Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

So You Want to Write a Patch Editor

Sean Luke
sean@cs.gmu.edu

Technical Report GMU-CS-TR-2023-1

Abstract

A patch editor is a software tool which assists in the programming of electronic music synthesizers. Synthesizers have a long history remote programmability via a communication protocol called MIDI, but there are many complexities involved in building a general-purpose patch editor tool for a large number of machines from a wide variety of manufacturers. This document introduces the reader to how synthesizers are remotely programmed, provides a tutorial with three example synthesizers, and discusses a variety of issues and challenges involved in building patch editors.

1 Introduction

This paper discusses the issues involved in developing *patch editors* and *librarians*, software tools designed to work with electronic music synthesizers to make them easier to program and manipulate. These tools work with synthesizers using a standardized communication protocol called MIDI, which unfortunately relies on a *non-standardized* sub-protocol called System Exclusive to do much of its heavy lifting.

This paper is organized as follows. Section 2 introduces the notion of patch editors. Section 3 is a tutorial on MIDI and System Exclusive. Section 4 introduces a music synthesizer's memory model, and Section 5 discusses common interactions with a synthesizer needed to build a patch editor for it. Sections 6, 7, and 8 offer case studies for three classic synthesizers, the Yamaha DX7, the Dave Smith Instruments Prophet '08, and the Waldorf Blofeld, which illustrate how MIDI is used, how their internal models differ, and the challenges of writing software to work with each of them. Section 9 details the very wide range of absurd decisions made by manufacturers which unnecessarily complicate the development of patch editors for their products and which you will almost certainly encounter. Finally Section 10 laments recent trends in synthesizers which portend the end of tools to assist them for the benefit of the community.

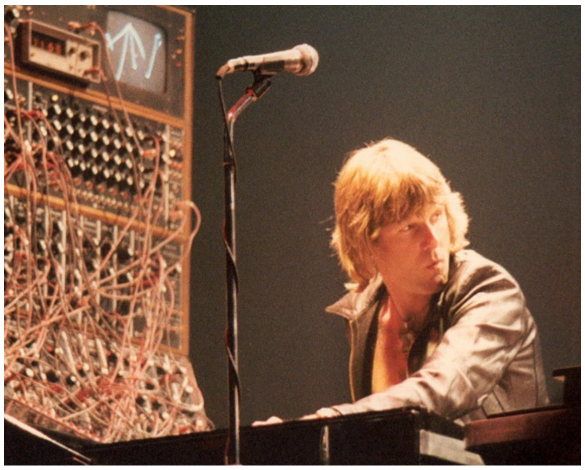


Figure 1: Moog modular synthesizer being played by Keith Emerson. Note the vertical modules with knobs and patch cables.

2 What is a Patch Editor?

A patch editor is a software tool that simplifies the programming of electronic music synthesizers and related equipment.

A *patch* is the setting of parameters which collectively program a music synthesizer to make a certain sound. The term *patch* dates from the 1960s, when early synthesizers consisted of multiple modules screwed into a rack and connected to one another via *patch cables*[1] (typically 1.5 inch tip-sleeve cables such as one might use with an electric guitar), such as in Figure 1. The various settings of knobs and buttons on the modules, plus the arrangement of patch cables connecting them, were known as a *patch* and collectively defined the programming of the synthesizer to create a certain sound. Even now, sans cables and sometimes sans knobs and buttons, a music synthesizer's program is still known as a "patch".

[1]Patch cables were originally used by early telephone operators to *patch* one phone to another to establish a phone call.

Figure 2: Dave Smith Instruments Prophet '08.

Image by Bernd Sieker, CC-BY-2.0, https://www.flickr.com/photos/pink_dispatcher/13804312464

Figure 3: Yamaha FS1R rackmount synthesizer.

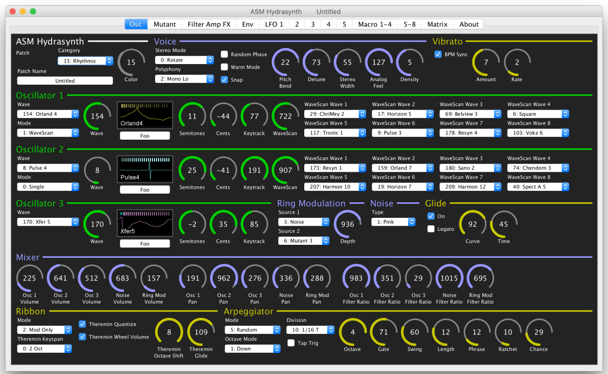
Figure 4: Oberheim Matrix 1000 rackmount synthesizer.

Some synthesizers are festooned with knobs and switches to make it easy and fun to program their patches. These synthesizers usually have relatively few patch parameters, and have a nearly one-to-one relationship between parameters and knobs/buttons, as in Figure 2. But other synthesizers have a great many more parameters and cannot be programmed this way. These synthesizers usually replace replace the knobs with a few buttons, a big jog dial or cursor keys, and a small screen, and rely heavily on menu diving to access their myriad of options. See Figure 3 for an example. These synths are very difficult to program from their front panels.[2]

An alternative to programming from the front panel is to connect a computer to the synthesizer and use a *patch editor* to program them remotely. A patch editor typically has a graphical user interface presenting virtual knobs for all of the synthesizer's parameters, plus many tools to make programming easier. Because synthesizers often hold many patches in their memory, patch editors also often sport *librarians*, which are large spreadsheets of patches designed to help a musician collect, organize, and update all the patches on a synthesizer.

The way a patch editor talks to a synthesizer is via MIDI, a standardized communication protocol for synthesizers to talk to one another and to computers. Synthesizers can be programmed remotely via MIDI: indeed some *can only* be programmed via MIDI, as in the synthesizer in Figure 4.

[2]Figure 3 is a *rackmount* synthesizer. These can be difficult to program, but are far from the only kinds of synths with this problem.



ASM Hydrasynth
Patch
Category: 15: Rhythmic
Patch Name: Untitled
Color: 15

Voice
Stereo Mode: 0: Rotate
Polyphony: 2: Mono Lo
Random Phase: off
Warm Mode: off
Snap: ✓
Pitch Bend: 22
Detune: 73
Stereo Width: 55
Analog Feel: 127
Density: 5

Vibrato
BPM Sync: ✓
Amount: 7
Rate: 2

Oscillator 1
Wave: 154: Orland 4
Mode: 1: WaveScan
Wave: 154
Orland4
Foo
Semitones: 11
Cents: -44
Keytrack: 77
WaveScan: 722
WaveScan Wave 1: 29: ChriMey 2
WaveScan Wave 2: 17: Horizon 5
WaveScan Wave 3: 69: Belview 3
WaveScan Wave 4: 6: Square
WaveScan Wave 5: 117: Tronic 1
WaveScan Wave 6: 9: Pulse 3
WaveScan Wave 7: 178: Resyn 4
WaveScan Wave 8: 103: Vokz 6

Oscillator 2
Wave: 8: Pulse 4
Mode: 0: Single
Wave: 8
Pulse4
Foo
Semitones: 25
Cents: -41
Keytrack: 191
WaveScan: 907
WaveScan Wave 1: 173: Resyn 1
WaveScan Wave 2: 159: Orland 7
WaveScan Wave 3: 180: Sano 2
WaveScan Wave 4: 74: Chendom 3
WaveScan Wave 5: 207: Harmon 10
WaveScan Wave 6: 19: Horizon 7
WaveScan Wave 7: 209: Harmon 12
WaveScan Wave 8: 40: Spect A 5

Oscillator 3
Wave: 170: Xfer 5
Wave: 170
Xfer5
Foo
Semitones: -2
Cents: 35
Keytrack: 85

Ring Modulation
Source 1: 3: Noise
Source 2: 6: Mutant 3
Depth: 936

Noise
Type: 1: Pink

Glide
On: ✓
Legato: off
Curve: 92
Time: 45

Mixer
Osc 1 Volume: 225
Osc 2 Volume: 641
Osc 3 Volume: 512
Noise Volume: 683
Ring Mod Volume: 157
Osc 1 Pan: 191
Osc 2 Pan: 962
Osc 3 Pan: 276
Noise Pan: 336
Ring Mod Pan: 288
Osc 1 Filter Ratio: 983
Osc 2 Filter Ratio: 351
Osc 3 Filter Ratio: 29
Noise Filter Ratio: 1015
Ring Mod Filter Ratio: 695

Ribbon
Mode: 2: Mod Only
Theremin Keyspan: 0: 2 Oct
Theremin Quantize: ✓
Theremin Wheel Volume: ✓
Theremin Octave Shift: 8
Theremin Glide: 109

Arpeggiator
Mode: 5: Random
Octave Mode: 1: Down
Division: 10: 1/16 T
Tap Trig: off
Octave: 4
Gate: 71
Swing: 60
Length: 12
Phrase: 12
Ratchet: 10
Chance: 29

Figure 5: Edisyn patch editor pane for the ASM Hydrasynth showing the first tab pane ("Osc": oscillators, voice and controller parameters, and the mixer). More parameters are found on additional tab panes.

Edisyn I am the author of several synthesizers and support tools. I maintain a free open source patch editor and librarian called Edisyn[3] which supports a large number of synthesizers and which serves as a good example of a patch editor and librarian tool. Edisyn is written in Java and runs on MacOS, Windows, and Linux.

Edisyn provides a patch editor for a given synthesizer in a single window with multiple tabbed panes. You can see a screenshot of an Edisyn patch editor in Figure 5. The window has a large number of widgets in it: *dials* to set numerical parameters, *checkboxes* for boolean parameters, *comboboxes* (drop-down menus) for categorical parameters, as well as a variety of *displays* to graphically explain the effects of certain parameter combinations. The widgets are organized into labelled rows called *categories*. The user interacts with Edisyn by asking it to request patches from the synthesizer, editing them and auditioning them on the machine, making parameter changes to the synthesizer in real-time via the editor, saving patches to disk, and uploading patches to the synthesizer. Edisyn has a librarian: you can download and rearrange large groups of patches as you see fit.

All this is not atypical of a patch editor. But because it must support a large number of synthesizers, Edisyn tries hard to be very general purpose and not custom for any synthesizer unless it absolutely must. This generality also enables one of Edisyn's super powers: it has a very extensive collection of randomized patch exploration tools, such as real-time morphing between different patches, blending many patches to form a new one, patch randomization and mutation, and so on. One particular tool, *evolutionary optimization* (Edisyn's "Hill-Climber") iteratively produces a random collection of patches, auditions them, gathers feedback from the user as to which he prefers, and then uses this information to produce the next collection of patches biased towards his preferences. I have published a paper on Edisyn's

[3]https://github.com/eclab/edisyn

Hill-Climber [1]. In some cases the Hill-Climber also employs a variational auto-encoder (a form of neural network) to improve the space in which the hill-climber searches for good candidates to audition.

3 About MIDI

Prior to the rise of CPUs and RAM, synthesizers were manually programmed and used proprietary methods to communicate with one another, usually for the purposes of synchronizing beats such as in drum machines. Then in 1981 at AES, Dave Smith and Chet Wood introduced what they called the *Universal Synthesizer Interface* [2]. This was later refined at the NAMM Show in January 1982, and introduced as the *Musical Instrument Digital Interface* or MIDI. The first MIDI-enabled synthesizer was Dave Smith's Sequential Prophet-600.

MIDI has since established itself as one of the longest-running, most stable, and most successful data transfer protocols in history. This was because it was well designed from the start, because it was easy to implement on devices with little memory and CPU power, and most critically, because it was an open standard. MIDI is the method by which most synthesizers and related devices communicate with one another, with keyboards and other controllers, and with computers. Patch editors send commands to synthesizers using MIDI. MIDI is specified and maintained by the MIDI Association, a nonprofit industry group.[4]

MIDI's wide adoption imposed a uniformity on how synthesizers would communicate with one another and with computers. This has made possible patch editor libraries, since large numbers of synthesizers would *more or less* present similar interfaces to the patch editor, with similar capabilities and similar protocols. Unfortunately MIDI did not go far enough in this respect, and synthesizers still vary significantly in their methods, capabilities, and documentation with respect to communicating with a patch editor. I have been asked many times why one couldn't just write a patch editor which interpreted a set of rules for each synthesizer. The answer has always been: if only it were that easy. We'll get back to that.

Design MIDI is little more than a one-direction serial port running at an unusual bitrate (31,250 bits per second). This rate was chosen because $31,250 \times 32 = 1,000,000$, making MIDI easy to clock on 1MHz CPUs.

MIDI classically runs over a 5-pin DIN serial connector, and such MIDI devices usually have three serial ports: MIDI IN, MIDI OUT, and MIDI THRU. In most cases MIDI is used by one device to instruct another device with no feedback, so all that is needed is a single cable from the upstream device to the downstream device. This cable would stretch from the upstream device's MIDI OUT port to the downstream device's MIDI IN port. When two devices need to communicate back-and-forth with one another, such as a synthesizer communicating with a patch editor on a computer, this typically requires *two* cables, each going from the MIDI IN of one device to the MIDI OUT of the other.

MIDI-enabled devices can also be daisy chained so multiple downstream devices hear the same messages from a single upstream device. This is the function of the MIDI THRU port: any information fed into the MIDI OUT port is automatically routed to the MIDI THRU port so it can be sent to second (or third, etc.) device. MIDI can also be put through a *MIDI router*, which has a single MIDI IN port and routes received data to many parallel MIDI THRU ports to go to downstream devices.

In lieu of a 5-pin DIN connector, some modern devices use a slimmer 3.5mm tip-sleeve jack. MIDI can also be run over Ethernet, bluetooth, and most importantly USB: all three of these protocols are bidirectional, so there's no need for two cables. MIDI can also be set up between two software programs in the same computer (such as a patch editor and a software synthesizer).

Data Format MIDI is a stream of bytes divided into a series of packets called *messages*. Each message starts with a *status byte* followed by zero or more *data bytes*. The status byte has its high bit set to 1, and the data bytes have their high bits set to 0, thus making it clear which are which and when the next message is arriving. This also means that the available data per byte is only 7 bits. As a result, 7-bit data is so prevalent in MIDI that a great many synthesizers only have parameters which go from 0 to 127 at most, and in the MIDI world a 7-bit data string, not an 8-bit one, is commonly known as a "byte".

There are several different categories of messages:

- *One-byte* messages consist only of the status byte. These are largely timing messages such as announcements that the timing clock has started or has pulsed, or heartbeat messages, and so on. One-byte messages are considered high priority, so much so that they are permitted to appear right in the middle of the byte strings of other messages (in a kind of "I'm gonna let you finish, but first I need to let everyone know....").

- *Two* and *three-byte* messages are the meat and potatoes of MIDI. They signify events such as requesting that a note be played or ended, or that various general-purpose parameters be modified on the downstream synthesizer. Many of these messages are *voiced*, which means they are tagged with a four-bit (0–15) *channel*[5] to specify the recipient synthesizer for which they are intended.

  Synthesizers can be set up to listen only for messages only with a given channel, ignoring other

[4]https://midi.org

[5]Perhaps nowadays this would be best thought of as an *address*.

channels: this allows multiple daisy-chained synthesizers to be controlled independently by having each one listen in on a different channel. Some synthesizers, called *multitimbral synthesizers*, can be set up to play different sounds in response to messages on different channels; and other synthesizers can be set up to listen to messages arriving on *any* channel.

- There is a single kind of *arbitrary-length* message called a *System Exclusive message*, or *Sysex message*. The purpose of a sysex message is to provide synthesizer manufacturers with an escape hatch to send data outside the bounds defined by standard MIDI messages. Most synthesizer manufacturers employ sysex to request, upload, and download patches, and so sysex messages are very important to the development of a patch editor.

There are many different MIDI messages, but when building a patch editor, five kinds of messages are particularly important, and the rest can largely be ignored. These five are *Note* messages, *Program Change* or *PC* messages, *Control Change* (or *CC*) messages, *Reserved* and *Non-Reserved Parameter Numbers* (or *RPN* and *NRPN*), and of course *System Exclusive* (or *Sysex*) messages. We'll discuss each of them next.

Note Messages A patch editor will likely need to send test notes to a synthesizer. Note messages take the form of *Note On* and *Note Off*. A Note On message asks the synthesizer to start playing a given note. To send a Note On message on channel *X*, you send a status byte of the form `0x9`*X*, followed by a data byte (0–127) indicating the *note number* (middle C is `0x60`), followed by a second data byte (0–127) indicating the *note velocity*, which is how fast the key was struck and so correlates with volume. If you have no velocity information, send `0x40`.

If you are sending a stream of Note On messages in a row, you can omit the status byte of all but the first message: this compression scheme, called *running status*, is helpful because MIDI is not particularly fast.

A Note Off message asks the synthesizer to stop playing a given note, and is paired with a prior Note On message. To send a Note Off message on channel *X*, you send a status byte of the form `0x8`*X*, followed by a data byte (0–127) indicating the *note number*, followed by a second data byte (0–127) indicating the *release velocity*, which is how fast the key was released. If you have no release velocity information, send `0x40`. A series of Note Off messages can also take advantage of running status.

If you don't care about release velocity, then instead of sending a Note Off message, you can send a Note On message with a note velocity of `0x00`. This is interpreted as Note Off with release velocity `0x40`, and makes it possible for you to only send a stream of Note On messages, which better takes advantage of running status.

PC Messages A *Program Change* or *PC* message instructs the synthesizer to load a new patch. To send this instruction on channel *X*, first send the status byte `0xC`*X*, followed by a data byte (0–127) indicating the *patch number* to load.

This scheme was devised at a time when it was rare for a synthesizer to have more than 128 patches stored in RAM, but this is no longer the case. It is now conventional for a synthesizer to divide its patches into some *N banks* of up to 128 patches each. In a modern synthesizer, to load a patch you'd first send a *Bank Select* request (see *CC Messages* below) to specify the bank, followed by a Program Change to select the patch in the bank.

CC Messages Synthesizers are historically covered with knobs and switches which control their parameters. To change these parameters remotely — a very important task for a patch editor — MIDI provides so-called *Control Change* or *CC* messages. A CC message on channel *X* consists of a status byte of the form `0xB`*X*, followed by a data byte (0–127) indicating the *parameter number*, followed by a second data byte (0–127) indicating the *parameter value*. CC messages are a weakness in MIDI because there are only 128 CC parameters, and because each parameter can have only 128 different values. Many synthesizers have far more than 128 parameters, and many such parameters (famously filter cutoff or filter resonance) require much, much higher resolution than 128 to avoid stepping and generally sounding awful.

An early attempt to fix this problem was so-called *14-bit CC*. The idea was to send two CC messages in series. CC parameters 32 through 63 could be paired with their corresponding parameters 0 through 31 to form 14-bit values. CC number n (< 32) defined the *Most Significant Byte* or *MSB* of the parameter, and CC number $n + 32$ defined the *Least Significant Byte* or *LSB*. The parameter value was then interpreted as $MSB \times 128 + LSB$.

For example, a synthesizer might require a 14 bit number to describe its volume, and so to set it to, say, 3006, you'd send CC parameter 7 with the value 23, and CC parameter $32 + 7 = 39$ with the value 62, since $23 \times 128 + 62 = 3006$. The LSB and MSB messages can be in any order and either can be missing. If an LSB is received, the last-sent MSB value is used (or a default if the MSB had never been sent yet); and likewise if an MSB is received, the last-sent LSB value is used. Because this might temporarily produce the wrong value, it is recommended that after receiving an MSB, a device should wait for just a bit to see if a new accompanying LSB arrives. 14-bit CC is not particularly common but does show up in some synthesizers.

A synthesizer can interpret CC messages in any way it wishes, but there are many conventions. For discussion here, the most important one is sending a *Bank Select* message, which by convention is either CC parameter 0 or, in some cases 32. As discussed in *PC Messages* above, a bank select message sets the bank from which a PC changes the patch number. The bank in question is the CC message value. There exist a few synthesizers which require more than 128 banks; in this case you'd use 14-bit CC to perform the bank select: first you'd send the MSB of the bank in parameter 0, and then the LSB in parameter 32.

RPN and NRPN Plain old CC messages are problematic because (1) there are only 128 of them — and modern synthesizers have many more than 128 parameters — and (2) each message can only have 128 values, which is too low a resolution for many parameters. The 14-bit CC convention fixes the second problem, but it makes the first problem even worse because it repurposes up to 32 CC parameters to be the LSB pair of the MSB of 32 other parameters. As a result, 14-bit CC is rarely used. We need a scheme which allows for many high-resolution parameters. Enter *RPN* and *NRPN*, which stand for *Reserved* and *Non-Reserved Parameter Numbers* respectively.

RPN and NRPN are *groups* of CC messages which together define a high-resolution parameter and its high resolution value. RPN and NRPN are exactly the same except that they define two different spaces of parameters. The RPN space is reserved for the MIDI Association to use for official functions, while the NRPN space is set aside for synthesizer manufacturers to do with as they wish. We will focus on NRPN.

In NRPN, you first tell the synthesizer the NRPN parameter number you wish to change, and then you tell the synthesizer what value to change it to. The parameter number is 14-bit, meaning that it can be any number 0–16383. You will send this as two CC messages. You will send the MSB of the parameter as the value of a CC 99 message: and you will send the LSB of the parameter as the value of a CC 98 message. Together the parameter number $n = MSB \times 128 + LSB$.

Once you have set the parameter number by sending these two messages, you can then send a *stream* of CC messages repeatedly changing the *value* of the parameter. This value is also 14 bits (0–16383), and so has two CC messages: CC 6 specifies the MSB of the value, and CC 38 specifies the LSB of the value. As usual, the value $v = MSB \times 128 + LSB$. As was the case for the 14-bit CC convention, the MSB and LSB can be sent in either order and either can be missing. If an LSB is received, the last-sent MSB value is used (or a default if the MSB had never been sent yet); and likewise if an MSB is received, the last-sent LSB value is used. Because this might temporarily produce the wrong value, it is recommended that after receiving an MSB, a device should wait for just a bit to see if a new accompanying LSB arrives.

There are two interpretations of NRPN MSB and LSB. First, they are often interpreted together as the integer $MSB \times 128 + LSB$. But the spec suggests that the MSB might be interpreted as an integer value and the LSB be interpreted as a "fine tune" value, suggesting $MSB + LSB/128.0$. This distinction matters because some devices use NRPN for parameters whose range is < 128. In the first interpretation, sometimes called the "fine" interpretation, the parameter value would be sent solely with a single LSB (the MSB would be assumed to be 0). The second interpretation, sometimes called the "coarse" interpretation, is exactly the opposite. Unfortunately, the standard does not specify which interpretation should be used: I strongly prefer the "fine" interpretation, but different manufacturers do things differently.

Another way to set NRPN values is to use the *Increment* and *Decrement* messages. The Increment message is sent with a CC 96 and the Decrement message is sent with CC 97: and their value is the amount to increment or decrement by. These messages are not used very often, and if they are, the value is often ignored and the synthesizer just increments or decrements by 1.

RPN works just like NRPN except the parameter LSB and MSB CCs are 100 and 101 respectively. NRPN and RPN are just a convention: synthesizers are free to ignore them and repurpose the CCs 6, 38, 96, 97, 98, 99, 100, and 101 for their own purposes (and often do). RPN and NRPN are more expressive than simple CCs, but they are significantly slower as they involve sending several CC messages to express a single parameter change. It is very common for synthesizers to use *both* ordinary CC messages *and* RPN/NRPN.

Sysex Messages CC and NRPN are useful for sending individual parameters, and PC for changing patches, but often synthesizers need to do much more than this, and often it involves much larger chunks of data. A synthesizer might wish to efficiently dump to a computer or another synthesizer the entire contents of a patch, or all the patches in a bank, or all the patches stored in memory. A synthesizer might need to respond to a *request* to dump a patch, or to change a mode in some special way, or to load a sound sample, or to customize the tuning of all its keys. Early MIDI synthesizers also needed a way to change parameters better than CC afforded in the age before RPN and NRPN. MIDI was silent on how to do any of this: all it provided was an "escape hatch" to let manufacturers design their own proprietary protocols on top of MIDI. This escape hatch is *System Exclusive* (or *Sysex*) messages.

A Sysex message is arbitrarily long. It starts with the status byte `0xF0`, followed by one or three data bytes which define the manufacturer's *MIDI Manufacturer ID*. Manufacturers receive an ID after registering with the MIDI Association: the ID `0x7D` can also be used for personal use, and `0x7E` and `0x7F` are reserved by the MIDI Association for standardized protocols.

After sending the ID, the manufacturer is free to send as many data bytes as he wishes and in whatever format he likes. Different manufacturers concoct different protocols which are unfortunately not very consistent with one another (or even internally consistent). A Sysex message technically doesn't have to be terminated, but it almost always is, using another status byte, `0xF7`.

For more information about MIDI, see my text *Computational Music Synthesis* [3], Chapter 11.

4 A Synthesizer's Patch Model

Almost all synthesizers on the market use exactly the same model for their patches: a patch is a fixed-length array of integers with varying ranges per-parameter. For example, an Oberheim Matrix 6 or 1000 patch consists of the following 134 numerical parameters:

name0	portamentolegato	env2mode	vcffmenv3mod
name1	lfo1speed	env3triggermode	vcffmpressuremod
name2	lfo1trigger	env3delay	lfo1speedmod
name3	lfo1lag	env3attack	lfo2speedmod
name4	lfo1shape	env3decay	mod1source
name5	lfo1retrigger	env3sustain	mod1amount
name6	lfo1source	env3release	mod1destination
name7	lfo1amplitude	env3amplitude	mod2source
keyboardmode	lfo2speed	env3lfotriggermode	mod2amount
dco1frequency	lfo2trigger	env3mode	mod2destination
dco1shape	lfo2lag	trackingsource	mod3source
dco1pulsewidth	lfo2shape	trackingpoint1	mod3amount
dco1fixedmods1	lfo2retrigger	trackingpoint2	mod3destination
dco1waveenable	lfo2source	trackingpoint3	mod4source
dco2frequency	lfo2amplitude	trackingpoint4	mod4amount
dco2shape	env1triggermode	trackingpoint5	mod4destination
dco2pulsewidth	env1delay	ramp1rate	mod5source
dco2fixedmods1	env1attack	ramp1mode	mod5amount
dco2waveenable	env1decay	ramp2rate	mod5destination
dco2detune	env1sustain	ramp2mode	mod6source
mix	env1release	dco1frequencymod	mod6amount
dco1fixedmods2	env1amplitude	dco1pulsewidthmod	mod6destination
dco1click	env1release	dco2frequencymod	mod7source
dco2fixedmods2	env1amplitude	dco2pulsewidthmod	mod7amount
dco2click	env1lfotriggermode	vcffrequencyenv1mod	mod7destination
dco1sync	env1mode	vcffrequencypressuremod	mod8source
vcffrequency	env2triggermode	vca1velmod	mod8amount
vcfresonance	env2delay	vca2env2mod	mod8destination
vcffixedmods1	env2attack	env1amplitudemod	mod9source
vcffixedmods2	env2decay	env2amplitudemod	mod9amount
vcffm	env2sustain	env3amplitudemod	mod9destination
vca1	env2release	lfo1amplitudemod	mod10source
portamento	env2amplitude	lfo2amplitudemod	mod10amount
portamentomode	env2lfotriggermode	portamentomod	mod10destination

(These are the names Edisyn gives to these parameters.) Some of these parameters, like `name0`, are integers representing ASCII characters. Others, like `dco1pulsewidth`, are integers in some range such as 0–127. Some parameters, like `dco1sync`, are integers representing a set of categorical values (in this case, the values 0, 1, 2, 3, representing `Off`, `Soft`, `Medium`, and `Hard`), while others, like `dco2click`, represent booleans with 0 and 1.

The number and ranges of the parameters vary from synthesizer to synthesizer, but usually the array length is fixed. Even in the rare cases where the number of parameters varies, the structure is still just an array.

This is convenient because it allows for a consistent interface. In Edisyn's case, the interface largely consists of knobs for numerical parameters, drop-down menus (comboboxes) for categorical parameters, and checkboxes for boolean parameters. Edisyn tends to gather name parameters into a single String parameter called *name*, and it adds two more parameters as appropriate: *patch* and *bank*. The synth's parameter values, plus the values for *name, patch,* and *bank*, are stored in a hash table keyed by parameter name. Because of this consistency among synthesizers, with some care Edisyn can treat a patch as a point in a multidimensional space and so employ various techniques to wander about the space in search of new or better patches.

Some synthesizers have linkages among parameters which complicate matters. For example, setting one parameter to a certain value might change the available options or range of one or more other parameters. This particularly happens in synthesizers with built-in effects, where setting an effect type might change the options for several effects parameters.

Multitimbral Patches A *monophonic* synthesizer can only play a note at a time: we call the sound produced in response to a request to play a note a *voice*. Modern *polyphonic* synthesizers can play many *voices* (hence many notes) at once, enabling chords or polyphony.

But synthesizers with a very high voice count can often do more than this: they can essentially divide themselves up into several synthesizers, each playing its own different patch, and divvy their voices up among these "synthesizers". Perhaps we might split the synth's keyboard range among two "synthesizers"; or perhaps up to 16 of these "synthesizers" would each respond to different MIDI channels. This would allow a synthesizer to play a bass, a lead, a pad, and a drum pattern, all inside the same box in response to instructions from a remote computer. We call these *multitimbral* synthesizers.

Synthesizers handle multitimbrality by having several different types of patches. Typically the synthesizer would store, say, 128 standard patches in its memory, now called *single-mode* patches, and would also have, say, 64 patches of a different type called *multimode* patches. A multimode patch might contain *N parts*, each a slot holding a *reference* to a different single-mode patch, plus information on how that patch is to be used, such as its MIDI channel, whether it responds to a region on the keyboard, how loud it is relative to the others, how it is panned, how many voices are allocated to it, and so on.

Synthesizers might have additional kinds of patches as well. For example, many ROM sample-based synthesizers from the 1990s sported a *drum kit patch,* with one drum sound per key on the keyboard. This drum kit could respond to MIDI (usually channel 10) and could be played along with multitimbral collections of patches. A few synthesizers, such as the Kawai K4, also had *effects patches* which defined collections of effects that could be referenced by single-mode patches. Waldorf synthesizers had *wavetable patches* and *wave patches*. The Korg Wavestation had *wave sequence* patches. The Yamaha FS1R had *Fseq patches*. All of these might be referenced by single-mode patches. In short: it's common for synthesizers to have patches which refer to other patches stored in RAM, thus producing a kind of patch hierarchy.

As a result, a synthesizer cannot be thought as having a single kind of patch, but multiple types of patches, each requiring its own patch editor. Some patch editors try to gloss over this with a unified patch editor interface, but this does a disservice to the musician, who surely knows that his synthesizer has two or more kinds of patches inside.

Synthesizers themselves also try to gloss over the issue with their own strange interfaces. Most multitimbral synthesizers either allow a user to play a single patch (as if the synthesizer was merely polyphonic) or to put the synthesizer into a multimode arrangement, or to play a drum patch etc. But some of them try to present a hierarchical arrangement: you can only have the synthesizer in "multimode", and to play a single patch you must turn off all multimode patch parts but one. This is both deceitful and very inconvenient and confusing, but there you have it. These synths still usually present separate patches via their MIDI and Sysex interfaces thankfully.

Permanent Storage and Current Working Memory Synthesizers typically store collections of patches in battery-backed RAM or on Flash RAM: but only certain patches are used at any given time. Let's say a synthesizer is playing in single-patch mode. When the musician requests it (either from the front panel or via a bank select / program change over MIDI) the synthesizer will *load* a copy of the patch from patch RAM to its current working memory. There the patch can be played, edited and modified, and if so desired, ultimately saved back to patch RAM, overwriting the original version. If the synthesizer is in multimode, the current working memory likely would hold both one multimode patch and several single-mode patches to which the multimode patch is referring.

It's also common for some collections of patches to be stored in permanent ROM (so they can be loaded into current working memory but not saved from it). Patches may also be stored on removable cartridges or SIMM cards, either as RAM or ROM. Cartridges or SIMM cards can hold other elements as well: for example, they might hold basic sound samples from which a patch can be constructed, and so a patch might have to refer to an underlying sound in its parameters not only via its sound number but also the particular card on which its found.

5 Common Actions

In order to control a synthesizer, an editor must be able to do send it a variety of demands. Here is a list of the ones Edisyn often relies on:

- Play a note
- Stop playing a note
- Stop playing all notes. This is often achieved with CC 120 or 123.[6]
- Change mode, such as from multimode to single-mode (typically a Sysex message).
- Switch to a different patch stored on the synthesizer (typically done with one or more Bank Select CCs followed by a PC, but sometimes requiring Sysex).
- Update a single parameter to a new value in current working memory (often via CC, NRPN, or Sysex).
- Update all parameters to new values in current working memory. This can be done by updating each parameter in turn, but this is inefficient, so it is common to have a special Sysex command, called a *patch dump*, to send the data. Edisyn calls this *sending a patch*.
- Load a patch from current working memory. The request to do this would be a sysex message called a *dump request*, and the synthesizer would respond with a patch dump sysex message to provide the data.
- Load a patch from RAM. Again, this would be another kind of dump request sysex message, and the synthesizer would again respond with a patch dump sysex message to provide the data.
- Store a patch directly in RAM. This would be a patch dump sysex message providing the synthesizer with the data. Edisyn calls this *writing a patch*.
- Load an entire bank from RAM, or all banks from RAM. This would be yet another kind of dump request sysex message, and the synthesizer might respond by emitting each patch separately or via a special large bank- or multi-bank-dump sysex message.
- Store an entire bank to RAM, or to all banks in RAM. This would be a similar large bank- or multi-bank-dump sysex message. Usually Edisyn tries to write individual patches rather than implement this message.

A patch editor should also be able to respond to data offered spontaneously by a synthesizer, such as:

- A manual update to a parameter in current working memory (typically sent as CC, NRPN, or Sysex). Some synthesizers provide this feature; others do not, particularly synthesizers with poor interfaces.
- A patch dump from RAM or from current working memory.
- A dump of an entire bank or multiple banks as a special sysex message.

[6]CC 120 with value 0 is customarily the *All Sounds Off* message. This instructs the synthesizer to cut all sound immediately and reset all of its voices. CC 123 with value 0 is customarily the *All Notes Off* message. This tells the synthesizer to stop playing all notes: as a result some notes may continue to make sound as they fade away. Some early synthesizers support one but not the other message. And at least one synthesizer (the ASM Hydrasynth) not only ignores CC 120, but has actually reallocated 120 for another purpose.

Finally a patch editor should also be able to pass along to the synthesizer MIDI data from a *controller* such as a MIDI keyboard being played by the musician. This would allow the musician to play the synthesizer to test a patch he had created. Data might include:

- Play a note
- Stop playing a note
- Update a single parameter to a new value in current working memory (usually via CC or NRPN, not Sysex).

Synthesizers can and do sport or require additional unusual sysex messages and commands to manipulate them; but the above are the most common. Note that many synthesizers only support a subset of these actions. Table 1 shows the broad and unfortunate diversity of capabilities of synthesizers with respect to certain common commands. In some cases we can work around it: for example in lieu of updating all parameters, we might update each parameter individually. But in other cases there is no way around it and the patch editor simply cannot provide that capability. For example, many E-Mu synthesizers inexplicably cannot respond to requests to dump a patch from current working memory. The famed Yamaha DX7 has no ability to load a patch into RAM: it can only load entire banks of patches into RAM. And so on. Synthesizers vary greatly here, with some synthesizers capable of a great many actions, while others are hobbled with terribly limited MIDI capabilities.

	Send Parameter	Receive Parameter	Request Specific Patch	Request Current Patch	Send to Current Patch	Write to Specific Patch	Change Mode	Receive Error or Ack	Standard Sysex File
Alesis D4/DM5	✓*		✓		✓	✓			✓
Audiothingies Micromonsta	✓	✓				✓*			✓
Casio CZ Family			✓	✓	✓	✓			✓*
DSI Prophet '08/Mopho/Tetra	✓	✓	✓	✓	✓	✓			✓
E-Mu Morpheus/UltraProteus	✓		✓			✓			✓
E-Mu Proteus Family	✓		✓			✓			✓
E-Mu Proteus 2000 Family	✓	✓	✓	✓	✓	✓		✓	✓
E-Mu Planet Phatt/Orbit/Carnaval/Vintage Keys	✓		✓			✓			✓
JL Cooper MSB Plus/Plus Rev2			✓	✓		✓			✓
Kawai K1/K1r/K1m	✓*		✓			✓		✓	✓
Kawai K4/K4r	✓*		✓		✓	✓		✓	✓
Kawai K5/K5m	✓	✓	✓			✓		✓	✓
Korg Microsampler	✓*	✓*							
Korg SG Rack			✓	✓	✓	✓	✓		✓
Korg Volca Series	✓*								
Korg Wavestation SR	✓		✓	✓*	✓*	✓	✓*		✓
M-Audio Venom	✓		✓*	✓	✓*	✓*	✓*	✓	✓*
Novation Drumstation/D Station					✓*	✓*			✓
Novation SL Family					✓	✓			✓*
Oberheim Matrix 6	✓*	✓	✓	✓		✓			✓
Oberheim Matrix 1000	✓	✓	✓	✓	✓	✓			✓
PreenFM2	✓	✓	✓	✓	✓				
Red Sound DarkStar/DarkStar XP2					✓	✓			✓
Roland D-110	✓		✓*	✓*	✓*	✓*			✓
Roland JV-80/880	✓		✓	✓	✓	✓			✓*
Roland U-110	✓			✓	✓	✓			✓*
Roland U-20/220	✓		✓	✓	✓	✓			✓*
Sequential Prophet Rev2	✓	✓	✓	✓	✓	✓			✓
Waldorf Microwave II/XT/XTk	✓	✓	✓	✓*	✓	✓	✓		✓
Waldorf Blofeld	✓*	✓*	✓	✓*	✓	✓			✓
Waldorf Kyra	✓	✓	✓*	✓*	✓*	✓*			✓*
Waldorf Pulse 2	✓	✓	✓	✓	✓	✓			✓
Waldorf Rocket	✓	✓		✓*	✓*				
Yamaha DX7 Family	✓	✓	✓*	✓					✓
Yamaha 4-Op Family	✓*		✓	✓	✓				✓*
Yamaha FB-01	✓*		✓	✓	✓	✓*		✓	✓*
Yamaha FS1R	✓	✓	✓	✓		✓			✓
Yamaha TG33/SY22/SY35	✓*		✓	✓	✓				✓

* *With significant caveats or restrictions*

Table 1: Transfer Capabilities of Some Synthesizers (Edisyn patch editors as of January 2023)

6 Case Study: Yamaha DX7

The Yamaha DX7 is one of the most successful synthesizers in history, and certainly it is the synthesizer which changed the industry the most. Introduced at a time when the market was entirely analog synthesizers, the DX7's digital Frequency Modulation (FM) synthesis approach was so successful it caused the analog market to entirely collapse. If you've heard *any* music from the 1980s or early 1990s, you've heard the DX7: it's the sound of the 80s. The DX7 spawned a large number of FM synthesizers and its approach is still very popular, living on in open source replicas such as Dexed.[7] The DX7 came in keyboard, desktop, and rackmount variations. Figure 6 shows a DX7 keyboard.

The DX7 has a single bank of 32 patches, plus an optional cartridge which provides up to two more banks of 32 patches each. The current bank being used is specified by pressing buttons on the unit itself. You cannot upload or download a patch to a bank: you must send or receive all of them in bulk. However you can upload a single patch to current working memory. You can also update individual parameters.

[7]https://asb2m10.github.io/dexed/

Below I discuss the general capabilities of the DX7 with regard to working with a patch editor (and some of its failures).

Updating Individual Parameters in Working Memory This is done with the sysex string

`0xF0 0x43` *substatus(0x70)+channel MSB LSB value* `0xF7`

Let's go through of these bytes in turn. All sysex messages start with `0xF0` and are terminated with the start of some other message, but usually `0xF7`. Following `0xF0` is `0x43`, which is the manufacturer ID for Yamaha. Manufacturer IDs are one byte for early manufacturers and certain special cases, and three bytes for later ones.[8]

[8]See https://github.com/eclab/edisyn/blob/master/edisyn/Manufacturers.txt for a list of manufacturer IDs.

Figure 6: Yamaha DX7.

Steve Sims, Public Domain, https://commons.wikimedia.org/w/index.php?curid=12379187

0	operator6rate1	77	operator3amplitudemodulationsensitivity
1	operator6rate2	78	operator3keyvelocitysensitivity
2	operator6rate3	79	operator3operatoroutputlevel
3	operator6rate4	80	operator3oscillatormode
4	operator6level1	81	operator3frequencycoarse
5	operator6level2	82	operator3frequencyfine
6	operator6level3	83	operator3frequencydetune
7	operator6level4	84	operator2rate1
8	operator6keyboardlevelscalingbreakpoint	85	operator2rate2
9	operator6keyboardlevelscalingleftdepth	86	operator2rate3
10	operator6keyboardlevelscalingrightdepth	87	operator2rate4
11	operator6keyboardlevelscalingleftcurve	88	operator2level1
12	operator6keyboardlevelscalingrightcurve	89	operator2level2
13	operator6keyboardratescaling	90	operator2level3
14	operator6amplitudemodulationsensitivity	91	operator2level4
15	operator6keyvelocitysensitivity	92	operator2keyboardlevelscalingbreakpoint
16	operator6operatoroutputlevel	93	operator2keyboardlevelscalingleftdepth
17	operator6oscillatormode	94	operator2keyboardlevelscalingrightdepth
18	operator6frequencycoarse	95	operator2keyboardlevelscalingleftcurve
19	operator6frequencyfine	96	operator2keyboardlevelscalingrightcurve
20	operator6frequencydetune	97	operator2keyboardratescaling
21	operator5rate1	98	operator2amplitudemodulationsensitivity
22	operator5rate2	99	operator2keyvelocitysensitivity
23	operator5rate3	100	operator2operatoroutputlevel
24	operator5rate4	101	operator2oscillatormode
25	operator5level1	102	operator2frequencycoarse
26	operator5level2	103	operator2frequencyfine
27	operator5level3	104	operator2frequencydetune
28	operator5level4	105	operator1rate1
29	operator5keyboardlevelscalingbreakpoint	106	operator1rate2
30	operator5keyboardlevelscalingleftdepth	107	operator1rate3
31	operator5keyboardlevelscalingrightdepth	108	operator1rate4
32	operator5keyboardlevelscalingleftcurve	109	operator1level1
33	operator5keyboardlevelscalingrightcurve	110	operator1level2
34	operator5keyboardratescaling	111	operator1level3
35	operator5amplitudemodulationsensitivity	112	operator1level4
36	operator5keyvelocitysensitivity	113	operator1keyboardlevelscalingbreakpoint
37	operator5operatoroutputlevel	114	operator1keyboardlevelscalingleftdepth
38	operator5oscillatormode	115	operator1keyboardlevelscalingrightdepth
39	operator5frequencycoarse	116	operator1keyboardlevelscalingleftcurve
40	operator5frequencyfine	117	operator1keyboardlevelscalingrightcurve
41	operator5frequencydetune	118	operator1keyboardratescaling
42	operator4rate1	119	operator1amplitudemodulationsensitivity
43	operator4rate2	120	operator1keyvelocitysensitivity
44	operator4rate3	121	operator1operatoroutputlevel
45	operator4rate4	122	operator1oscillatormode
46	operator4level1	123	operator1frequencycoarse
47	operator4level2	124	operator1frequencyfine
48	operator4level3	125	operator1frequencydetune
49	operator4level4	126	pitchegrate1
50	operator4keyboardlevelscalingbreakpoint	127	pitchegrate2
51	operator4keyboardlevelscalingleftdepth	128	pitchegrate3
52	operator4keyboardlevelscalingrightdepth	129	pitchegrate4
53	operator4keyboardlevelscalingleftcurve	130	pitcheglevel1
54	operator4keyboardlevelscalingrightcurve	131	pitcheglevel2
55	operator4keyboardratescaling	132	pitcheglevel3
56	operator4amplitudemodulationsensitivity	133	pitcheglevel4
57	operator4keyvelocitysensitivity	134	algorithm
58	operator4operatoroutputlevel	135	feedback
59	operator4oscillatormode	136	oscillatorkeysync
60	operator4frequencycoarse	137	lfospeed
61	operator4frequencyfine	138	lfodelay
62	operator4frequencydetune	139	lfopitchmodulationdepth
63	operator3rate1	140	lfoamplitudemodulationdepth
64	operator3rate2	141	lfokeysync
65	operator3rate3	142	lfowave
66	operator3rate4	143	lfopitchmodulationsensitivity
67	operator3level1	144	transpose
68	operator3level2	145	name1
69	operator3level3	146	name2
70	operator3level4	147	name3
71	operator3keyboardlevelscalingbreakpoint	148	name4
72	operator3keyboardlevelscalingleftdepth	149	name5
73	operator3keyboardlevelscalingrightdepth	150	name6
74	operator3keyboardlevelscalingleftcurve	151	name7
75	operator3keyboardlevelscalingrightcurve	152	name8
76	operator3keyboardratescaling	153	name9
		154	name10

Table 2: Yamaha DX7 Parameters (using Edisyn names). Note that many parameters oddly are in reverse order from what one would expect.

Thus the string `0xF0 0x43` defines a namespace that only Yamaha synthesizers will respond to.

Yamaha made a design error at this point: it didn't add a byte differentiating the DX7 from other synthesizers. Thus Yamaha synthesizers can't tell if this message is meant for a DX7 or for some other Yamaha synth (such as themselves). Most manufacturers tack on a byte here to indicate which model this message is meant for.

Next comes Yamaha's so-called "substatus" and channel. The "substatus" is Yamaha's weird term for the type of the command, such as sending a patch, requesting a patch, or sending a single parameter. The substatus here is set to 112 (0x70), otherwise known as binary `01110000`. This sets the top three bits to 1 and indicates a parameter change. The channel (0...15) is the lower three bits. Thus the byte is 112 plus the channel.

Yamaha chose to embed the current MIDI channel of the synthesizer in the sysex message, a common strategy for early synthesizers. The idea was that you might have multiple DX7 synthesizers on the same MIDI daisy-chain and putting the channel here made it clear to which synthesizer you were sending commands. Most later manufacturers have instead opted for an ID of some sort, to be set on the synthesizer itself.

Next come the MSB and LSB of the parameter number. Yamaha has 155 parameters, as shown in Table 2. The parameter number is equal to $MSB \times 128 + LSB$.

Finally comes the parameter value. Different parameters have different value ranges, though one oddity of the DX7 is that its larger range parameters don't go from 0...127, but rather are restricted to 0...99.

The DX7 does not use CC to change parameters; CC is reserved for a small number of tasks such as indicating that the sustain pedal has been depressed. The reason is simple: CC can handle at most 128 parameters (and realistically at most 119 or so), but the DX7 has 155 parameters. When the DX7 came out, NRPN had not yet been invented. Thus the DX7 resorts to sysex for all of its parameter changes. This is very common among synthesizers of the period, especially Japanese ones.

Changing Patches This is straightforward on the DX7: you just send a Program Change (PC) message with a value from 0...31 for the internal bank, and 32...63 for the first bank in the cartridge.

Requesting the Current Working Patch The sysex message is:

`0xF0 0x43` *substatus(0x20)+channel* `0x00 0xF7`

Here the substatus is 32 (0x20), and the following byte is `0x00`, indicating a patch request.

Sending/Receiving the Current Working Patch The DX7 will send you the current patch with this message, and you can also update the current patch by sending it the same:

`0xF0 0x43` *substatus(0x0)+channel* `0x00 0x01 0x1B` *data... checksum* `0xF7`

Here the substatus is 0, indicating a patch dump. Next comes `0x00`, indicating a single patch is being sent (and to current memory). Next comes the byte count in MSB and LSB: all DX7 patches are 155 bytes long, so the MSB is 1 and the LSB is 27 (`0x1B`).

Following this are all 155 parameters from Table 2, in order, one byte each. This is very simple and elegant: many other early synthesizers try to pack the parameters into fewer bytes, which makes things very difficult.

Finally we come to the *checksum*. Early MIDI transmission was unreliable, and so many synthesizers computed a checksum at the end of a long sysex string to verify that it had been sent correctly. In the case of the Yamaha DX7, the checksum is computed as follows. First, add up all the bytes in the data (as unsigned integers). Then Mod this with 128 (this just gets the lower 7 bits). Finally subtract the result from 128.

After updating the patch, you must wait for 50ms to let the synthesizer complete its task, or you will overwhelm the MIDI buffer.

Requesting a Patch from RAM This cannot be done on the DX7 at all. What you can do instead is change the patch to the given patch via a PC, thus loading it into current working memory, and then request the current working patch.

Writing a Patch to RAM This cannot be done on the DX7 at all. You can only write an entire bank of patches at a time (see below).

Requesting a Bank Patch This cannot be done via MIDI on the DX7 but it can be done on later incarnations, such as the Yamaha TX216/816. The message is

`0xF0 0x43` *substatus(0x20)+channel* `0x09 0xF7`

Here the substatus is 32 (0x20), and the following byte is `0x09`, indicating a bank request.

If you're on a DX7 where this isn't possible, you can still manually send the patch editor a patch from the DX's front panel.

Writing/Receiving a Bank Patch The DX7 will send you the current patch with this message, and you can also update the current patch by sending it the same:

`0xF0 0x43` *substatus(0x0)+channel* `0x09 0x20 0x00` *data... checksum* `0xF7`

Here, the substatus is 0, and the following byte is `0x09`, indicating a bank dump. The bank data is 4096 bytes in length (hence the MSB and LSB are `0x20` and `0x00`). Unlike a patch dump, the bank dump is tightly packed into bytes to save space, which is frustrating. Google for "DX7 VMEM" to read the format. The checksum is computed on the data in the same way as before.

Most modern synthesizers have abandoned the strategy of custom packed messages for an entire bank of patches, if only because banks have grown in size. Instead modern synthesizers would respond to some kind of bank request by sending each patch separately. This means that if you're going to build a librarian, you'll need to be able to handle *both* bank-dump messages *and* streams of individual-patch messages, depending on the synthesizer in question.

Other Features Later versions of the DX7 (such as the DX7-II) added additional features. However the original DX7 format didn't provide space for these features, so Yamaha included them as separate messages. This meant that if you wanted the full set of parameters for a given patch, you'd need to do send multiple messages. In later Yamaha FM synthesizers, such its 4-Operator series,Yamaha followed this strategy extensively, sending up to four messages for a single patch.

This is very frustrating to deal with in a patch editor designed to handle all synthesizers in a family, because *some* synthesizers would only send a single message, others would send two messages, others three, and others four: and some machines would send one set of four messages, while others sent a *different set*. Thankfully Yamaha would always send the "classic" message *last* so we would know when a collection of messages had ended for a given patch.

7 Case Study: the Dave Smith Instruments Prophet '08

Dave Smith founded a company called Sequential Circuits in 1974, and released the first synthesizer which used a CPU and RAM, the Prophet 5, in 1978. He then went on to invent MIDI in order to better take advantage of CPUs on synthesizers. Sequential Circuits failed in 1987 and the trademark was acquired by Yamaha, while Smith and others on his team went on to develop synthesizers for Korg. In 2002 Smith decided to start another company, this time called Dave Smith Instruments. In 2008 the company released the Prophet '08 synthesizer.

The Prophet '08 came in keyboard and desktop versions. Figure 2 (page 2) shows a desktop version.

In 2015, the heads of Roland and Yamaha jointly arranged for Yamaha to return the Sequential Circuits trademark to Smith as a gesture of goodwill. Dave Smith Instruments was then promptly renamed to Sequential. As of the writing of this document (January 2023), Dave Smith has recently passed away.

The Prophet '08 is a modern synthesizer, part of the analog renaissance. It is an analog synthesizer with eight voices and two banks of 128 patches each. The Prophet '08's interface with patch editors is elegant, clean, and straightforward, as is befitting a machine designed by the inventor of MIDI.

Updating Individual Parameters in Working Memory This is done very simply with the NRPN. Simply send an NRPN message with the given parameter number as shown in Table 3. Most Prophet '08 parameters have ranges within 0–128, but a few are larger. As a rule, the Prophet '08 uses the "fine" NRPN scheme: the parameter value is just $MSB \times 128 + LSB$.

Changing Patches This is done in the obvious way: by first setting the bank (via a Bank Change using CC 32), and then doing a Program Change (PC). There is a bug on the Prophet '08: you cannot change the patch immediately after writing a patch or the patch will be corrupted.

Requesting the Current Working Patch The sysex message is:

```
0xF0 0x01 0x23 0x06 0xF7
```

Being both founded by Dave Smith, Sequential and Dave Smith Instruments both use a manufacturer's ID of `0x01`. `0x23` represents the Prophet '08 synthesizer to distinguish it from commands for other DSI/Sequential synths. `0x06` is the command to request the current working patch. Unlike the DX7, the Prophet '08 sysex has no way to distinguish between multiple Prophet '08s: it just assumes there's only one.

Sending/Receiving the Current Working Patch This is pretty straightforward, with some caveats:

`0xF0 0x01 0x23 0x03` *data...* `0xF7`

As usual, `0x01` is Sequential / Dave Smith Instruments and `0x23` represents the Prophet '08. `0x03` indicates a dump of a patch to/from current working memory.

The *data...* represents 384 parameters (all but the last 16, which are empty anyway, see Figure 3). Some of these parameters have ranges 0...256 and so cannot fit one to a sysex "byte". They have to be encoded into 7-bit bytes somehow. To do this, the Prophet '08 treats each parameter as an 8-bit byte. It goes through the parameters seven at a time, packing them into eight 7-bit sysex bytes as follows. For each of the seven bytes in the 8-bit group (or fewer if that's all that's left), it forms seven 7-bit sysex bytes by removing the high bit from the original bytes. It then prepends an additional 7-bit sysex byte consisting of the high bits that that had been removed, forming eight sysex bytes in all. This is then added to the *data...*, and this is done a total of 55 times until all the parameters have been consumed. All told the *data...* holds 439 bytes.

Korg (another manufacturer) uses a similar scheme, as does Yamaha for some machines. Other manufacturers, such as Oberheim, use a different approach: *nybblization*. Here each 8-bit byte is broken into two 4-bit nybbles, and each of them is sent as its own sysex byte.

There is no checksum.

0	layer1dco1frequency	67	layer1mod1destination	134	layer1track1note15
1	layer1dco1finetune	68	layer1mod2source	135	layer1track1note16
2	layer1dco1shape	69	layer1mod2amount	136	layer1track2note1
3	layer1dco1glide	70	layer1mod2destination	137	layer1track2note2
4	layer1dco1key	71	layer1mod3source	138	layer1track2note3
5	layer1dco2frequency	72	layer1mod3amount	139	layer1track2note4
6	layer1dco2finetune	73	layer1mod3destination	140	layer1track2note5
7	layer1dco2shape	74	layer1mod4source	141	layer1track2note6
8	layer1dco2glide	75	layer1mod4amount	142	layer1track2note7
9	layer1dco2key	76	layer1mod4destination	143	layer1track2note8
10	layer1sync	77	layer1track1destination	144	layer1track2note9
11	layer1glidemode	78	layer1track2destination	145	layer1track2note10
12	layer1slop	79	layer1track3destination	146	layer1track2note11
13	layer1mix	80	layer1track4destination	147	layer1track2note12
14	layer1noise	81	layer1wheelamount	148	layer1track2note13
15	layer1vcffrequency	82	layer1wheeldestination	149	layer1track2note14
16	layer1vcfresonance	83	layer1pressureamount	150	layer1track2note15
17	layer1vcfkeyboardamount	84	layer1pressuredestination	151	layer1track2note16
18	layer1vcfaudiomodulation	85	layer1breathamount	152	layer1track3note1
19	layer1vcfpoles	86	layer1breathdestination	153	layer1track3note2
20	layer1env1amount	87	layer1velocityamount	154	layer1track3note3
21	layer1env1velocityamount	88	layer1velocitydestination	155	layer1track3note4
22	layer1env1delay	89	layer1footamount	156	layer1track3note5
23	layer1env1attack	90	layer1footdestination	157	layer1track3note6
24	layer1env1decay	91	layer1tempo	158	layer1track3note7
25	layer1env1sustain	92	layer1clockdivide	159	layer1track3note8
26	layer1env1release	93	layer1pitchbendrange	160	layer1track3note9
27	layer1vcainitiallevel	94	layer1sequencertrigger	161	layer1track3note10
28	layer1vcaoutputspread	95	layer1unisonmode	162	layer1track3note11
29	layer1vcavoicevolume	96	layer1unisonkeymode	163	layer1track3note12
30	layer1env2amount	97	layer1arpeggiatormode	164	layer1track3note13
31	layer1env2velocityamount	98	layer1env3repeat	165	layer1track3note14
32	layer1env2delay	99	layer1unison	166	layer1track3note15
33	layer1env2attack	100	layer1arpeggiator	167	layer1track3note16
34	layer1env2decay	101	layer1sequencer	168	layer1track4note1
35	layer1env2sustain	102	[empty]	169	layer1track4note2
36	layer1env2release	103	[empty]	170	layer1track4note3
37	layer1lfo1frequency	104	[empty]	171	layer1track4note4
38	layer1lfo1shape	105	layer1tetraassignableparameter1	172	layer1track4note5
39	layer1lfo1amount	106	layer1tetraassignableparameter2	173	layer1track4note6
40	layer1lfo1moddestination	107	layer1tetraassignableparameter3	174	layer1track4note7
41	layer1lfo1keysync	108	layer1tetraassignableparameter4	175	layer1track4note8
42	layer1lfo2frequency	109	[empty]	176	layer1track4note9
43	layer1lfo2shape	110	layer1tetrafeedbackgain	177	layer1track4note10
44	layer1lfo2amount	111	layer1tetrapushitnote	178	layer1track4note11
45	layer1lfo2moddestination	112	layer1tetrapushitvelocity	179	layer1track4note12
46	layer1lfo2keysync	113	layer1tetrapushitmode	180	layer1track4note13
47	layer1lfo3frequency	114	layer1tetrasuboscillator1level	181	layer1track4note14
48	layer1lfo3shape	115	layer1tetrasuboscillator2level	182	layer1track4note15
49	layer1lfo3amount	116	layer1tetrafeedbackvolume	183	layer1track4note16
50	layer1lfo3moddestination	117	layer1tetraeditorbyte	184	[empty]
51	layer1lfo3keysync	118	splitpoint	185	[empty]
52	layer1lfo4frequency	119	keyboardmode	186	[empty]
53	layer1lfo4shape	120	layer1track1note1	187	[empty]
54	layer1lfo4amount	121	layer1track1note2	188	[empty]
55	layer1lfo4moddestination	122	layer1track1note3	189	[empty]
56	layer1lfo4keysync	123	layer1track1note4	190	[empty]
57	layer1env3moddestination	124	layer1track1note5	191	[empty]
58	layer1env3amount	125	layer1track1note6	192	[empty]
59	layer1env3velocityamount	126	layer1track1note7	193	[empty]
60	layer1env3delay	127	layer1track1note8	194	[empty]
61	layer1env3attack	128	layer1track1note9	195	[empty]
62	layer1env3decay	129	layer1track1note10	196	[empty]
63	layer1env3sustain	130	layer1track1note11	197	[empty]
64	layer1env3release	131	layer1track1note12	198	[empty]
65	layer1mod1source	132	layer1track1note13	199	[empty]
66	layer1mod1amount	133	layer1track1note14		

Table 3: DSI Prophet '08 Parameters, Layer 1 only (using Edisyn names). Layer 2 parameters are identical and are numbers 200–399.

Requesting a Patch from RAM This is simply:

`0xF0 0x01 0x23 0x05` *bank number* `0xF7`

`0x05` indicates a request for a patch from RAM, with the provided *bank* and *number*. That's it.

Writing/Receiving a Patch to/from RAM Unlike the DX7, this can be easily done:

`0xF0 0x01 0x23 0x02` *bank number data...* `0xF7`

Here `0x02` indicates dumping a patch to RAM, with the given *bank* and *number*. The *data...* is encoded in the same way as in sending the current working patch above. Again, there is no checksum.

Requesting a Bank Unlike the DX7, the Prophet '08 does not have separate whole-bank sysex messages. Banks are requested simply by requesting each individual patch in turn.

Writing/Receiving a Bank Patch Unlike the DX7, the Prophet '08 does not have separate whole-bank sysex messages: to write a bank, you write each patch individually.

Other Features Like many synthesizers, the Prophet '08 can also update and dump global parameters in a manner similar to per-patch dumps.

Other DSI Synthesizers There are several other synthesizers which are derived from the Prophet '08 and which more or less share the same parameters and sysex. These are the Mopho (`0x25`), Tetra (`0x26`), Mopho Keyboard and Mopho SE (`0x27`), and Mopho X4 (`0x29`). Whereas the Prophet '08 has eight voices, the Tetra has only four, and the Mopho devices have only one. The Tetra is four-voice multitimbral, however: each of its four voices can be assigned a different patch. Multitimbral patches ("combos" in Tetra-speak) have their own separate sysex. We'll see an example of a multitimbral synthesizer in the next Section.

These synthesizers overlap so closely with one another and with the Prophet '08 that with care it is possible to create a unified patch editor for all of them: but they differ enough to require different-length sysex messages, different ranges for some parameters, and (for some unknown reason) a different ordering of NRPN messages for individual parameters. Particularly problematically, the Mopho and Tetra devices have a few parameters whose values are *not contiguous* — there's a hole in the middle of them which the user cannot set.[9] This can pose a challenge to an editor's GUI widgets.

[9]And in fact if you set them to values in this hole, it can crash the synthesizer.

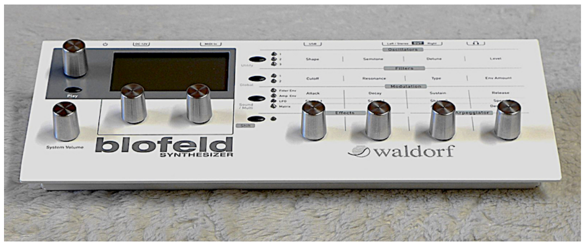


Figure 7: Waldorf Blofeld.

Image by "deepsonic", CC BY-SA 2.0, https://flickr.com/photos/73143485@N02/50013762981

Sequential went on to make a successor to the Prophet '08 called the Prophet Rev2 with many more features. Indeed the two are similar enough in internal structure that it is possible to convert Prophet '08 patches to the Prophet Rev2 and (to the degree possible) the other way. Unfortunately, while the Prophet '08 has meticulously described sysex and MIDI control, the Prophet Rev2's sysex is *completely undocumented* and has certain unexpected bugs. The sysex spec had to be entirely reverse engineered from scratch.[10]

8 Case Study: the Waldorf Blofeld

Waldorf is a German synthesizer company founded in 1988, and is strongly associated with so-called *wavetable synthesis*. The company was restructuring from bankruptcy in 2007 when it produced its most successful synthesizer, the Blofeld.[11] The Blofeld is a combination wavetable and virtual analog synthesizer, and is famous for having many capabilities packed into a small package. The Blofeld came in keyboard and desktop versions: the module version is shown in Figure 7.

The Blofeld is multitimbral, and has eight banks of 128 single-mode patches, plus one bank of 128 multimode patches. It can also play a large number of samples loaded by the user as part of single patches. Like many multitimbral synthesizers, the Blofeld has two modes: single and multi. In single mode the Blofeld plays a single patch and so has a single current working memory area. In multimode, the Blofeld plays a multimode patch referring to up to 16 single patches. Each of the single patches has its own current working memory area and can have its parameters updated independently.

Because the Blofeld has both single-mode and multimode patches, this means that there are two different sets of sysex messages for everything, which requires two separate editors in Edisyn, one for each patch type.

Though early Waldorf was famous for its meticulously documented and carefully debugged sysex, the Blofeld's sysex is a mixed bag. Its single-mode sysex is well

[10]For the reverse engineered Rev2 sysex spec, see the very end of https://github.com/eclab/edisyn/blob/master/edisyn/synth/sequentialprophetrev2/SequentialProphetRev2.java

[11]Yes, it's named after the Bond villian.

amplifiermodamount	arp08glide	arp16glide	effect2parameter2	envelope4mode	lfo1shape	modulation11destination	modulation8source	osc2balance
amplifiermodsource	arp08length	arp16length	effect2parameter3	envelope4release	lfo1speed	modulation11source	modulation9amount	osc2bendrange
amplifiervelocity	arp08step	arp16step	effect2parameter4	envelope4sustain	lfo1startphase	modulation12amount	modulation9destination	osc2brilliance
amplifiervolume	arp08timing	arp16timing	effect2parameter5	envelope4sustain2	lfo1sync	modulation12destination	modulation9source	osc2detune
arp01accent	arp09accent	arpeggiatorclock	effect2parameter6	envelope4trigger	lfo2clocked	modulation12source	name0	osc2fmamount
arp01glide	arp09glide	arpeggiatordirection	effect2parameter7	filter1cutoff	lfo2delay	modulation13amount	name1	osc2fmsource
arp01length	arp09length	arpeggiatorlength	effect2parameter8	filter1drive	lfo2fade	modulation13destination	name2	osc2keytrack
arp01step	arp09step	arpeggiatormode	effect2parameter9	filter1drivecurve	lfo2keytrack	modulation13source	name3	osc2level
arp01timing	arp09timing	arpeggiatoroctave	effect2type	filter1envamount	lfo2shape	modulation14amount	name4	osc2limitwt
arp02accent	arp10accent	arpeggiatorpattern	envelope1attack	filter1envvelocity	lfo2speed	modulation14destination	name5	osc2octave
arp02glide	arp10glide	arpeggiatorpatternlength	envelope1attacklevel	filter1fmamount	lfo2startphase	modulation14source	name6	osc2pulsewidth
arp02length	arp10length	arpeggiatorpatternreset	envelope1decay	filter1fmsource	lfo2sync	modulation15amount	name7	osc2pwmamount
arp02step	arp10step	arpeggiatorsortorder	envelope1decay2	filter1keytrack	lfo3clocked	modulation15destination	name8	osc2pwmsource
arp02timing	arp10timing	arpeggiatortempo	envelope1mode	filter1modamount	lfo3delay	modulation15source	name9	osc2samplebank
arp03accent	arp11accent	arpeggiatortimingfactor	envelope1release	filter1modsource	lfo3fade	modulation16amount	name10	osc2semitone
arp03glide	arp11glide	arpeggiatorvelocitymode	envelope1sustain	filter1pan	lfo3keytrack	modulation16destination	name11	osc2shape
arp03length	arp11length	category	envelope1sustain2	filter1panamount	lfo3shape	modulation16source	name12	osc2synctoosc3
arp03step	arp11step	effect1mix	envelope1trigger	filter1pansource	lfo3speed	modulation1amount	name13	osc3balance
arp03timing	arp11timing	effect1parameter0	envelope2attack	filter1resonance	lfo3startphase	modulation1destination	name14	osc3bendrange
arp04accent	arp12accent	effect1parameter1	envelope2attacklevel	filter1type	lfo3sync	modulation1source	name15	osc3brilliance
arp04glide	arp12glide	effect1parameter10	envelope2decay	filter2cutoff	modifier1constant	modulation2amount	name16	osc3detune
arp04length	arp12length	effect1parameter11	envelope2decay2	filter2drive	modifier1operation	modulation2destination	noisebalance	osc3fmamount
arp04step	arp12step	effect1parameter12	envelope2mode	filter2drivecurve	modifier1sourcea	modulation2source	noisecolour	osc3fmsource
arp04timing	arp12timing	effect1parameter13	envelope2release	filter2envamount	modifier1sourceb	modulation3amount	noiselevel	osc3keytrack
arp05accent	arp13accent	effect1parameter2	envelope2sustain	filter2envvelocity	modifier2constant	modulation3destination	osc1balance	osc3level
arp05glide	arp13glide	effect1parameter3	envelope2sustain2	filter2fmamount	modifier2operation	modulation3source	osc1bendrange	osc3octave
arp05length	arp13length	effect1parameter4	envelope2trigger	filter2fmsource	modifier2sourcea	modulation4amount	osc1brilliance	osc3pulsewidth
arp05step	arp13step	effect1parameter5	envelope3attack	filter2keytrack	modifier2sourceb	modulation4destination	osc1detune	osc3pwmamount
arp05timing	arp13timing	effect1parameter6	envelope3attacklevel	filter2modamount	modifier3constant	modulation4source	osc1fmamount	osc3pwmsource
arp06accent	arp14accent	effect1parameter7	envelope3decay	filter2modsource	modifier3operation	modulation5amount	osc1fmsource	osc3semitone
arp06glide	arp14glide	effect1parameter8	envelope3decay2	filter2pan	modifier3sourcea	modulation5destination	osc1keytrack	osc3shape
arp06length	arp14length	effect1parameter9	envelope3mode	filter2panamount	modifier3sourceb	modulation5source	osc1level	oscallocation
arp06step	arp14step	effect1type	envelope3release	filter2pansource	modifier4constant	modulation6amount	osc1limitwt	oscglide
arp06timing	arp14timing	effect2mix	envelope3sustain	filter2resonance	modifier4operation	modulation6destination	osc1octave	oscglidemode
arp07accent	arp15accent	effect2parameter0	envelope3sustain2	filter2type	modifier4sourcea	modulation6source	osc1pulsewidth	oscgliderate
arp07glide	arp15glide	effect2parameter1	envelope3trigger	filterrouting	modifier4sourceb	modulation7amount	osc1pwmamount	oscpitchamount
arp07length	arp15length	effect2parameter10	envelope4attack	lfo1clocked	modulation10amount	modulation7destination	osc1pwmsource	oscpitchsource
arp07step	arp15step	effect2parameter11	envelope4attacklevel	lfo1delay	modulation10destination	modulation7source	osc1samplebank	ringmodbalance
arp07timing	arp15timing	effect2parameter12	envelope4decay	lfo1fade	modulation10source	modulation8amount	osc1semitone	ringmodlevel
arp08accent	arp16accent	effect2parameter13	envelope4decay2	lfo1keytrack	modulation11amount	modulation8destination	osc1shape	unisono
								unisonodetune

Table 4: Waldorf Blofeld single mode patch parameters in alphabetical order (using Edisyn names).

documented by the company, but that's it. The multimode[12] and wavetable uploading[13] sysex is undocumented, though both have been reverse engineered and are now reasonably well understood. Its sample uploading sysex is still undocumented and its format is unknown. I hope to convince the company to release that information.

8.1 Single-Mode Patches

When in single mode, the Blofeld plays only a single patch at a time, and thus only has a single current working memory buffer.

Updating Individual Parameters in Working Memory Some parameters can be updated via CC. All parameters can be updated via sysex. Whether parameters are updated via CC, sysex, or both, is specified by the user on the front panel. Updates via sysex look like this:

`0xF0 0x3E 0x13` *synth-id* `0x20 0x00` *MSB LSB data* `0xF7`

`0x3E` of course indicates Waldorf. `0x13` declares that this message is for the Waldorf Blofeld. The *synth-id* is a value 0...127 which you manually set on your Blofeld and which distinguishes it from other Blofelds you might own.[14] If you set the *synth-id* to 127, all Blofelds listening will respond to your message. The *synth-id* strategy has been adopted by a number of modern synthesizers, displacing the earlier embedded-channel approach that the DX7 took.

`0x20` indicates that this is a parameter update message, and `0x0` indicates that the parameter to be updated is in current working memory (there are other options in multimode). The parameter being updated is one of 307 parameters spread among 387 values (the others being unused). It's not worthwhile showing them here. However note that some parameters are actually multiple parameters encoded together, such as "oscillator allocation" and "unison"; or "arp01step", "arp01glide", and "arp01accent". This means that when you update a parameter of this kind, in your patch editor, you must combine it with several other parameters and send a single message with all of them glommed together. That is not pretty.

Changing Patches This is done by first setting the bank (via a Bank Change using CC 32 or 0, it doesn't matter which), and then doing a Program Change. You must pause for 200ms after a patch change or the Blofeld will get confused.

[12]For the reverse engineered Blofeld Multimode sysex spec, see https://github.com/eclab/edisyn/blob/master/edisyn/synth/waldorfblofeld/WaldorfBlofeldMulti.java

[13]For the reverse engineered Blofeld Wavetable sysex spec, see https://github.com/eclab/edisyn/blob/master/edisyn/synth/waldorfblofeld/WaldorfBlofeldWavetable.java

[14]In the manual for the Waldorf Blofeld synthesizer, it says that if you have purchased 127 Blofelds and so have run out of IDs to distinguish them, to contact Waldorf and the head of the company will invite you for a hamburger dinner.

arptempo	inst2lowkey	inst3sustain	inst5hikey	inst6pressure	inst8channel	inst9modwheel	inst10volume	inst12lowkey	inst13sustain	inst15hikey	inst16pressure
inst1bank	inst2lowvel	inst3transpose	inst5hivel	inst6progchange	inst8detune	inst9number	inst11bank	inst12lowvel	inst13transpose	inst15hivel	inst16progchange
inst1bend	inst2midi	inst3usb	inst5local	inst6status	inst8edits	inst9panning	inst11bend	inst12midi	inst13usb	inst15local	inst16status
inst1channel	inst2modwheel	inst3volume	inst5lowkey	inst6sustain	inst8hikey	inst9pressure	inst11channel	inst12modwheel	inst13volume	inst15lowkey	inst16sustain
inst1detune	inst2number	inst4bank	inst5lowvel	inst6transpose	inst8hivel	inst9progchange	inst11detune	inst12number	inst14bank	inst15lowvel	inst16transpose
inst1edits	inst2panning	inst4bend	inst5midi	inst6usb	inst8local	inst9status	inst11edits	inst12panning	inst14bend	inst15midi	inst16usb
inst1hikey	inst2pressure	inst4channel	inst5modwheel	inst6volume	inst8lowkey	inst9sustain	inst11hikey	inst12pressure	inst14channel	inst15modwheel	inst16volume
inst1hivel	inst2progchange	inst4detune	inst5number	inst7bank	inst8lowvel	inst9transpose	inst11hivel	inst12progchange	inst14detune	inst15number	name0
inst1local	inst2status	inst4edits	inst5panning	inst7bend	inst8midi	inst9usb	inst11local	inst12status	inst14edits	inst15panning	name1
inst1lowkey	inst2sustain	inst4hikey	inst5pressure	inst7channel	inst8modwheel	inst9volume	inst11lowkey	inst12sustain	inst14hikey	inst15pressure	name2
inst1lowvel	inst2transpose	inst4hivel	inst5progchange	inst7detune	inst8number	inst10bank	inst11lowvel	inst12transpose	inst14hivel	inst15progchange	name3
inst1midi	inst2usb	inst4local	inst5status	inst7edits	inst8panning	inst10bend	inst11midi	inst12usb	inst14local	inst15status	name4
inst1modwheel	inst2volume	inst4lowkey	inst5sustain	inst7hikey	inst8pressure	inst10channel	inst11modwheel	inst12volume	inst14lowkey	inst15sustain	name5
inst1number	inst3bank	inst4lowvel	inst5transpose	inst7hivel	inst8progchange	inst10detune	inst11number	inst13bank	inst14lowvel	inst15transpose	name6
inst1panning	inst3bend	inst4midi	inst5usb	inst7local	inst8status	inst10edits	inst11panning	inst13bend	inst14midi	inst15usb	name7
inst1pressure	inst3channel	inst4modwheel	inst5volume	inst7lowkey	inst8sustain	inst10hikey	inst11pressure	inst13channel	inst14modwheel	inst15volume	name8
inst1progchange	inst3detune	inst4number	inst6bank	inst7lowvel	inst8transpose	inst10hivel	inst11progchange	inst13detune	inst14number	inst16bank	name9
inst1status	inst3edits	inst4panning	inst6bend	inst7midi	inst8usb	inst10local	inst11status	inst13edits	inst14panning	inst16bend	name10
inst1sustain	inst3hikey	inst4pressure	inst6channel	inst7modwheel	inst8volume	inst10lowkey	inst11sustain	inst13hikey	inst14pressure	inst16channel	name11
inst1transpose	inst3hivel	inst4progchange	inst6detune	inst7number	inst9bank	inst10lowvel	inst11transpose	inst13hivel	inst14progchange	inst16detune	name12
inst1usb	inst3local	inst4status	inst6edits	inst7panning	inst9bend	inst10midi	inst11usb	inst13local	inst14status	inst16edits	name13
inst1volume	inst3lowkey	inst4sustain	inst6hikey	inst7pressure	inst9channel	inst10modwheel	inst11volume	inst13lowkey	inst14sustain	inst16hikey	name14
inst2bank	inst3lowvel	inst4transpose	inst6hivel	inst7progchange	inst9detune	inst10number	inst12bank	inst13lowvel	inst14transpose	inst16hivel	name15
inst2bend	inst3midi	inst4usb	inst6local	inst7status	inst9edits	inst10panning	inst12bend	inst13midi	inst14usb	inst16local	name16
inst2channel	inst3modwheel	inst4volume	inst6lowkey	inst7sustain	inst9hikey	inst10pressure	inst12channel	inst13modwheel	inst14volume	inst16lowkey	volume
inst2detune	inst3number	inst5bank	inst6lowvel	inst7transpose	inst9hivel	inst10progchange	inst12detune	inst13number	inst15bank	inst16lowvel	
inst2edits	inst3panning	inst5bend	inst6midi	inst7usb	inst9local	inst10status	inst12edits	inst13panning	inst15bend	inst16midi	
inst2hikey	inst3pressure	inst5channel	inst6modwheel	inst7volume	inst9lowkey	inst10sustain	inst12hikey	inst13pressure	inst15channel	inst16modwheel	
inst2hivel	inst3progchange	inst5detune	inst6number	inst8bank	inst9lowvel	inst10transpose	inst12hivel	inst13progchange	inst15detune	inst16number	
inst2local	inst3status	inst5edits	inst6panning	inst8bend	inst9midi	inst10usb	inst12local	inst13status	inst15edits	inst16panning	

Table 5: Waldorf Blofeld multimode patch parameters in alphabetical order (using Edisyn names).

Requesting the Current Working Patch The sysex message is:

`0xF0 0x3E 0x13` *synth-id* `0x00 0x7F 0x00 0x00 0xF7`

`0x3E 0x13` as usual indicate Waldorf and the Blofeld, and the *synth-id* is self-explanatory. `0x00` indicates a request for a patch, and `0x7F 0x00` indicates that it is from the current working memory in singe-mode. The final `0x00` is notionally a checksum but in fact is ignored.

Requesting a Patch from RAM The sysex message is:

`0xF0 0x3E 0x13` *synth-id* `0x00` *bank number* `0x00 0xF7`

This is just like requesting from current working memory, except that *bank* indicates the bank (A–H, thus 0–7) and *number* represents the patch number.

Sending/Receiving the Current Working Patch This is:

`0xF0 0x3E 0x13` *synth-id* `0x10 0x7F 0x00` *data...* *checksum* `0xF7`

As usual `0x3E 0x13 and` *synth-id* represent the Waldorf Blofeld with a given ID. `0x10` indicates a patch dump, and `0x7F 0x00` indicates that it is from the current working memory in singe-mode. The *data...* is 383 bytes representing all of the parameters, one sysex byte per parameter. However recall that some "parameters" are amalgams of up to three parameters when updating individual parameters. It's the same here: they are combined into a single byte, which complicates matters.

The *checksum* is somewhat similar to how the DX7 does it. First we sum all the *data...* bytes as unsigned integers. The checksum is simply this value mod 128.

Writing/Receiving a Patch to/from RAM This is almost the same as sending the current working patch:

`0xF0 0x3E 0x13` *synth-id* `0x10` *bank number data...* *checksum* `0xF7`

The only difference here is that we specify the bank and number of the patch in RAM. The Blofeld requires a pause of 75ms after a patch is written to it in order to process the message.

Requesting All Single-Mode Patches There is a way to request all the patches from the Blofeld:

`0xF0 0x3E 0x13` *synth-id* `0x00 0x40 0x00 0x00 0xF7`

This is just like requesting patches from RAM or the current working memory, except that `0x40 0x00` instead asks for *all the single-mode patches* on the machine. The Blofeld will respond by dumping every single patch, one by one, from RAM.

There is no way to request a single bank of patches nor to write a bank: to do that you must request or write patches one by one.

8.2 Multimode Patches

Multimode patches are not very complicated. They consist of just a patch name, an overall volume, an overall tempo (Blofeld single patches have features which can be synced to beats); and then, for each part, various parameters defining that part. This includes the single-mode patch to be used in the given part, whether various features are turned on (like pitch bend or the sustain pedal), the volume, pan, transpose, an detuning of the part, the MIDI channel used by the part, and the lowest and highest velocity and key (in order to split the keyboard among multiple parts). That's it.

When a multimode patch is loaded, the single-mode patches are also loaded into their respective parts. Thus in addition to the current working memory of the multimode patch, there are dedicated current working memory parts for each of the single-mode patches.

Updating Individual Parameters in Working Memory It is not possible to update multimode parameters. It *is* possible to update parameters of each of the single mode patches in the parts' current working memory areas, but this isn't particularly useful in practice. To do this, you'd say:

`0xF0 0x3E 0x13` *synth-id* `0x20` *part MSB LSB data* `0xF7`

These values have the same meaning as updating parameters in single-mode: except that *part* refers to the particular current working memory part (0–15).

Changing Patches There is just one multimode bank on the Blofeld: but you still need to "change" to that bank. To do this, do a Bank Change (using CC 32 or 0, it doesn't matter which) to bank 32 (that's the magic number). Then you can do a Program Change to the given patch number. This is a slow process: you must pause for a full 800ms after a patch change or the Blofeld will get confused.

Requesting the Current Working Patch The sysex message is:

`0xF0 0x3E 0x13` *synth-id* `0x01 0x7F 0x00 0xF7`

This is similar to requesting the current working patch in single mode, except `0x01` requests a patch in multimode. There is no additional `0x00` as there was in single mode.

Requesting a Patch from RAM The sysex message is:

`0xF0 0x3E 0x13` *synth-id* `0x01` *bank number* `0xF7`

This is similar to requesting the a patch from RAM in single mode, except `0x01` requests a patch in multimode. Again, there is no additional `0x00` as there was in single mode.

Sending/Receiving the Current Working Patch This is:

`0xF0 0x3E 0x13` *synth-id* `0x11 0x7F 0x00` *data... checksum* `0xF7`

This is just like sending/receiving the current working patch in single mode, except that `0x11` indicates a multimode patch.

Writing/Receiving a Patch to/from RAM This is:

`0xF0 0x3E 0x13` *synth-id* `0x11` *bank number data... checksum* `0xF7`

Again, this is just like writing/receiving a patch to/from RAM in single mode, except that `0x11` indicates a multimode patch. The Blofeld requires a pause of 75ms after a patch is written to it in order to process the message.

Requesting All Multimode Patches Just as is the case for single-mode, we can request all the multimode patches (effectively the whole bank) like this:

`0xF0 0x3E 0x13` *synth-id* `0x01 0x40 0x00 0xF7`

Note again the `0x01` indicating multimode, and the lack of an additional final `0x00`.

8.3 Other Features

The Blofeld has sysex for a number of other features. Notably it can also update and dump global parameters in a manner similar to per-patch dumps. But most importantly, sysex is used to load sound samples into the Blofeld as well as wavetables. The sound sample sysex format has not been deciphered, but the wavetable sysex has been reverse engineered, and it is just a series of chunks of sound data corresponding to each wave in the wavetable. See Footnote 13 (page 13).

9 The Sysex Clown Show

In developing a patch editor, you will be forced to write a lot of hooks and customization. A *lot*. This is because while MIDI is standardized (thankfully), sysex is not, and even CC and NRPN have lots of wiggle room. As a result synthesizer manufacturers interpret these protocols in their own broken, mistaken, buggy, user-hostile, and sometimes utterly insane ways.

You will need to be prepared. Let's go over some examples, shall we?

Sending to Current Memory is Job #1, Yet... The Kawai K5, Oberheim Matrix 6, Kawai K1, Audiothingies Micromonsta, and Casio CZ-230S have no command to send a patch to current working memory. Many E-Mu machines have no way to *request or send to* current working memory. This is like building a house and forgetting to include the front door. It is the single most important capability a synthesizer must offer a patch editor and these stupid machines can't even get that right. Instead, Edisyn is forced to repurpose one of the RAM patches as a "scratch patch"—it writes to the patch, then tells the synth to change to that patch, thus loading it into current memory (but making that patch number useless).[15] Note that this wastes one of just *four* available Casio CZ-230S patch slots. The only alternative would be to update each parameter separately, a potentially very slow process, if it's available on the machines at all (and this is the only option for the Micromonsta).

The Ashun Sound Machines (ASM) Hydrasynth can send to current memory but with a very problematic twist. The Hydrasynth doesn't store a patch in current memory: it stores *all patches* in current memory. There's no way to write a patch: you must upload a patch to current memory, then write *all patches* from current memory to Flash. But there is no command to upload the current working patch: you must specify *which patch number* in current memory you want to upload to. This is a problem for the following reason. Let's say you have modified patch A4. You abandon it and switch to patch B7 and start editing it. Then you decide to write patch B7 to Flash. This will *also* write the modified A4 patch as well! Remember that *all* patches are written. The only real workaround is, once again, to use a scratch patch: instead of editing A4, we copy A4 to scratch patch H127 and edit there instead. Did I mention that the Hydrasynth can't request the current patch?

Sysex Reuse The Novation SL series uses the same exact, very long, sysex dump message for all of the machines in its family: but they all use the message in completely different ways depending on the machine. There is no way to tell for which kind of machine a message was intended by examining the message itself.

NRPN Only The PreenFM, Futuresonus Parva, ASM Hydrasynth, and Audiothingies Micromonsta cannot upload or download patches to current working memory via a single sysex message: instead you must S L O W L Y send every single parameter individually via NRPN. To make matters worse, the Audiothingies Micromonsta has weird NRPN bugs. Some Micromonsta parameters expect the MSB to be sent first, and others expect the LSB to be sent first, and if you do it wrong, the synthesizer will register bad values.

Pauses Many early machines had small cheap buffers that couldn't hold very much, and so you had to send long sysex messages piecemeal with significant pauses in-between the fragments. It's not entirely clear if this is legal MIDI. But the Casio CZ series really pushed the boundary of what was permissible in MIDI. The CZ requires that you send a *piece* of a sysex message, then wait for a *piece* of a sysex message response, then send the remainder of your message, then wait for the remainder of the response. Edisyn is written in Java, and Java is supposed to handle the ability to break sysex messages into fragments, but in fact it was never implemented because it's so obscure. This means that on some platforms (like Windows) it's impossible to write a patch editor in Java for the CZ.

[15]In development of Edisyn's E-Mu Proteus 2000 editor, it appeared for some time that, like earlier E-Mu machines, the Proteus 2000s couldn't to send to current memory. This would be particularly bad because the Proteus 2000 memory is not battery-backed RAM but is Flash, and sending to current memory via a scratch patch would burn Flash, eventually ruining the machine. But sending to current memory is critical. The alternative, updating parameter-by-parameter, is very slow as the Proteus 2000 has 800 or so parameters. Fortunately, it turned out that there was a secret way to send to current memory, but E-Mu had forgotten to document it.

Speed Continuing the pause mess: the M-Audio Venom can require as much as a 1 second pause between patch updates. The ASM Hydrasynth may be even slower. The Yamaha FS1R has a massive buffer, but if you stuff it full of patch writes, it will sit for many, many minutes slowly processing them.

Insane Complexity The Korg Wavestation's sysex is so complex it requires six long, long documents to explain it. Likewise the Kawai K5000. For their Proteus 2000 series, E-Mu decided to develop a massive, extraordinarily overwrought, future-proof protocol (requiring a book), which was still filled with bugs: then the company was promptly shut down and never made any future machines.

Baroque Parameters To change individual parameters, the Korg Wavestation actually requires that their values be embedded in sysex as *text strings*. For example, to set a parameter to -42, its sysex will actually look like `0xF0 ... '-' '4' '2' ... 0xF7`

Request What? You cannot request a patch from the Red Sound Darkstar, Yamaha DX7, or many Yamaha 4-Op synthesizers. The only way to get a patch from them is to manually send the patch from the synthesizer itself.

Bank Only Some synthesizers, such as certain Yamaha 4-Op synthesizers, the Korg Poly-800 and EX-800, and the Korg Wavestation (Wavesequences only), cannot send or receive individual patches: instead you must send or receive the entire bank at once. Thus in order to update a patch, you have to request the entire bank, modify it, and then resend the entire bank.

Bit Packing Some devices have exactly one byte or exactly two bytes per parameter. That's great! It allows an editor to write a patch to the synth using just a for-loop and a list of parameters. But many devices, especially early ones, cram their parameters together in crazy ways to squeeze every last bit out of them and save memory. Indeed, some devices actually break parameters into pieces in order to stuff them into the crannies of different bytes. This results in extremely tedious, complex writing and reading.

MIDI Only The Yamaha FB-01, Yamaha TQ5, and M-Audio Venom cannot be fully programmed by the user directly. You *have to use a patch editor*. It doesn't help that the some of the Venom's sysex is undocumented. A few machines, like the DSI Evolver Rack and the Oberheim Matrix 1000, *cannot be edited at all* from the front panel and *must* be edited via a patch editor.

Required Global Editors Edisyn as a rule does not have editors for global synthesizer-wide parameters (as opposed to patches), as they are normally set-and-forget affairs. But there are two exceptions. The Oberheim Matrix 1000 and M-Audio Venom both require global parameter editors because, in the absence of a editor, *there is no other way to set these parameters*. They cannot be set from the front panel of the machine itself! And they're not unimportant parameters either.

Pretend Hierarchies The Roland D-110 is a standard multitimbral synthesizer, with single-mode patches and multimode patches. But it pretends to the user to *only have multimode patches* with the single-mode patches "embedded" inside them. You cannot play a single patch: you have to play 8 patches at a time, and so to hear just one patch, you must turn the other 7 of them off. This is extraordinarily inconvenient, but it also means that how it presents itself to a patch editor over sysex and how it presents itself to the user is entirely different, making users extremely confused when they use a patch editor. Lest you think this was a fluke, a number of Yamaha machines caught this disease as well, up to and including the Yamaha FS1R.

Inability to Save State The 1990s saw the introduction of a variety of so-called *General MIDI* machines. Many of these were miniature synthesizers with no keyboard and few controls, meant to play fixed preset sounds, and were often designed to attach to a computer sound card to play music when you were playing a video game. In many cases you could program the patches on a General MIDI machine to some degree (so your video game can customize its music) but critically you often could not *save the patches* on the machine. If you power cycled the synthesizer, it was reset to its default state. These machines are thus mostly useless for modern purposes.

Custom Manual Modes In order to upload or download patches to the Audiothingies Microsmonsta, you have to *manually reboot the machine* into a special upload/download mode. Similarly, the Novation Drumstation and D station must be switched into "receive Sysex" mode (with an actual switch on the front panel) in order to accept patches.

Arbitrary Data Transfer A few machines (PreenFM2, Roli Seaboard series) use sysex not as a standardized protocol but as a transport mechanism for the raw bits of their internal data structure, including floats and who knows what else for their particular CPU and memory model. Even when the spec is open, it's exceptionally hard to encode and decode properly.

Sysex Messages Aren't Patches Some machines, such as the Waldorf Kyra, Emu Proteus 2000, M Audio Venom, Roland JV880, Roland U110, Yamaha 4-Operator Series, and ASM Hydrasynth, unnecessarily break a patch dump into some *N* messages, thus greatly complicating the patch loading process. Even more fun: the Roland U-220 stuffs *multiple patches* into a *smaller number of multiple sysex messages*: and it's effectively impossible to determine if a series of messages represents a single patch, the start of a bank of patches, or something in the middle of a stream.

Odd Undocumented Byte Encodings As mentioned in Section 7, one common 8-to-7-bit encoding is to send seven bytes with the high bits stripped, plus a byte consisting of all the high bits. Another is to break the 8-bit byte into 4-bit nybbles, sent separately. These are typically well documented and even if not, they're very obvious and easy to work out. But the ASM Hydrasynth uses Base64, an encoding mechanism meant to embed data in *text email messages*. In Base64, you concatenate three 8-bit bytes into a 24-bit string, then break it into four 6-bit chunks. Base64 assigns each possible chunk value to a unique ASCII character from the 64 characters `A...Z a...z 0...9 + /` with `=` used to pad at the end. Since ASCII only uses the first 7 bits, you can then just write out the Base64 characters as four characters in your sysex message. On top of it, ASM embedded a 4-byte CRC32 checksum, reversed and not in proper network order, and with each of the checksum bytes then subtracted from 255. And refused to provide documentation on it. It took me a month to crack it.

Manufacturer Namespace Violations We're not done with ASM's shenanigans yet. The Hydrasynth has a second way of dumping patches and banks over MIDI, meant to be fed directly to another Hydrasynth. This wraps the data in the format `0xF0 0x01 0x03 0x05 0x07` *data...* `0xF7` (individual patches) or `0xF0 0x02 0x04 0x06 0x08` *data...* `0xF7` (banks). See the problem? `0x01` is the namespace for Sequential, and `0x02` is the namespace for Big Briar. That is, the Hydrasynth generates illegal data masquerading as other manufacturer's sysex packets. You'll have to special-case for this.

10 The End of Sysex

If I may rant a bit more.

In the late 1980s, synthesizer manufacturers were generally good about publishing specifications for their sysex protocols. MIDI was open, and that was the spirit. And any way, if people made tools to interoperate with your synthesizer, this could only help boost its sales. Specifications appeared in the back of manuals, or as separate booklets available to developers, or as part of a machine's service manual. In the 1990s with the proliferation of rackmounts, this tradition only strengthened as these machines were difficult (or impossible) to program directly and relied heavily on patch editors.

Manufacturers following this tradition included every single one of the big names: Yamaha, Sequential, Oberheim, Korg, Waldorf, Roland, Casio, Kawai, E-Mu, Ensoniq, and so on. Even later manufacturers like M-Audio or Red Sound made their specs available. This wasn't just synthesizer manufacturers: effects units, MIDI routers, and so on also made their specs available.

Many manufacturers put a lot of effort into these specifications: they were large, detailed, and only *somewhat* buggy. It was a point of pride. Casio even put its spec in a cute booklet on MIDI control of the device, including silly cartoon characters to help explain what MIDI and sysex were [4].

In the 2000s we began to see synthesizer manufacturers neglecting sysex and their own third-party and user communities, necessitating reverse engineering. Some manufacturers began to treat sysex as a trade secret, perhaps to enable the sale of their own proprietary tools; and sometimes for no good reason at all. Arturia was an early adopter of this awful behavior, and ASM does it too. Other manufacturers didn't bother to release their sysex documentation, or lost it, requiring complete reverse engineering: I've had experiences along these lines from Novation, Audiothingies, M-Audio, Sequential, and Waldorf.[16] But the worst has been very recent machines with no sysex at all. Instead patches are uploaded via proprietary USB protocols. Korg has been doing this for quite some time, starting with the Microsampler,[17] and recently the WaveState. This is nothing less than planned obsolescence, as it makes the synthesizers dependent on proprietary software which eventually stops working.

Why would manufacturers cut out the very patch editor and tool developers who were helping their communities grow? People imagine a variety of invalid reasons, such as worries that buggy sysex would bomb the machine (as if not documenting would fix this), or being embarrassed by the low quality of one's sysex protocol, or somehow thinking that opening sysex in turn opens the manufacturer to reverse engineering their firmware (they have nothing to do with one another).

In fact I have seen only three reasons.

- **Selling Software** Manufacturers starting selling patch editor and librarian software as an add-on to their devices, and didn't want the competition. 1K Multimedia has gone so far as to require you to pay $20 to transfer your license to their (notionally free) only patch editor when you sell your synthesizer used.

- **Free Software** In other cases, manufacturers sold their synthesizer with a free software package, and figured there was no need for third-party tools. Unfortunately in practically every case this software became obsoleted with operating system upgrades (particularly on the Mac) and the manufacturers had no incentive to revise their software or to make it open source.

- **Laziness** The 2000s saw a dramatic decrease in the technical sophistication of the typical synthesizer customer. As a result, manufacturers likely felt that the customers who would use a patch editor, librarian, or external tool was dwindling, and so didn't bother to clean up and release a polished spec at all.

These reasons are all pretty bad, and the downside (killing their developer community) is not pretty.

There are some bright spots. Sequential still release specs, with some exceptions (such as the Rev2). They do this despite it not being in their interest to do so: they contract with a company to sell patch editor software. Waldorf also releases specs, though a few of their newer machines (Quantum, Iridium) unfortunately don't support sysex at all. Kawai still proudly publishes the specs for their machines even though they've been out of the synthesizer business for 20 years.

I think that the era of patch editors for *new* synthesizers is probably coming to a close. Many newer machines have poor sysex, or none, or secret sysex, and so a patch editor will largely be relegated to supporting older machines. Still, if you've got an older synthesizer you'd like to support for the community, a patch editor is a great thing to write. It's can be an exercise in reverse engineering (hopefully not), bit twiddling, working out bugs hidden in 30 year old hardware. But it's fun! That's why I started Edisyn: I wanted to learn how patch editors, and my synthesizers, worked. For fun.

If you've read this far, maybe you're serious about building an editor. If so, send me email (`sean@cs.gmu.edu`) and I'd be glad to tell you anything you want to know, including how to build off of Edisyn and so take advantage of its capabilities.

[16]Novation apparently lost a lot of internal documentation in the early 2000s due to, I was told, a hard drive failure. They're been kindly forthcoming with what they have, but it hasn't helped much.

[17]Indeed, the Microsampler's proprietary USB protocol appears to embed a custom version of sysex *in the USB protocol* to transfer samples to the machine. But you can't use the same sysex to transfer samples over standard MIDI.

References

[1] Sean Luke. Stochastic synthesizer patch exploration in Edisyn. In *International Conference on Computational Intelligence in Music, Sound, Art and Design (EvoMUSART)*, 2019.

[2] Dave Smith and Chet Wood. The 'USI', or Universal Synthesizer Interface. *Journal of the Audio Engineering Society*, October 1981.

[3] Sean Luke. *Computational Music Synthesis*. First edition, 2021. Available for free at http://cs.gmu.edu/∼sean/book/synthesis/.

[4] Casio Computer Co. Ltd. *Guidebook for MIDI*, 1985.