 to Bank 0

Figures 25 and 26 show the mapping of the four Program Change buttons of Bank 1 to a pair of MIDI CC controls, CC#12 and CC#13. In our example, the first two "program" switches turn these controls on, while the second two turn them off.

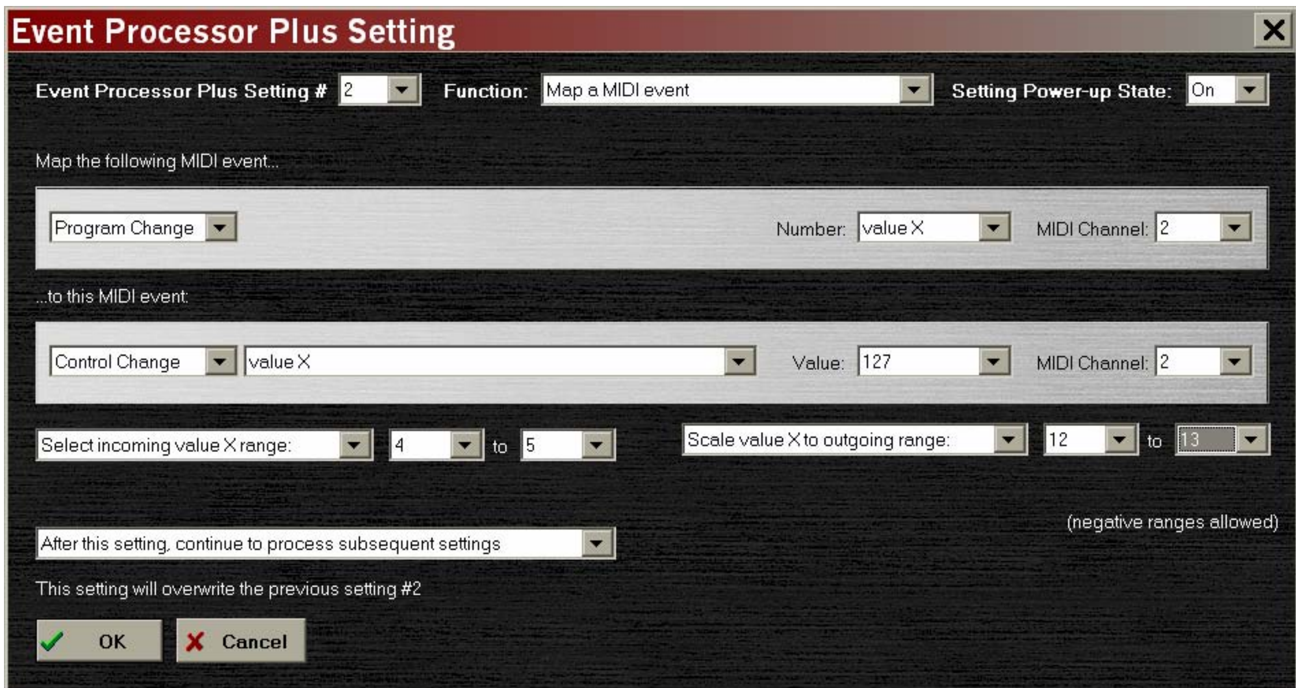


Figure 25 – Using Program Changes to Turn CCs On

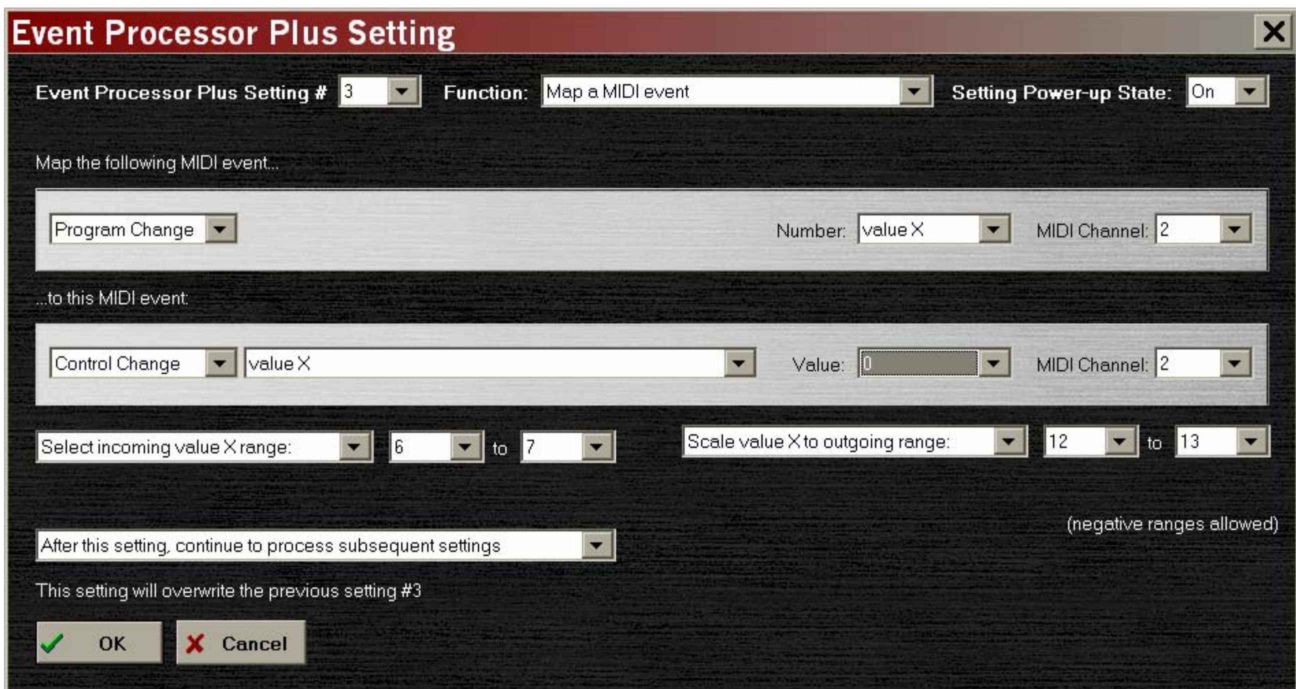


Figure 26 –Turning the CCs Off

Because of the number of banks in the foot controller, we will need four variations of each of these two Settings to support the full configuration of the foot controller, thus making this example suitable only to the Event Processor Plus. Why so many – can't Banks 1, 3, 5, and 7 be combined, and *then* mapped as CC messages? Unfortunately, no: while the outside world sees only the final results of the Event Processor, inside the device the 32 individual Program Change messages are still intact.

SUPER-CONTROLLER #2

OK, we're happy with our pedal controller, except for the on-on-off-off order of the effects controls. What if we want to arrange CC#12 and CC#13 in the more common on-off order? It's easier to leave a foot in between two switches, and learn to nudge left or right to turn a particular control on or off, than to make the larger jump over an unwanted switch. Is there a way to work around this seeming limitation, and put each controller's switches next to each other?

Figure 27 and 28 show another way to solve this problem, which results in a very compact solution that is not as intuitive to *program,* but very intuitive to *use.* Instead of breaking our problem down into 'on' events and 'off' events, let's work on each CC function separately, and use reverse scaling to call up on and off values. The lower Program Change of each Setting calls the 'on' value, while the higher Program Change calls the 'off' value. Since there are only two values to the scaling, we could safely call any two numbers, rather than just on and off; our reverb switch pair could just as easily call up values of 127 for "lots of reverb" and 25 for "a little reverb."

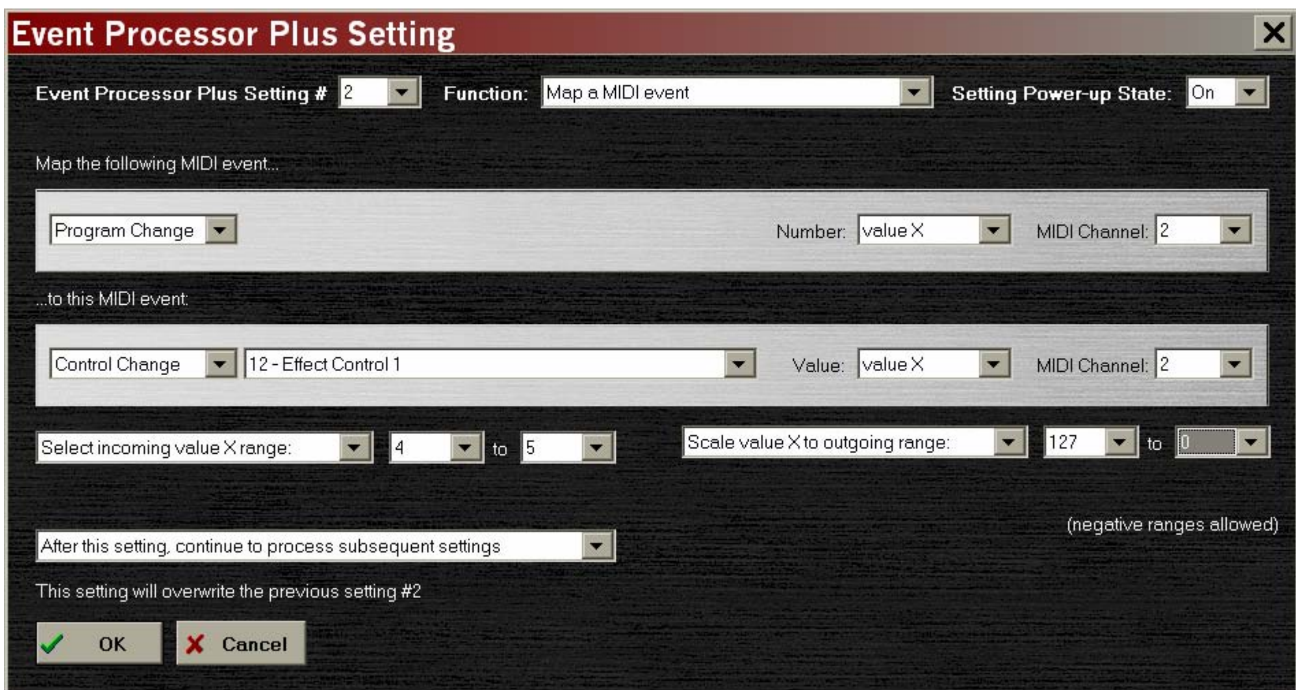


Figure 27 – Controlling CC#12 Directly

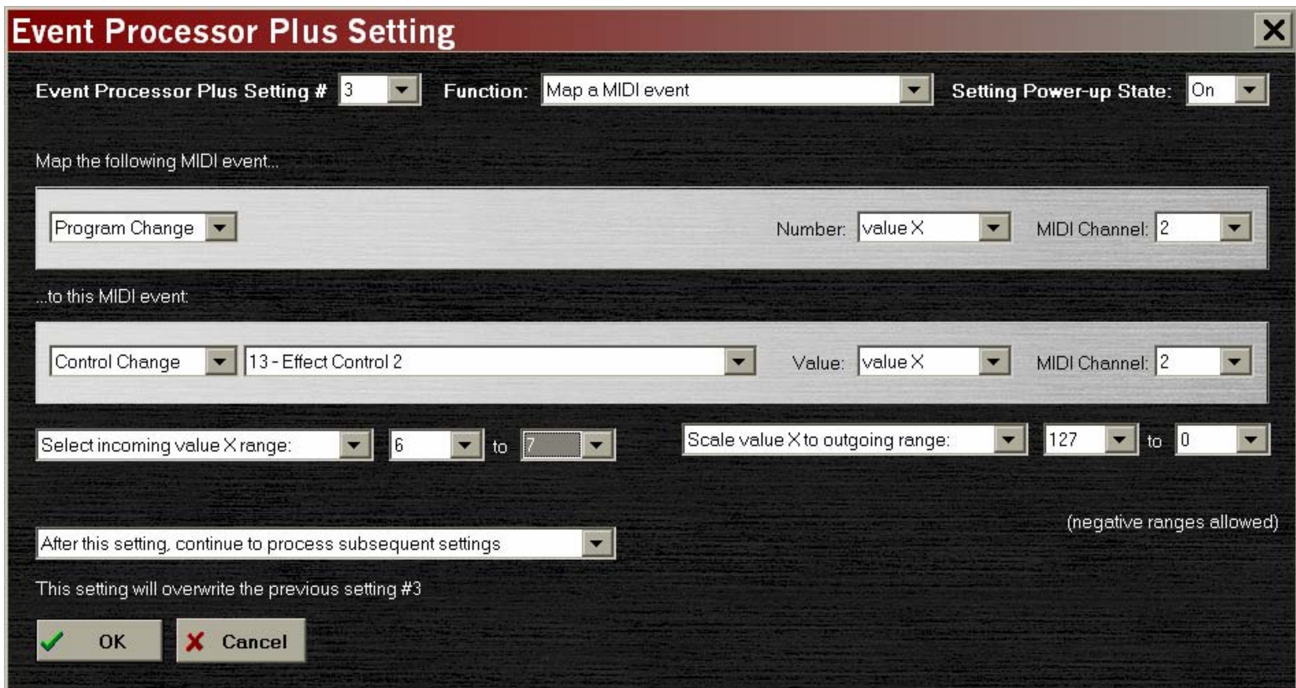


Figure 28 –Controlling CC#13 Directly

This illustrates a useful point to remember: There are often several ways to solve a problem in the Event Processor, and some of the ways may be more efficient in the long run, if the mapped items can be grouped properly. Sometimes it's possible to do *more* than 10 tasks in an Event Processor, or more than 32 tasks in an Event Processor Plus. In this case, a *single* Setting is both changing the switch function and scaling the values.

VELOCITY SENSITIVITY FROM AN ORGAN

For our final Variable-based example, let's consider using a MIDI organ to control a piano module. While many modern MIDI organs can send velocity information, let's say that we want to use a vintage organ with a MIDI retrofit, and these retrofits normally send fixed velocity. This is a very poor combination, because a piano's real strength is in the timbre changes brought on by changes in volume. We can use a volume pedal to change the output volume of the piano module, but it won't have anywhere near the expressiveness of a real piano. Unfortunately, the organ's MIDI retrofit doesn't send any CC that would help us out.

However, we could keep track of the organ's volume pedal, and use that information to change the velocity data that the keyboard sends. It's not a perfect answer, but most organists are used to using the expression pedal to provide tonal changes, so it just might work! Figure 29 and 30 describe how this process works. Setting #7 saves off the value of any expression pedal messages that come from the organ, and Setting #8 substitutes that value for the velocity data in the Note On messages.

Please note that we are only acting on the velocity range 1 – 127 because a Note On with velocity of zero is considered a Note Off.

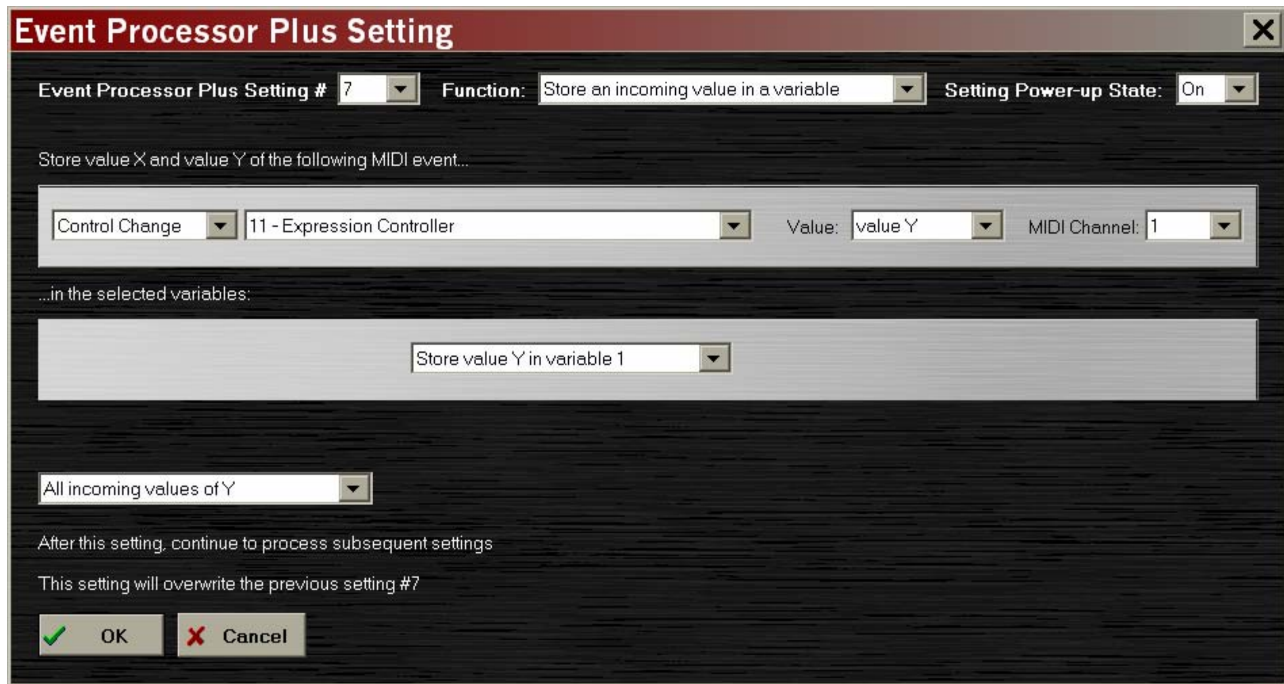


Figure 29 – Saving the Expression Data

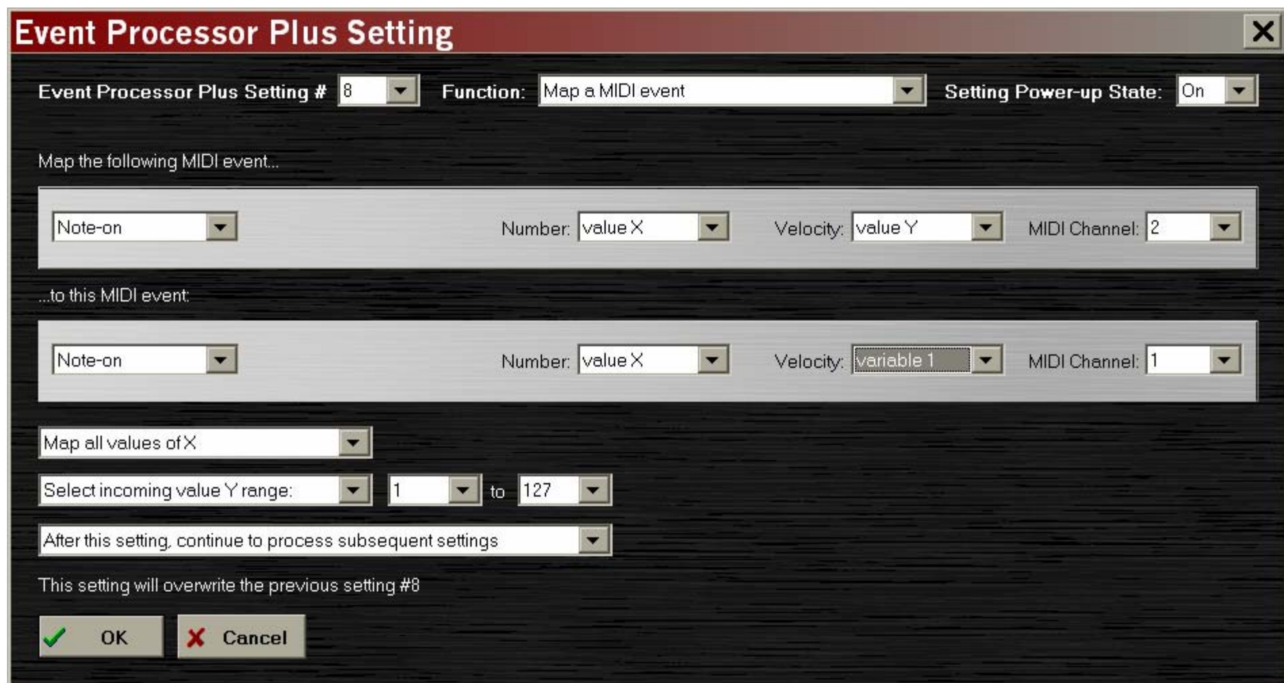


Figure 30 – Sending Expression as Velocity

DELAYS AND OTHER 'HARMLESS' MIDI STRINGS

Sometimes it's necessary to delay between sending MIDI messages, or to pad an uneven cycle of Settings to meet the requirements of the Cycle function. One of the most common places where this limitation rears its ugly head is during Program Changes: some MIDI devices effectively go off-line for as much as a second when changing stored sounds! (Those of you who own some of these products are probably more than aware of the performance implications.) MIDI, unfortunately, has no 'hold' or 'wait' command, and it's not possible to pause the Event Processor to wait for a better time to send a message. One way to get around this restriction is to send some extra MIDI data after a required message, using up the MIDI data line for a period of time.

One of the most harmless things that I can think of to send is a 'null SysEx' message. SysEx messages are a special case of MIDI. They always start with F0 and end with F7; in between those two bytes, pretty much anything goes, although the strict convention is to send a device's registered ID immediately after the F0, so all of the devices on the line can choose to obey or ignore the SysEx message. Each manufacturer who joins the MIDI Manufacturers Association is assigned a SysEx ID to use to route their own special messages to their devices.

There's a bit of a loophole here that we can exploit. By sending only the F0 to start the message, then a 00 – or a string of them – and then terminating the message, *nobody* recognizes the SysEx message as "theirs," and everything goes on as before! In the meantime, though, the MIDI connection has been tied up with the SysEx message, preventing further communications. We've tricked MIDI into generating a pause! Figure 31 shows an event that triggers a 25-byte null SysEx message, resulting in 8.3 milliseconds of delay.

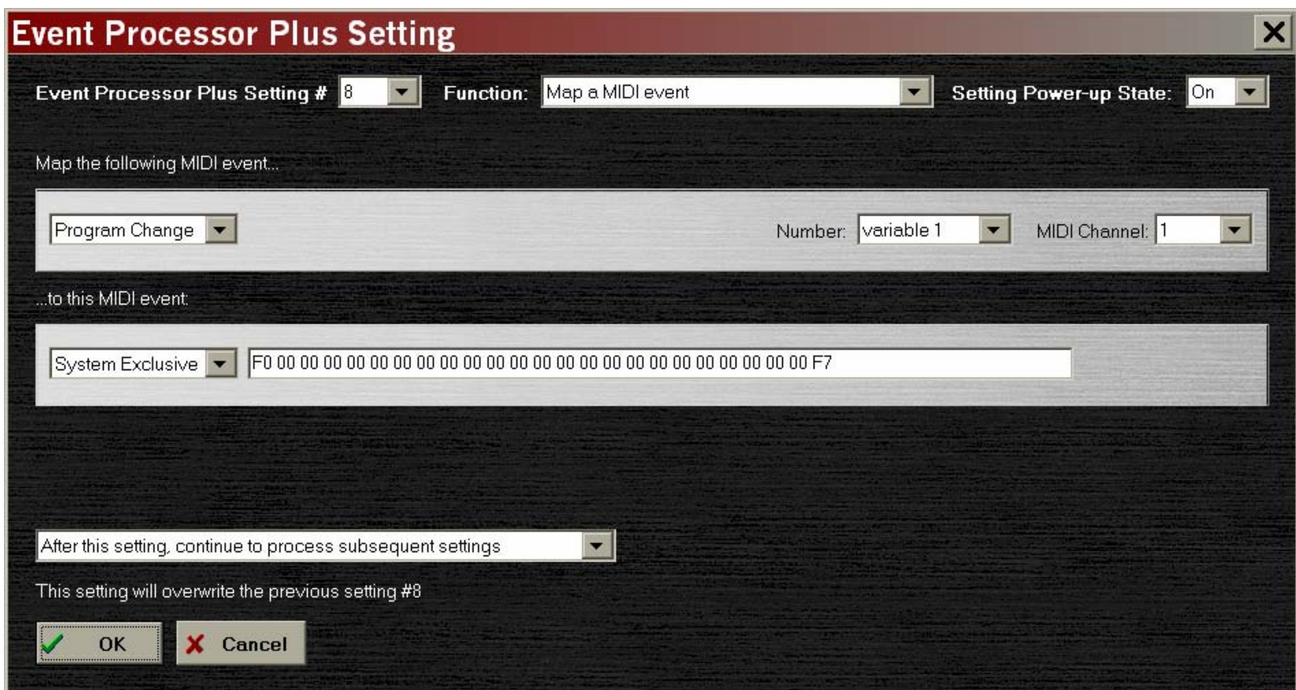


Figure 31 – Null SysEx Transmission

What's wrong with this method? Nothing really, even though MIDI devices don't plan for pauses. A SysEx message has a high priority, and most other MIDI communications must stop and let the SysEx go through. For brief pauses between songs, it works well. The Event Processor allows SysEx strings of up to 32 bytes, including the F0 and F7. Several SysEx events can be queued up back-to-back to create longer pauses. MIDI sends about 3,000 bytes of data per second, so figure that every three SysEx bytes equal 1 millisecond of pause, or 10 milliseconds per 32-byte SysEx event.

What else can we use, if SysEx doesn't look like a good option? Here's my personal favorite list, in order of decreasing harmlessness:

- **MIDI CC IDs between CC#102 and CC#119** – hardly ever used, especially the ones over 110. I've used CC#116-CC#119 all the time for Event Processor features, with great success.

- **Unused NRPN locations** – there are a lot of these, but you'll have to experiment to make sure that the chosen values are really unused. The nice thing is, with 16,000 choices, you're bound to find *one* that nobody else claimed. Once you've found a harmless NRPN and laid claim to it, multiple Data Slider (CC#6) messages can be used to create long pauses – unless your old-school keyboard already uses the Data Slider, outside of NRPN.

- **MIDI CC IDs between CC#33 and CC#50** – be careful here, these commands sometimes access lesser used functions like synthesizer envelopes and such. If they're free in your setup, though, sending them is completely harmless.

THE SELECTIVE SWITCH

There are times when it's necessary to break a connection between two MIDI devices. That thick, swirling combination of a dark synthesizer pad and classic rock organ sounds great during the verses, but it's way too muddy for the solo. Or maybe you've been "borrowing" your bandmate's harpsichord sound for a song break, but now it's time to go back to playing the piano sounds that your keyboard came with. Changing the transmit channel on the keyboard is one way to disconnect the MIDI path, but changing channels is sometimes hard to do on a keyboard with few dedicated knobs and switches.

Over the years, manufacturers have offered 'MIDI footswitches' to disconnect the cable between two MIDI keyboards or modules. These devices look and act like the A-B footswitches guitarists use to swap between two amplifiers, or between two channels of the same amp. Looks can be deceiving, though. Anyone who has used these devices knows that they have a dark side: The connection and disconnection is easy between songs, but harder to coordinate while playing. Disconnect the cable early, and sooner or later you're bound to find stuck notes, or bent pitches that no longer want to *un*-bend.

The Event Processor can create the same kind of switch, but only disconnect the MIDI messages *you* choose. That way, notes can end, volume can change, and bends can unbend, even after the channel has been disconnected. Figures 32 – 34 show how to use the Pitch Wheel as a 'valve' to connect and disconnect a second MIDI device. Settings #1 and #2 create the valve effect, while Setting #3 filters some, but not all of the note information going to Channel 8. The really slick part of this example is that the Pitch Wheel is still available for pitch bends, as long as they don't go all the way to either end of the control!

Notice that Setting #3 filters Note On events with velocity values from 1 to 127 only. Why should we keep zero velocity passing through? MIDI Note On events with zero velocity are treated as if they were Note Offs. In live performance, Note Offs are often far more important than Note Ons. This way, even if your right hand is still on a few keys when Channel 8 is disconnected, the notes will turn off when you remove your hands, regardless of whether your keyboards sends Note Offs or Note Ons with velocity = 0. We also allow Pitch Bend, and controllers like Sustain and Volume to pass though, so they will also remain active.

Note also that Setting #3 starts out in the On state, blocking Channel 8 notes. This requires a first push to the upper end of the Pitch Wheel to connect the keyboard to its slave device. We could have easily programmed it the other way. We can switch additional items like controllers, Program Change messages, or Note-Off commands in the same way. Just remember that it takes three Settings per item or range of items switched, because each Setting can only enable or disable one other Setting.

[This example is based on a real-world requirement for an Event Processor Plus. I'd like to thank my friend Eric for his novel idea for controlling the switch.]

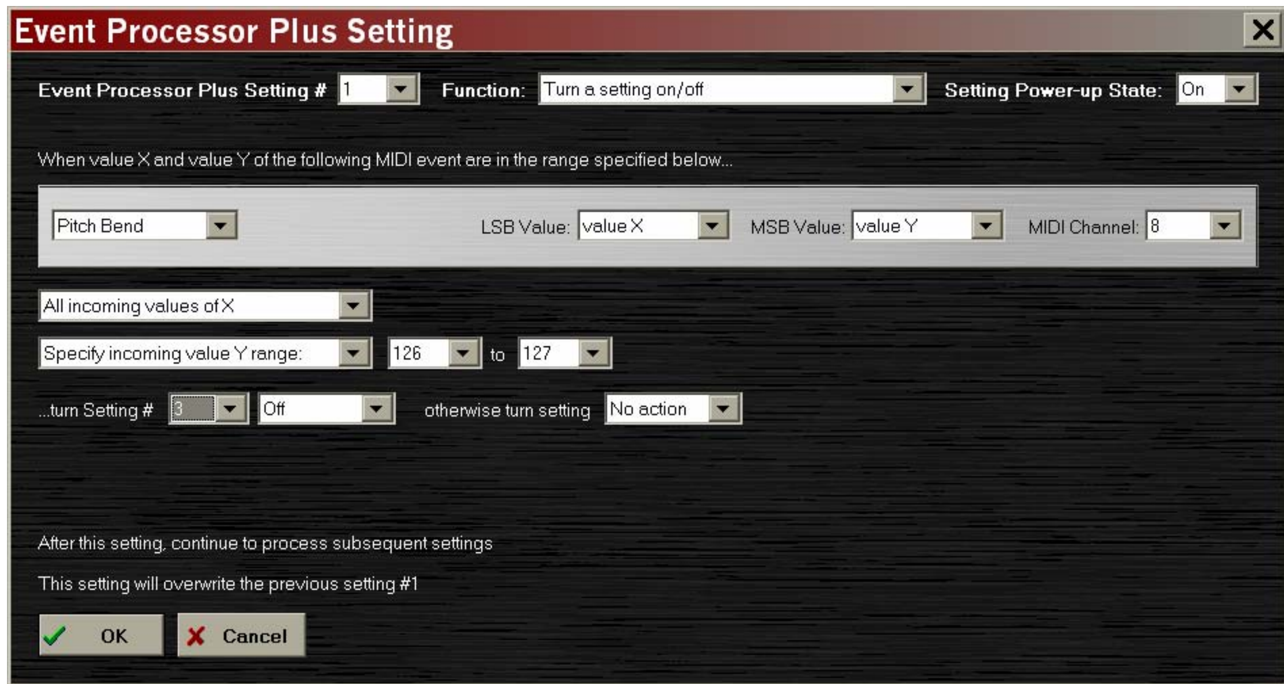


Figure 32 – Using the Pitch Wheel to Control Channel 8, part 1

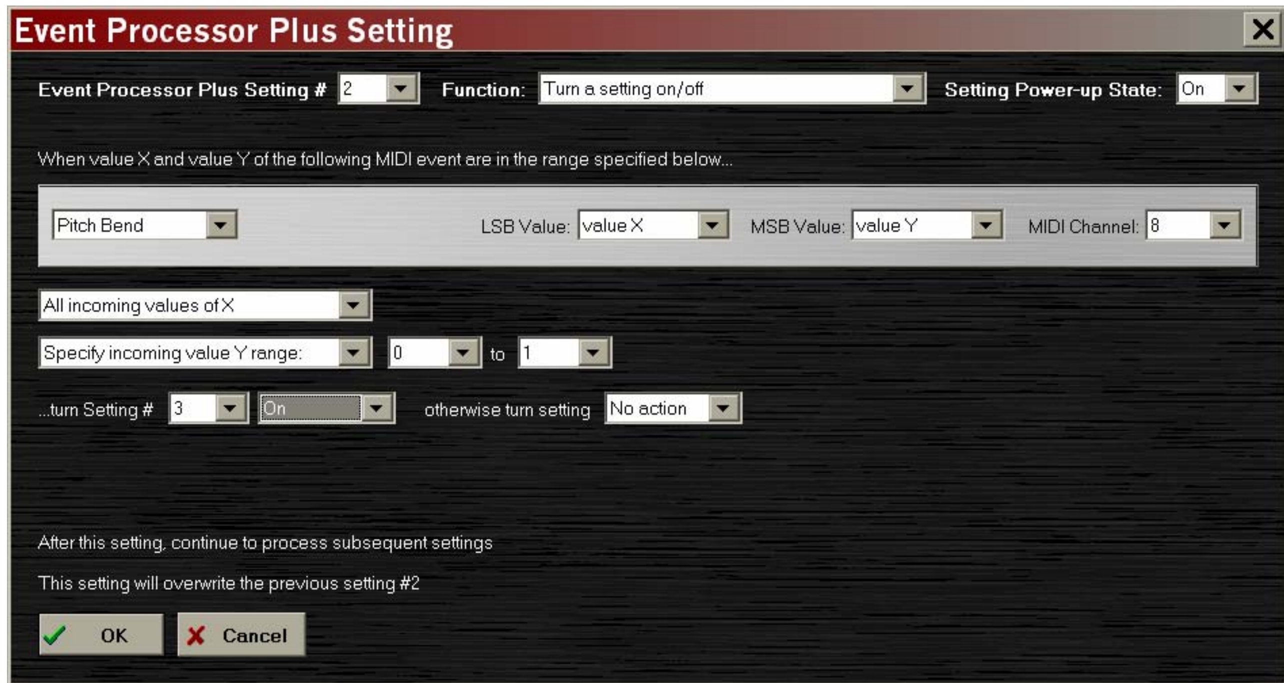


Figure 33 – Using the Pitch Wheel to Control Channel 8, part 2

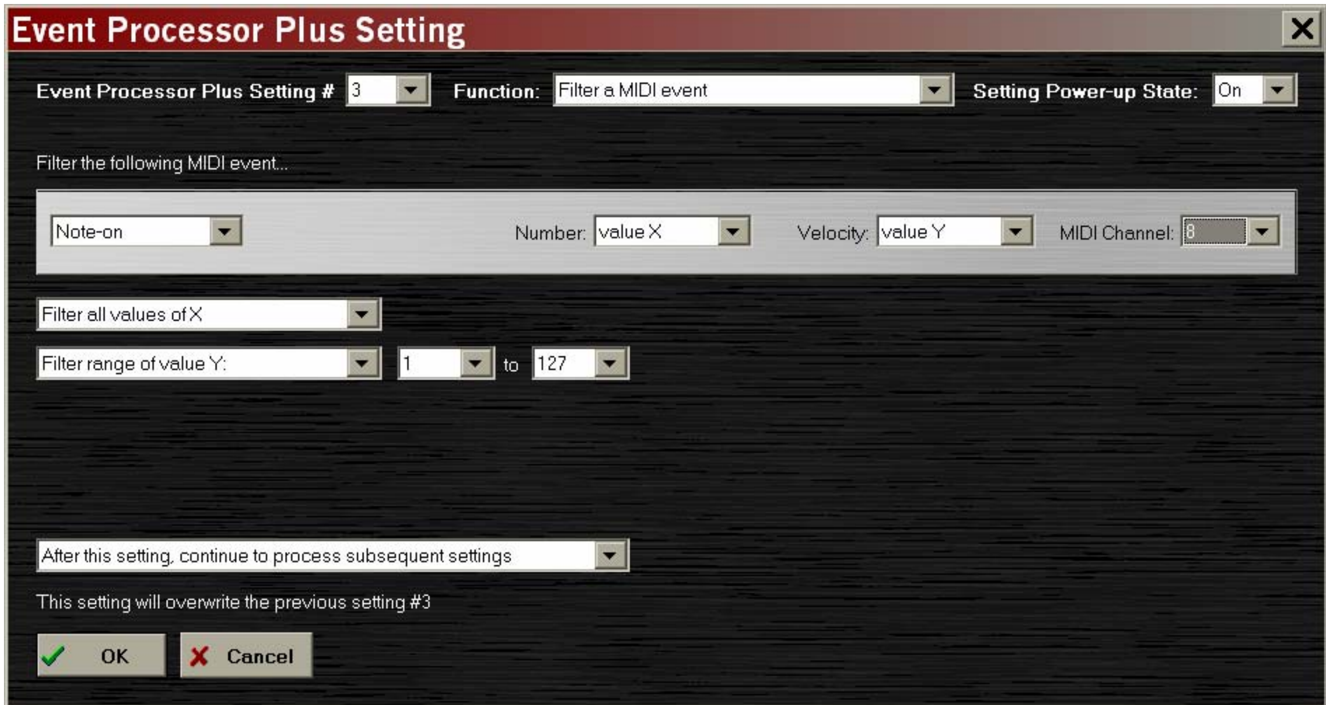


Figure 34 – Selective Filtering of Channel 8

CHAPTER 3 – LIMITATIONS AND PITFALLS

OK, by now you've probably figured out that there's nothing MIDI that the Event Processor can't deal with, right? Well, not quite. Did you ever get behind the wheel of a really powerful car or motorbike – one that had more horsepower than you ever dreamed possible? It was much easier to get from Point A to Point B; it was also much easier to end up in a ditch! Used within its designed intentions, the Event Processor can't help but improve your MIDI experience. Used outside the rules, though, it can sometimes make things very unpredictable. Here's a brief guide to understand the rules that the Event Processor *can't* break …

MIDI BANDWIDTH

MIDI runs at 31.25Kbit/sec., which works out to roughly 1,000 average MIDI commands per second. That sounds like an awful lot of time to send MIDI, but the figure can be deceiving. Certain styles of music – ex: Hammond Organ techniques like "smears" and "palm glisses" – require that a lot of notes be struck and released in a very short time, even though the average notes per second throughout the song might not be all that high. Can Joey DeFrancesco or Chester Thompson play faster than one note per millisecond for very short bursts? Sure they can. Can an average PC/Mac sequencer and an 8x8 MIDI interface up the ante? You betcha. Keep this in mind when you consider sending a merge of 4-5 MIDI outputs to one cable, and "let the devices sort it out."

The Event Processor itself is immune to failure due to fully loaded MIDI bandwidth; it's happy to send 31,250 bits per second. However, the MIDI keyboard may also enter the picture by dropping note information if it receives more notes than it can process. Dropping Note On messages isn't so bad, especially when there are dozens of them happening at the same time, but missing Note Off messages are big trouble. Some MIDI keyboards handle this better than others; it's just something to be aware of.

The one thing that can bring the Event Processor to its MIDI 'knees,' sooner or later, is ***data duplication.*** If the Event Processor is programmed to map several output commands for each one it receives, it's kind of like pouring buckets of water into a sink with a little drain pipe. The ability to empty the 'pipe' has to match the ability to fill it, or 'sink' is going to overflow. Although the Event Processor itself can have 'drainage' issues under severe cases, the real culprit is usually MIDI itself. The output of the Event Processor can't be offloaded any faster than 3,125 bytes per second. If you're playing 64th notes on Channel 1, and the Event Processor is copying each note to Channel 2-16, well, something's got to give. Even if the Event Processor can keep up with the task of duplicating the notes, they're being stockpiled faster than MIDI can use them.

The Event Processor uses built-in algorithms to try to thin the data in these cases, but anytime we throw things out, there's always a chance to throw out the wrong item. And unlike audio information, lost MIDI data sometimes has more serious consequences that a pop or click.

How can you help? Take a look at the kinds of data that your gear is sending, and whether you're using it or not. Aftertouch is a notorious hog of MIDI bandwidth, often without being used at the receiving end. Sequencer clock data is another area to watch: In many cases, the slave devices are reacting to the MIDI notes sent by the sequencer, and the clock data is simply being thrown away. One of the worst of all is Polyphonic Aftertouch, which can clog the entire bandwidth with the data from a single instrument! All of these troublesome elements can be filtered in the Event Processor.

Additionally, don't duplicate events "just in case they're needed." Save the extra Settings for later on – you'll find uses for them, I promise!

MIDI OVERFLOW

It sounds like the same thing as MIDI Bandwidth, but it's actually a totally different animal. MIDI Bandwidth is like a clogged pipe; MIDI Overflow is like having too small a bucket at the end of the pipe. MIDI Overflow occurs when the receiving device can't act on the data as fast as it comes in, even if the actual data fits in the rules of MIDI. This was a huge problem in the early days of MIDI devices, but has kind of faded as CPU power has increased. It still rears up now and then, especially with older or low-cost slave devices. Different devices react to MIDI Overflow in different ways, too: some devices thin the data; some throw out the oldest information; some reject the latest information; and some just lock up. [Ugh!]

The answer to both MIDI Bandwidth and MIDI Overflow problems usually boils down to eliminating some of the data, either by filtering it from the output of the Event Processor; or by reducing the amount of new data that the Event Processor is asked to generate.

In addition, certain kinds of data require extra time to process. This is especially true of Program Change commands, so don't try to slip one of them in between other events; send them when the MIDI data is less congested. SysEx is another potential culprit. If a particular task can be done with CC commands or SysEx, always reach for the CC approach, since it's channel-based, and requires less effort to process.

DUPLICATE SETTINGS

A Setting can be programmed many times in the Programming Tools, but can only be used *once* by the Event Processor. *The last command for each Setting number is the one that counts.* If you create six settings with the same Setting number, the first five of them are overridden by the sixth Setting, and have no effect. If you want to program the Event Processor to act on six different events, make sure that the program calls out six different Setting numbers. This sounds obvious, but it's easy to lose track of the events when programming a long series of Settings, especially in the Event Processor Plus.

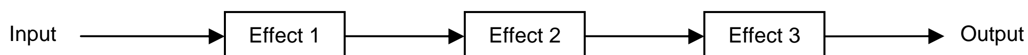
The Programming Tools software tries to help in this regard. When creating a new program using File → New, the Programming Tools start at Setting #1, and each time we click on OK, the program automatically increments to Setting #2, Setting #3, etc. There are two cases where this automatic numbering is disabled:

- When editing an existing program (or using an existing program as a template to create a new program), the Setting counter is not automatically incremented, because the software cannot be sure which Settings will be edited. Automatically jumping to the next Setting could create more problems than it solves.
- If the user manually changes the Setting number before clicking OK, the automatic numbering is disabled for the remainder of the work in the new file. The software assumes that the user has a particular reason to jump around in the Setting numbers, and makes no attempt to 'outguess' the programmer.

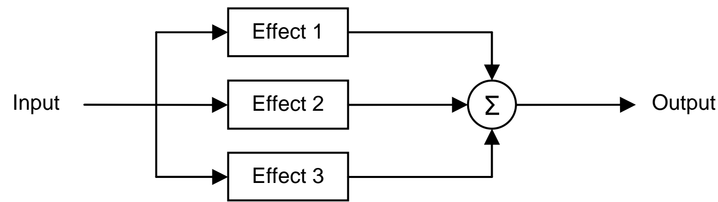
SERIAL VS. PARALLEL PROCESSING

Any processing device – MIDI or otherwise – is either a 'serial' processor, a 'parallel' processor, or some combination of the two.

- If a device takes an input and makes a change in it, then passes the changed signal to the next step, and on and on until the output, it's a *serial* processor:



- If a device takes an input signal and splits or duplicates the signal, then processes each of the duplicates before combining them back together, it's *parallel* processor:



The Event Processor is a parallel event processor. Keep this in mind when doing similar things to Events on different MIDI channels, or to different data types (notes, controllers, etc.). In most cases, each channel or data type must be processed separately by its own Event(s).

As an example, let's say that I want to re-map all CH1 notes to CH3, but only send the top five octaves of MIDI to a synthesizer on CH3. When I map CH1 to CH3 in Setting #1, I can select a range of MIDI notes that matches the five-octave range I want. However, the other notes will still pass through unmodified, because they don't apply to the setting. If I don't want these notes to show up, on CH3 *or* CH1, I need another Setting to perform the filtering on the other notes.

EVENT PROCESSING ORDER

This may be obvious to anyone who programs software for a living, but the Event Processor takes its commands in the order it sees them, period. Unlike some other MIDI Solutions boxes, the Event Processor doesn't assign a priority level to each Setting. So, it's first come, first served. When an event comes in, the first Setting on the list gets to work its magic, then the next one down the line, and so on. Most of the time, this has no bearing on how the end result works out, but there are exceptions.

In a complex group of Settings, it's possible to program the Event Processor to do one thing early in the list, only to counteract the first Setting later down the line. Two of the most important areas to watch this are 1) enabling or disabling rules; and 2) storing Variables. Once a rule is disabled, it gives up its chance to act on any data that is in process in the chain. If we turn it back on at the bottom of the list, it will act on the *next* round of data, but not on the data already in process. The same thing is true with Variables: If we act on a Variable in Setting #3, but store it in Setting #5, the action will work with the *previous* stored data, not the *current* value of the Variable.
Another point to keep in mind is that the data that leaves the Event Processor is stored in a first-in/first-out (FIFO) arrangement. If your intention is to send a Program Change, and *then* a Volume change, make sure that the Event Processor is programmed to act on events in that order. The Event Processor can't know that the Volume change at the end of the line is way more important than the Mod Wheel change you sent at the beginning; it treats all instructions as equally important.

A final point to remember is that the order of processing isn't related to the order that the instructions were entered into the Programming Tools; ***it's related to the Setting number assigned to the instruction.*** Setting #1 always gets executed before Setting #2, regardless of the way the program was written. This requirement is easy to break, because the Programming Tools actually allow Settings to be re-programmed many times in a program, with only the last instruction having any effect on the output. Be careful that the Setting with the lowest and highest numbers correspond to the first and last actions, respectively, that you want the Event Processor to take. A good practice is to always sort your program at the end, cutting and pasting commands so that they are in order. It will be easier to spot problems this way.

MISMATCHED VARIABLES AND VALUES

Generally, Value X represents the first data byte in a command, and Value Y represents the second data byte. And generally, each Value maps to the same location from in to out, X → X and Y → Y. However, there are some exceptions, especially when mapping between Program Change and Controller Change messages, because the Program Change *has* no Value Y. There are also cases where swapping the Values results in a very useful function, and is completely intentional. Just keep in mind that the Programming Tools remember the last Value used for a particular purpose, and automatically reuse the same Value the next time the same Setting type is called up. Most of the time, that's a good thing; sometimes, it's not. Pay attention to the Value use when programming your Settings.

A similar situation exists with how the Event Processor processes Variables in real time. Each Setting doesn't have its own set of Variables; the two or eight Variable locations in your Event Processor are *shared* among all of the Settings. Once Variable 1 is used to save an input value, it should not be reused until the value has been passed off to its appropriate 'output' Setting and keeping track of it is no longer necessary. If Setting #1 saves note velocity to Variable 2, and Setting #6 saves Mod Wheel position to the same Variable, the velocity data is erased. If you plan to use that velocity information in Setting #9, it's too late: the action that the Event Processor takes is now related to the Mod Wheel.

'LOSING' DATA IN THE TRANSLATION

The Event Processor can convert one type of MIDI information to another, but it can't really generate data that didn't exist in the original event. If, for example, we convert Note On messages into Program Changes, we have a *choice* of whether we want the note number (Value X) or the velocity value (Value Y) to be converted into a program number. In the opposite direction, though, the Program Change's one piece of data (program number) can't suddenly turn into both a note number and a velocity. (OK, technically it can, but the end result isn't often all that useful.) Instead, we must choose a constant output for the note number and map the velocity from the program number, or send constant velocity and map the program number as a specific note. We can also substitute one of the stored Variables for the missing parameter. We just can't gain a new parameter that didn't already exist!

NOTE NUMBERING VS. NOTE NAMING

This isn't so much of an Event Processor issue as it is a *global* MIDI situation. MIDI defines 128 notes, ranging from a C two octaves below the lowest C on an 88-note piano, to a G about 1-1/2 octaves above the High C on that same piano. It also defines Middle C to be at note #60 (3Ch). However, the earliest version of the MIDI specification didn't define any naming convention for the notes themselves. Do we call Middle C "C0"? "C3"? Something else?

Three de-facto standards arose in the industry. The first, and perhaps most common, one notes that keyboard music and notation is commonly based on the piano, and written in the key of C. Thus, it numbers the MIDI scale as "C-2" to "G8." This makes the lowest C on the piano C0. This is the notation system used in the MIDI Solutions Programming Tools. (This system is sometimes referred to as the 'Yamaha system,' because the largest maker of musical instruments, Yamaha Corporation, follows this convention.) Figure 35 shows this numbering system in greater detail.

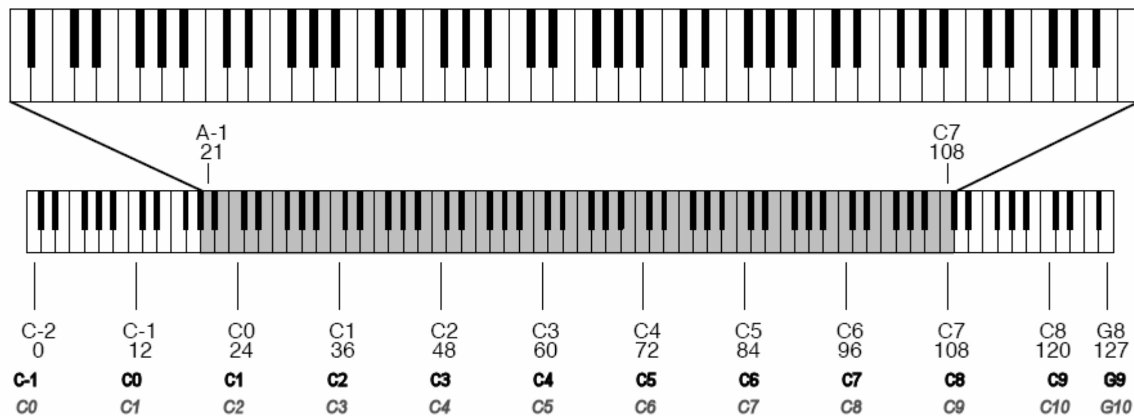


Figure 35 – Keyboard Notation Schemes

However, two other 'standards' are used by various manufacturers of MIDI hardware and software. Many keyboard manufacturers number the MIDI scale as "C-1" to "G9." This makes the lowest note on an 88-key piano A0, rather than the somewhat confusing A-1. (This notation system is sometimes referred to as the 'Roland system.') And finally, computer music programs often call the lowest C on the MIDI scale C0, thereby eliminating the entire negative-numbering convention. Both of these schemes are also shown in Figure 32.

What does this mean to you, the programmer? It means that while you can always depend on the MIDI *note number* when programming the Event Processor, the *note name* is not guaranteed. Be careful when creating keyboard zones, or when using individual Events to map, trigger, or sequence other Events. If the program doesn't seem to be doing what you expected, use a sequencer or other program to look at the actual data being sent, and compare it to the note(s) being programmed in the Event Processor.

Scaling and Round-off Errors

The Event Processor can scale data from one type to another, and can provide both compression and expansion when scaling. The Event Processor doesn't have the ability to scale into anything except integer number, though, so round-off errors are possible when scaling to or from more than two values.

As an example, scaling from a value of 34-35 on CC#12 to another CC *always* gives the correct value, because there are only two points to map: the beginning value and the end value. What happens if we map three points into a 0-127 range? Well, 128 ÷ 3 = 42.6667, so depending on whether the Event Processor rounds up or down at a particular point, the value is always going to be a little greater or less than the perfect number. The Event Processor has algorithms to minimize the impact of this problem, but be careful when depending on an exact output value when scaling.

Also, be aware that scaling 0-4 into 0-127 isn't scaling four values into 128 (which maps evenly); it's scaling *five* values, because zero counts as part of the scaling.

Buttons that Don't Send MIDI

Just as we cannot gain new MIDI parameters, we can't coax MIDI information out of a knob or switch that refuses to send any. A common example is a Split or Zone button: Pressing this button may affect future operations of the keyboard – even drastically – but the button itself probably generates no MIDI data when it's pressed. Even if you never use Zone 4 on your master keyboard, there's probably no direct way to convert the Zone 4 button into Reverb On/Off. Sorry!

Powering the Event Processor

The Event Processor can get its power from most MIDI Out or Thru jacks on keyboards, sound modules and computer MIDI interfaces. However, be aware that there is no industry standard for self-powered MIDI devices, and not every manufacturer uses the suggested MIDI interface circuits defined in the MIDI standard. If the red LED on your Event Processor refuses to light, chances are that your MIDI cable isn't providing adequate power.

There are several ways to work around this:

- Rearrange the order of your MIDI devices so that the Event Processor is connected to a device which can power it. Remember, more than one MIDI Solutions processor can be connected in series to share the power; I've found that 2-4 devices can be powered this way under almost all cases.

- Connect the device that is supposed to drive the Event Processor to another device's MIDI In port, and then connect that device's MIDI Thru port to the Event Processor. (The Thru connection produces an exact copy of the In data received.)

- Purchase a MIDI Solutions Power Adapter directly from MIDI Solutions. In addition to powering several devices, the Power Adapter allows a longer connection than 15 meters (50 feet) between MIDI devices, and improves the reliability of data in moderately long MIDI cables.

Recovering the Programming Data

Unlike many earlier MIDI Solutions devices, the Event Processor does not support dumping the contents of its memory through the MIDI Out connector. It is important to remember this, and to keep a backup copy of the programming data (the .RTF file) in a safe place. The Event Processor will not 'lose' the data once programmed, but if you want to make changes a few months down the road, it will be easier to edit the previous settings than to reprogram the Event Processor from scratch.

CLOSING THOUGHTS

Once programmed and used the way it was designed, the Event Processor is truly capable of making changes to your MIDI gear that can only be described as bordering on "magic." The limitations are few; the capabilities are enormous. Keep this guide handy, and refer back to it as you uncover new uses for your Event Processor. MIDI Solutions gear is very rugged: I've had some of my boxes for many years, and they still look and work like new.

You'll probably find that your Event Processor setup starts out kind of plain, but gets more complicated over time. Each new piece of gear that you buy will have certain traits that you really dislike, and the Event Processor will likely come to the rescue each time. This is where the extra Settings and Variables of the Event Processor Plus may make the extra expense well worth it, even if you aren't using all the capabilities at first.

Enjoy your new toy – make that, *tool.* In time, I think that you'll find that, like this author, you can't imagine *not* having an Event Processor around to set things right!

-BW
Bruce Wahler

v1.8 31 August 2005

Copyright © 2005 by Ashby Solutions. Used with permission by MIDI Solutions, Inc. This document may be freely copied and distributed in whole; sections of the document may not be used for other purposes without expressed written permission from Ashby Solutions.

MIDI Solutions, the Event Processor, and the Event Processor Plus are trademarks of MIDI Solutions, Inc. of Vancouver, BC, Canada. Ashby Solutions and the Ashby Solutions logo are trademarks of Ashby Solutions of Ashby, MA, USA. All rights reserved.