Chromasound

Programmable Sound Generation with FM Synthesis

Nir Jacobson

NIRJACOBSON.COM

In this book I describe a device that plays music using programmable sound generators. My goal is to provide you with tools and patterns that will help you build a device of your own.

Table of Contents

Chromasound went by the name FM-PSG during development (Programmable Sound Generation with FM Synthesis). Can you find each letter in the graphic above?

A Bit of History

Most of the audio devices we use today work using something called a **digital-to-analog converter** or simply **DAC**. Roughly stated, a DAC can convert a number stored on digital media into a voltage signal.

An **analog-to-digital converter** (ADC) is used to record as numbers the amplitude of sound over time measured in volts. A DAC is used to play back such a digital recording. It is essential in reproducing sound at a high quality that has been stored on digital media.

In the computer entertainment industry, the use of a DAC as a primary sound source didn't become economical until the early 1990's. This was due to high-quality audio needing at least 16 bits per sample. This was both expensive to store and expensive to process. Although DACs were available, there was a period during which home entertainment consoles and even some personal computers relied on **FM synthesis** as the primary source of sound.

FM synthesis works by generating sine waves, modulating the amplitude of the sine waves, and feeding the output into the modulation of sine wave frequency. As it turns out, many sounds can be reproduced this way. Setting the modulation parameters defines a voice. Once a voice is defined, the FM synth can be instructed to play notes through it.

In 1988, SEGA released the SEGA Genesis (known as the SEGA Mega Drive outside of North America), which was a home entertainment system that contained a Yamaha YM2612 FM synthesizer paired with a Texas Instruments SN76489 tone generator. The latter is a little simpler than an FM synth in the sense that there are no modulation parameters; each tone can only have its individual amplitude and frequency configured.

In 2010 I was introduced to something called the **demo scene** which is where computing enthusiasts share demos of their hardware and software. I noticed that many demos had a common trait: the style of the music. Chiptune is a style of music made using programmable sound generators.

I started looking for programmable sound generators that I could obtain, and for which there already existed music. I found the YM2612/SN76489, and the soundtracks of a catalog of SEGA Genesis games hosted on the Internet. I decided to build my own small demo that simply played these game soundtracks. After 14 years of iteration, I arrived at the implementation presented in this book.

Introduction to Circuits

DC Circuits

In this book we will be looking at a DC (direct current) circuit. In a DC circuit, electricity flows in one direction, starting and ending at some source of power. It passes through one or more devices that consume the electricity as it returns to the power source.
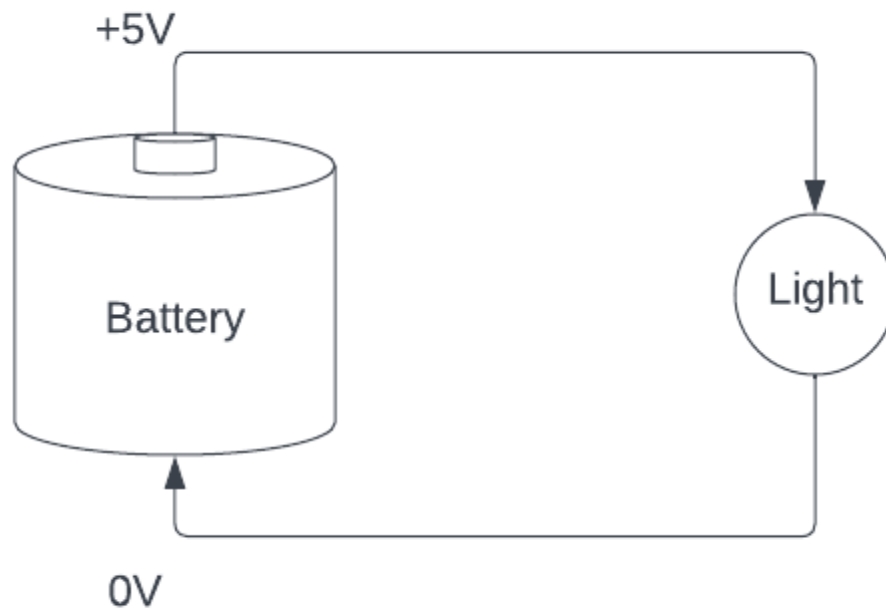
Let's look at the **schematic** of this circuit.
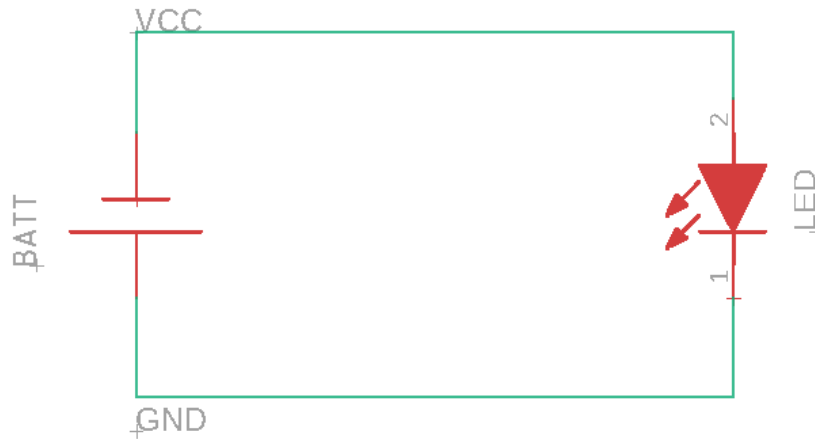
- The supply of electricity in the circuit is labeled VCC (Voltage Common Collector).
- The return path of electricity in the circuit is labeled GND (Ground).
- LED is short for "light-emitting diode".
- BATT is short for "battery".

There are two properties of the electricity that we are mainly concerned with: the **voltage** and the **current**. Voltage is measured in volts (V) and tells us how much energy one "unit of electricity" has. Current is measured in amperes (A) and tells us how many "units of electricity" there are, or how much electricity is flowing.

Let's add a power switch to the circuit so it isn't on all the time.
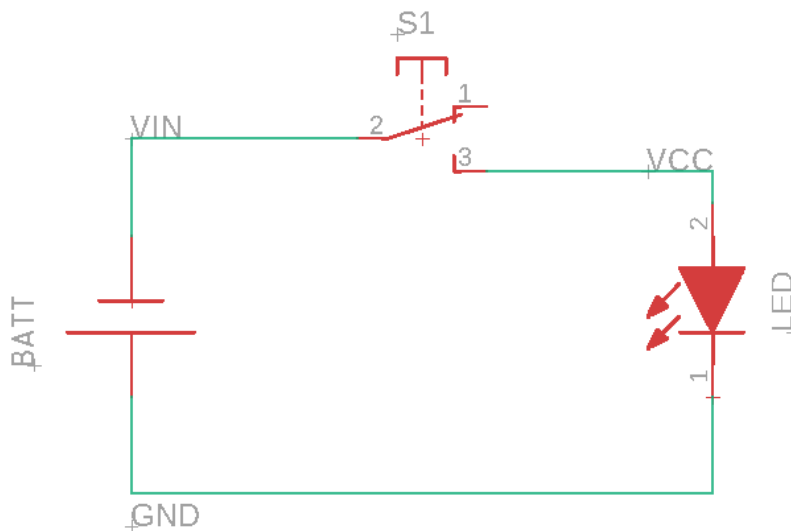
Resistors

Often the power supplied by a power source is too great to be used directly by a component such that it would damage it. A **resistor** resists the flow of electricity, lowering its voltage as it passes through the resistor. A resistor can be used to lower the amount of electricity delivered to a device.

Let's say our battery supplies 5 V. A typical LED has a forward voltage of 2.1 V and forward current of 20 mA. This is the voltage and current that are typically supplied to the LED. Greater values could damage it. A resistor can be used to control the amount of electricity passed from the battery to the LED.
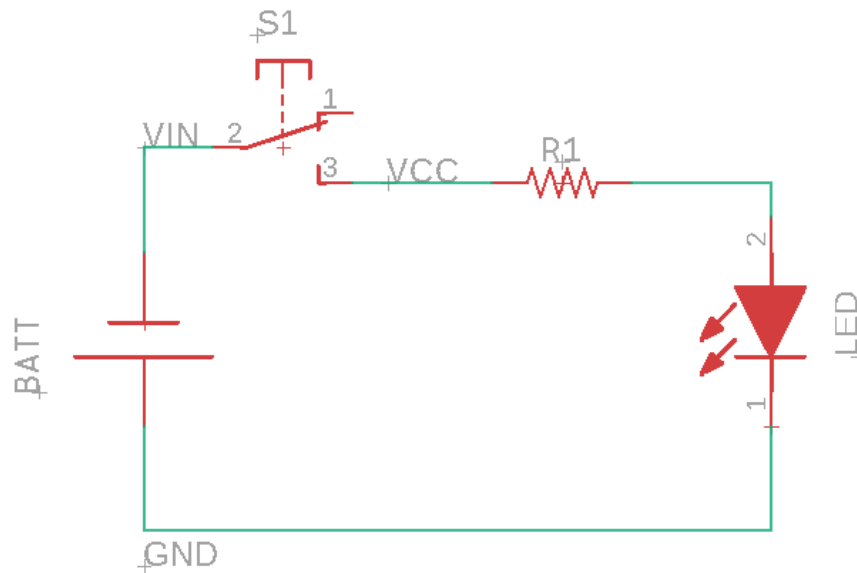
We use Ohm's Law to determine the value of the resistor we need to use. The value is measured in Ohms (Ω).

$$V = IR$$

- V is the voltage drop across the resistor.

- I is the current passing through the resistor, which is the same as the current passing through the LED.

- R is the value of the resistor.

$$5 \text{ V} - 2.1 \text{ V} = 20 \text{ mA} \left(\frac{1 \text{ A}}{1000 \text{ mA}}\right) R$$

$$2.9 \text{ V} = (0.020 \text{ A})R$$

$$R = \frac{2.9 \text{ V}}{0.020 \text{ A}}$$

$$R = 145 \text{ Ω}$$

Capacitors

Sometimes, the current consumed by a part of a circuit can spike in a way that the power source cannot support. A **capacitor** is placed next to that part of the circuit and stores a small amount of charge, similarly to a battery, that can be used in a spike. When a capacitor is used this way, it's called a **bypass capacitor,** because the part of the circuit it serves is effectively bypassing the power source.

Additionally, there can be noise on the path from one part of a circuit to another. If the "noise signal" oscillates, a capacitor can be used to short that signal to ground before it can enter. When a capacitor is used for this purpose, it's called a **decoupling capacitor**, because the part of the circuit it serves is isolated from noise in the rest of the circuit.

Finally, a capacitor placed between a voltage supply and ground can help to stabilize the voltage.
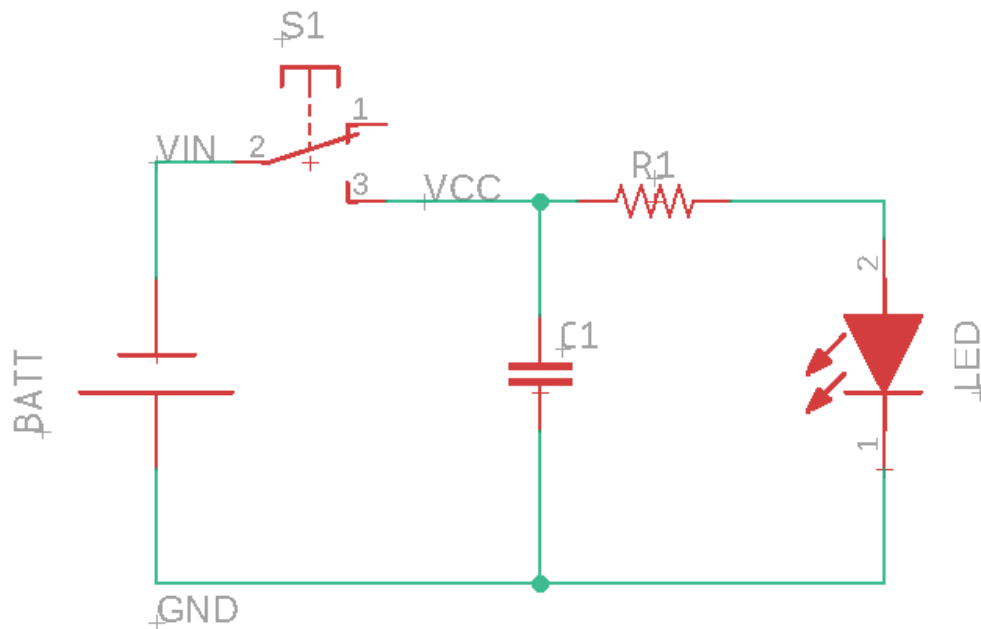
Capacitance is measured in Farads (F). There is no simple rule for calculating the required value of a bypass or decoupling capacitor. Typically, powers of 10 are used, and lower values are better at filtering higher-frequency noise and vice-versa. A typical bypass capacitor has a value of 0.1 µF.

Capacitors can be **polar** or **nonpolar**. The capacitor above is nonpolar. Polar capacitors have a positive and negative terminal.
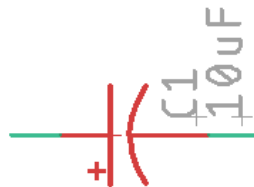
Digital Communication

Some components in a DC circuit are capable of **digital communication**. In digital communication, a signal (an electrical connection) that is close to or equal to VCC is interpreted as a 1, and a signal that is close to or equal to GND is interpreted as a 0. We call the signal **binary** because there are only two ways it can be interpreted.

We can use binary signals to communicate larger numbers. Let's take a look.

Binary Encoding

Consider the number 132. There is a hundreds place, tens place, and ones place. Because each digit is a multiple of a power of 10, we say that the number is written in base 10.

10^2	10^1	10^0
1	**3**	**2**

In binary, each digit is a multiple of a power of 2. We say that the number is written in base 2.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	**0**	**0**	**0**	**0**	**1**	**0**	**0**

When a number is encoded in base n, each digit can only be as high as $n - 1$.

To communicate a number, we can either use a binary signal for each digit or change the value of a single signal over time.

To communicate text, we can represent letters as numbers using an agreed-upon **encoding**, and then continue sending numbers. This is true for all kinds of information.

A 0 or 1 in a binary message is referred to as a **bit**. Eight bits (as above) comprise a **byte**.

When a signal is 0 or 1, we say the signal is **low** or **high**, respectively.

The Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI) is a protocol for sending data by changing a single binary signal over time. It requires an additional control signal in order to work. It can operate at very high speeds but only over relatively short distances.

The **clock** signal (purple) indicates when the sender is setting up the next bit on the data signal and when that bit is ready. The data signal can be in a transient state when the clock is low. As soon as the clock signal is high, the data signal can be read by the receiver.
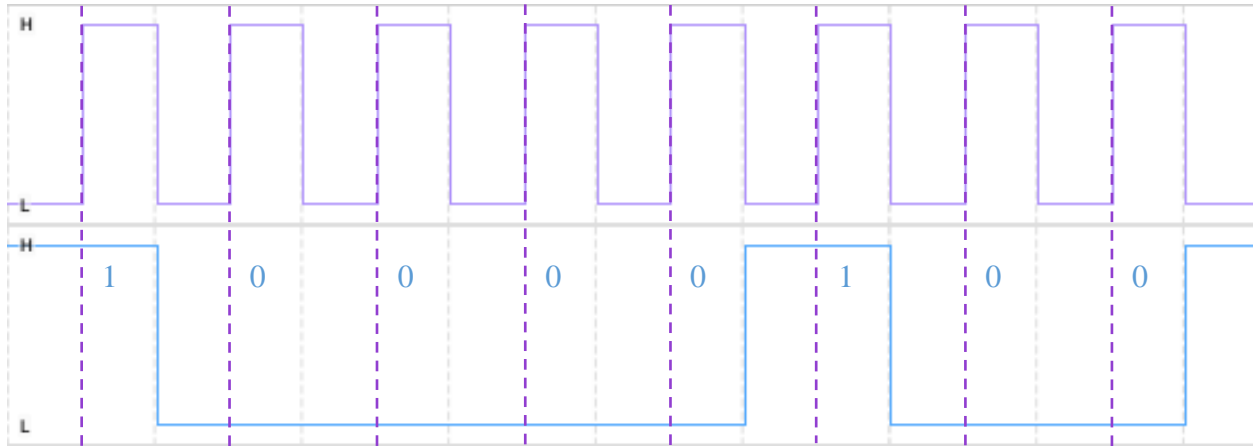
SPI defines one end of the communication as the **master** and one end as the **slave**. The master drives the clock signal and as such it is always the initiator of communication. The master sends data on one signal and receives data on another signal, synchronized to the clock. Since these are separate signals, they can be active at the same time, meaning the master can simultaneously send and receive one bit in one clock pulse.

The clock signal is commonly labeled SCK (Serial Clock). The data signals are called MOSI (Master Out Slave In) and MISO (Master In Slave Out).

Multiple slaves can be connected to a single master. The SCK, MOSI and MISO signals are shared between the slaves. We say that the slaves are attached to the master's SPI **bus**.

We use an additional signal for each slave, called its **chip select**, to instruct the slave to listen to MOSI and respond on MISO. The master drives the chip select signals too and is responsible for ensuring only one slave is selected at a time. When a chip select signal goes low, the slave it's attached to is selected. The signal goes high only when communication with that slave has completed.
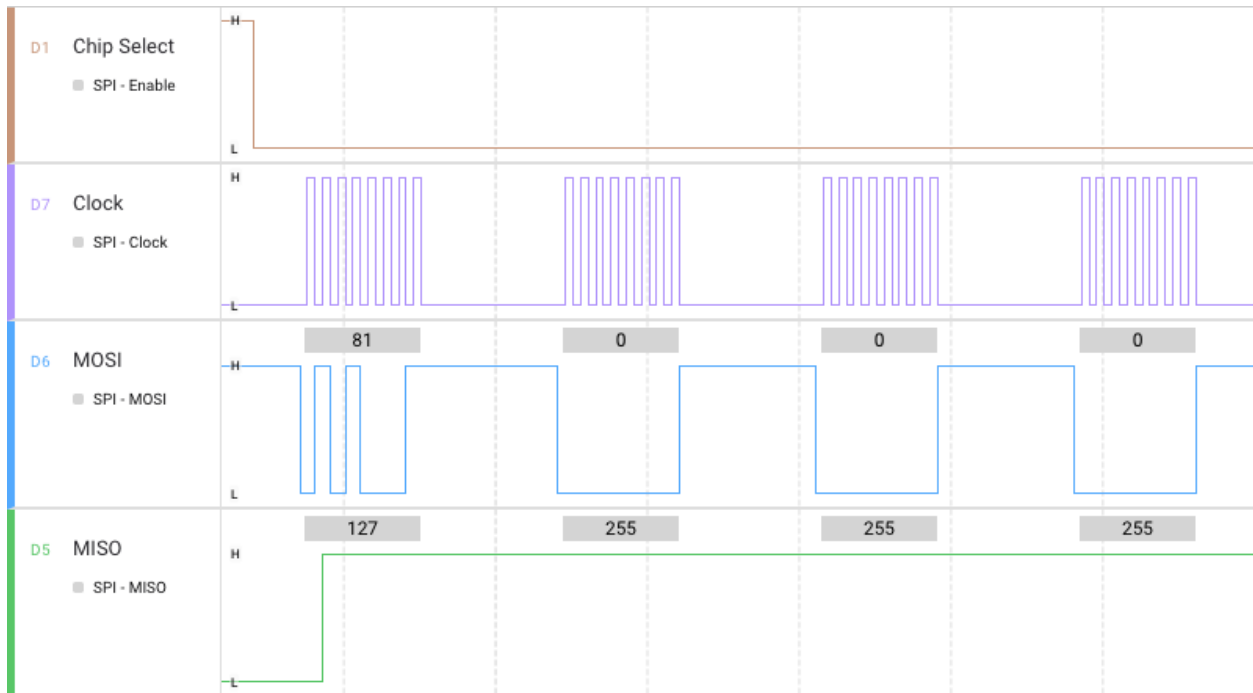
Above is shown the master sending and receiving 8-bit numbers after asserting chip select.

PC Serial Communication

Serial communication can be performed without a clock signal if the sender and receiver agree on a time delay between data bits. This is less reliable at higher speeds but generally works up to 115,200 bits per second. This is more than enough for a low-speed device such as a text display or keyboard.

The device we will be looking at will connect to a computer to display information and receive commands. The device will connect over USB but will emulate a PC serial port for simplicity. Historically, PC serial ports have used the bitrate scheme as opposed to clock signal. This is perhaps due to the fact that a high-frequency signal, such as the clock signal, degrades over relatively long distances, such as the length of a 3-6 ft cable.

On the serial port, a device again has separate transmit and receive signals, named TX and RX respectively. They're not on a bus; these lines are not shared with other devices, so there is no need for a chip select. The data signals are **full-duplex**, meaning bits can be transmitted and received simultaneously as with SPI. Some devices use additional signals in order to perform **handshaking**. These control signals are used to indicate when one side has data to send and when one side is ready to receive data. The music player's serial port only uses TX and RX.

Parallel Communication

We can also communicate an 8-bit number using a separate signal for each bit. In this scheme the chip select and clock functions are typically combined into a single **chip enable** signal that is also sometimes called chip select. A number is written on the data signals and chip enable is brought low for a short time and then back high. Since all 8 data bits are written at once, parallel communication can be much faster than serial communication.

Parallel data signals are also called a bus. A parallel data bus is often bidirectional, with a separate signal or the data protocol dictating the bus direction at any given time.

Gates

Imagine a circuit with two parts. Each part has a binary signal to indicate if it's ready for some operation. 0 means "not ready" while 1 means "ready". The operation has a single input signal to control when the operation is performed. 0 means "wait" while 1 means "perform". How do we connect the two ready signals to the single "perform" input?

To capture when both parts of the circuit are ready, we use a **gate**. Specifically, we use an AND gate to capture when one part *and* the other part are ready.
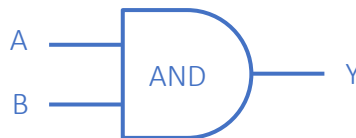
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

If we pass the Y signal into the operator, the operator will only be activated when both parts of the circuit are ready.

AND is called a **logical operation**. There is also the OR gate which outputs 1 when *either* A or B are 1. We will also see a NOT gate which takes a single binary input and produces the opposite output.

The Hardware

We're now ready to take a look at the schematic of the music player.

The Power Circuit

The player connects to a computer over USB in order to receive commands such as pause and skip. The player is powered by a supply line in the USB cable that is in turn powered by the PC. The music player uses its USB port as the source of power. If the ability to control the player from a computer isn't needed, the player can be plugged into a standard USB charger or battery.

Most of the player's internal circuits operate at 5 V. Some of them operate at 3.3 V. The USB cable will supply somewhere between 4.9-5.2 V. A **voltage regulator** can convert a voltage in a certain range to a fixed voltage below the range. A **boost converter** can convert a voltage in a certain range to a fixed voltage above the range. The player uses a pair of regulators that are capable of both down conversion and boost conversion. One of them regulates to 5 V, and the other to 3.3 V. They come from a family of devices called TPS6120x.

Often the datasheet for a component will provide an example schematic. The power circuit for the music player was derived from the example schematic in the datasheet and other freely available schematics for boost conversion devices using TPS6120x. Here's what the TPS6120x datasheet provides:
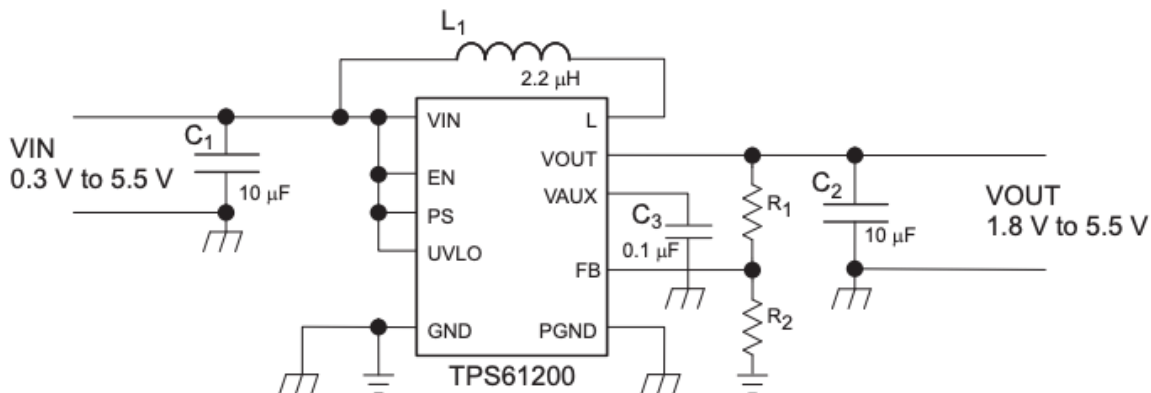
The TPS61200 has a programmable output voltage that is controlled by the values of R1 and R2. The music player uses TPS61201 and TPS61202, which have fixed outputs at 3.3 V and 5 V, respectively. As such R1 and R2 are not incorporated in the power circuit and FB is connected directly to VOUT at the instruction of the datasheet.

L1 is a device called an inductor which stores energy in the magnetic field surrounding it, kind of like an inverted capacitor.
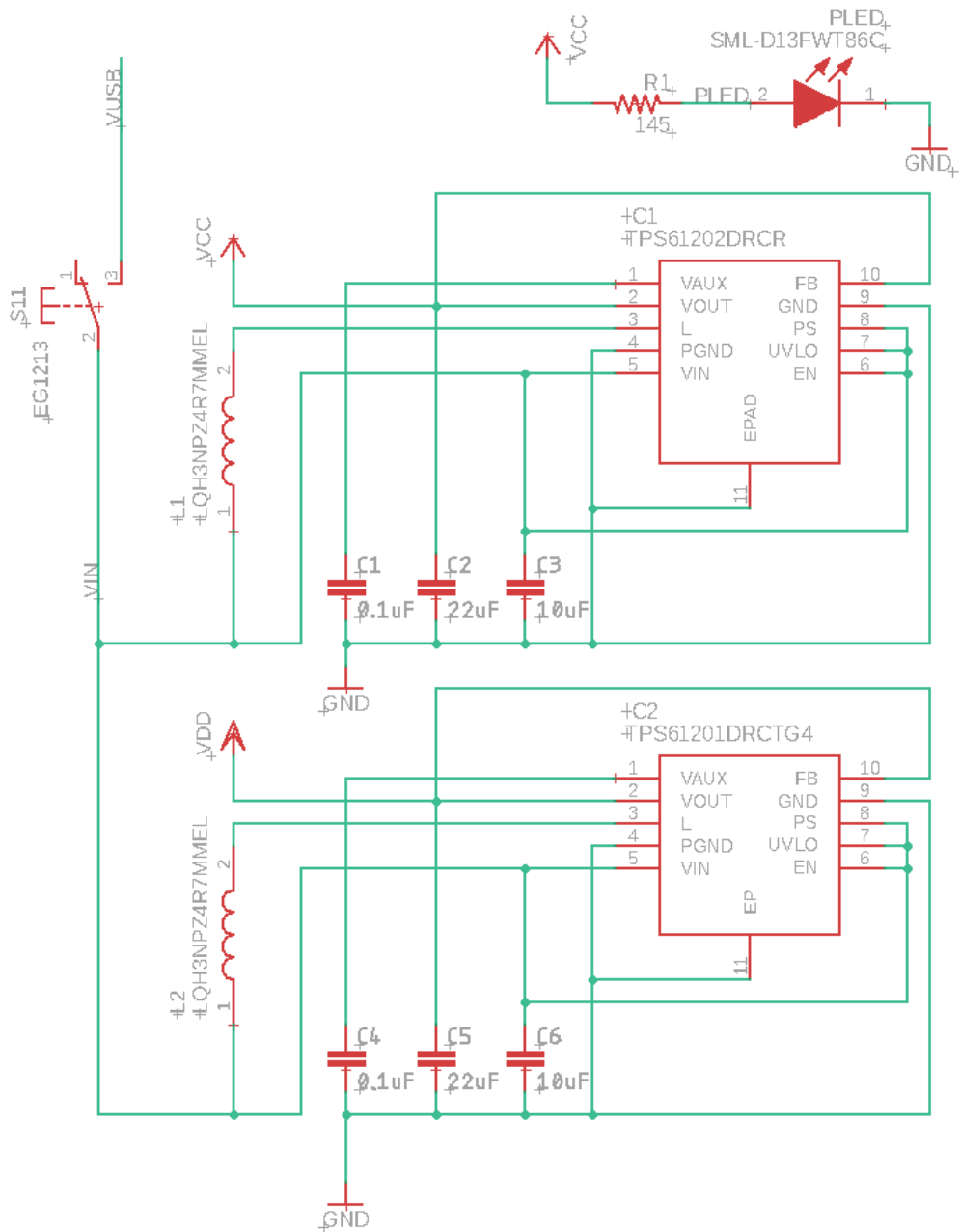
The USB power supply is gated by a power switch. When on, it supplies VIN for both regulators. The top regulator produces VCC (5 V) for the music player on the VOUT line while the bottom regulator produces VDD (3.3 V). The regulators share a common ground that is connected to the ground line of the USB cable.

Note the different symbols used for VCC, VDD and GND. We'll see them in each of the circuits in this book.

The music player has a set of eight general-purpose LEDs that are controlled via SPI.

A **shift register** is a device that accepts a serial binary input (clock and data) and outputs the bits it receives on independent outputs. A shift register is not a SPI device, but its clock and data inputs are compatible with SPI. The LEDs are attached to the outputs of a shift register which in turn is attached to the SPI bus.

A register stores one byte of data (8 bits). A shift register is so called because it shifts each bit further into the register with each clock pulse. The first bit it receives goes into the first bit position (2^0). After all bits have been shifted in that bit is in the last position (2^7).
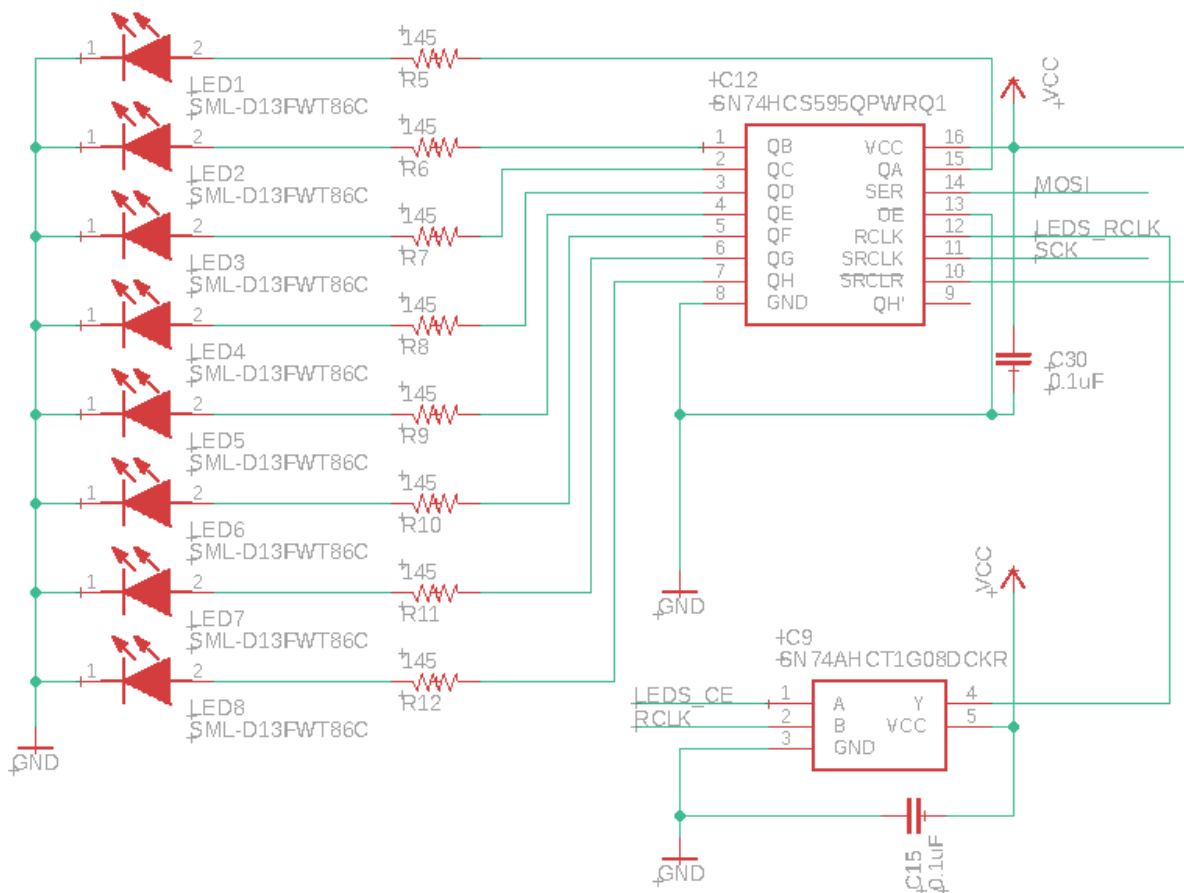
This schematic is a little unconventional because information passes from right to left.

MOSI and SCK are attached to the data and clock inputs of the shift register IC12.

$Q_A - Q_H$ reflect the data bits $2^0 - 2^7$ received and are passed on to the LEDs.

The first bit makes its way from Q_A to Q_H as the clock signal is pulsed, and the remaining data bits follow.

A shift register contains a second register referred to as the **storage register**. The storage register value is what is actually presented on $Q_A - Q_H$. The contents of the shift register are transferred to the storage register when the RCLK signal is brought high.

One way to control RCLK is to turn it on after all eight bits have been shifted in. But it might be desirable to update the storage register each time the shift register shifts. The sequence is the clock signal goes high to bring in a bit, then the RCLK signal goes high to transfer the contents of the shift register to the storage register. The RCLK signal actually ends up being the inverse of the clock signal (SRCLK below).
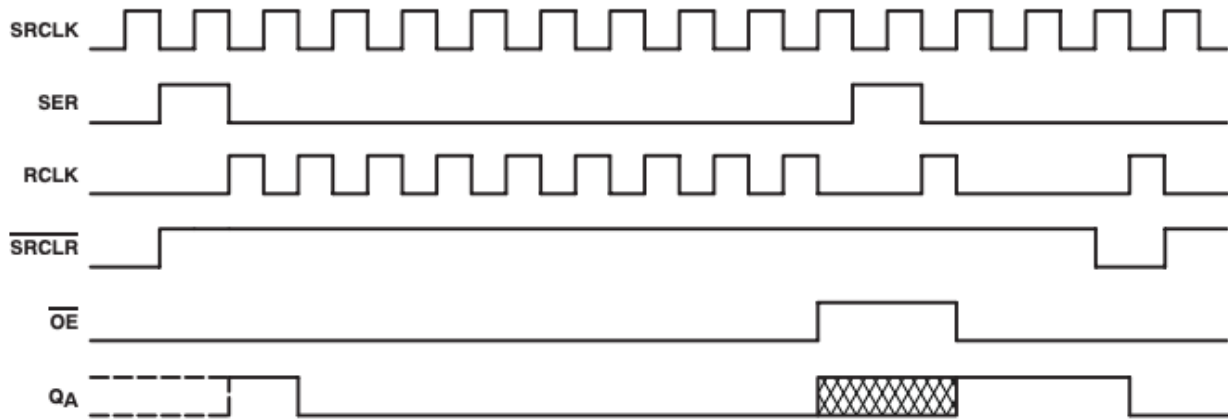
Notice that Q_A only reflects the 1 that was shifted in once RCLK goes high.

SCK is attached to SRCLK above and the RCLK signal is generated by a NOT gate attached to SCK. The LEDs have a chip enable signal. RCLK and the chip enable are fed to an AND gate which is fed to the RCLK input of the shift register. The contents of the shift register are only ever transferred to the storage register when chip enable is active. Chip enable ultimately controls whether the LEDs show what is sent on the SPI bus.

There are two ways to use the chip enable. Enable it before sending a byte over SPI to keep the LEDs as up to date as possible. This makes it more like a chip select signal. Enable it after sending a byte over SPI to only update the LEDs once all eight bits have been shifted in. This works because RCLK is 1 after a SPI transfer which we'll see later. Both inputs to the AND gate are 1.

The LEDs' chip enable signal is different from the others in the sense that it is active when high, not low.

The Sound Generators

The music player has two digital sound generators Yamaha YM2612 and Texas Instruments SN76489. Both chips have a parallel data interface. These devices only receive data, so these buses are unidirectional. That makes it possible to attach a shift register to write to them using fewer signals.

Both chips have a chip enable pin. The chip only reads its data bus when chip enable is brought low. That makes it possible to put both the YM2612 and SN76489 on the same bus, i.e., on the same shift register. The pattern is to write D0-D7 to the SPI bus for one of the chips, then select the chip that is the intended recipient.
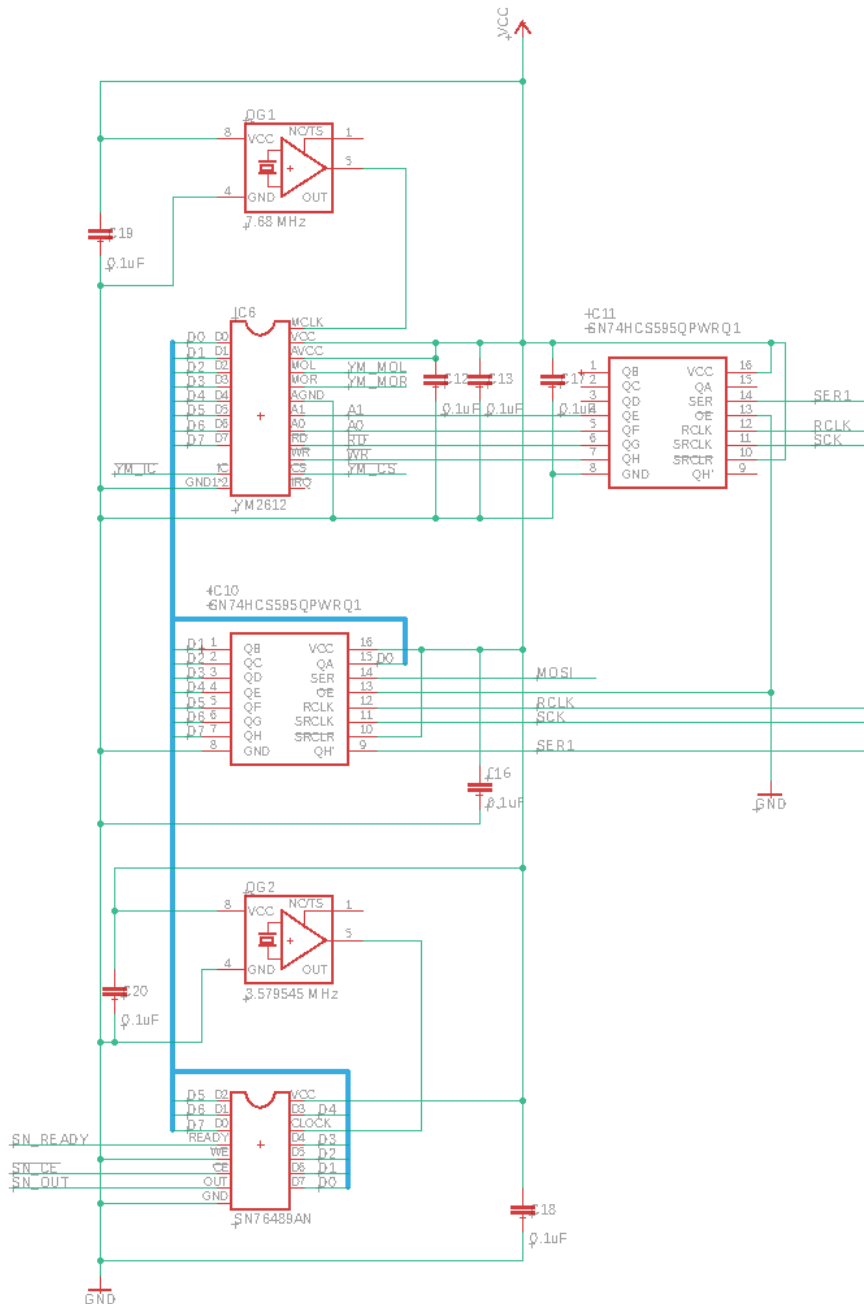
Four of the YM2612's control signals can also benefit from a shift register. $\overline{\text{YM\_IC}}$ and $\overline{\text{YM\_CS}}$ do not because the shift registers are wired directly to RCLK, without a gate, meaning their outputs have transient values until a SPI transfer is completed. $\overline{\text{YM\_IC}}$ and $\overline{\text{YM\_CS}}$ are the reset and select signals for the YM2612, and they take effect as soon as they're enabled so they cannot be transient.

There are actually two shift registers chained together to form a 16-bit register. The highest four bits in the second register are connected to four of the YM2612's control signals. To write to the YM2612 requires sending the control byte over SPI, followed by the data byte, before signaling chip enable.

Apart from the sound generators and the shift registers are shown two quartz crystals. Each sound generator requires a clock signal to drive its time-based behavior which is what these crystals provide.

The YM2612 outputs a stereo analog audio signal on MOL and MOR.

The SN76489 outputs a mono analog audio signal on OUT.

Later we will look at a circuit that filters, mixes, and amplifies the audio before sending it to the speaker.

The SD Card

The music played by the music player is stored on a microSD card. SD cards have a proprietary interface, but luckily, they all support SPI as well (albeit at a lower speed). The SD card runs on 3.3 V and needs a 5 V to 3.3 V translator for the SPI signals in order not to damage the card. It's okay to wire a 3.3V MISO directly to a 5V SPI bus, however. The bus uses $\frac{VCC}{2}$ as the boundary between 0 and 1, and 3.3V is above that boundary for a 5V bus.

+VDD

MISO

+C4
SN74LVC245APWR

1	DIR	VCC	20
2	A1	OE	19
3	A2	B1	18
4	A3	B2	17
5	A4	B3	16
6	A5	B4	15
7	A6	B5	14
8	A7	B6	13
9	A8	B7	12
10	GND	B8	11

SCK
MOSI
SD_CS

SCK'
MOSI'
SD_CS'

C11
0.1uF

+GND

CD_SW1 9
DAT1/X 8
DAT0/DO 7
VSS 6·5
CLK/SCLK 5
VDD 4
CMD/DI 3
DAT3/CS 2
DAT2/X 1

microSD
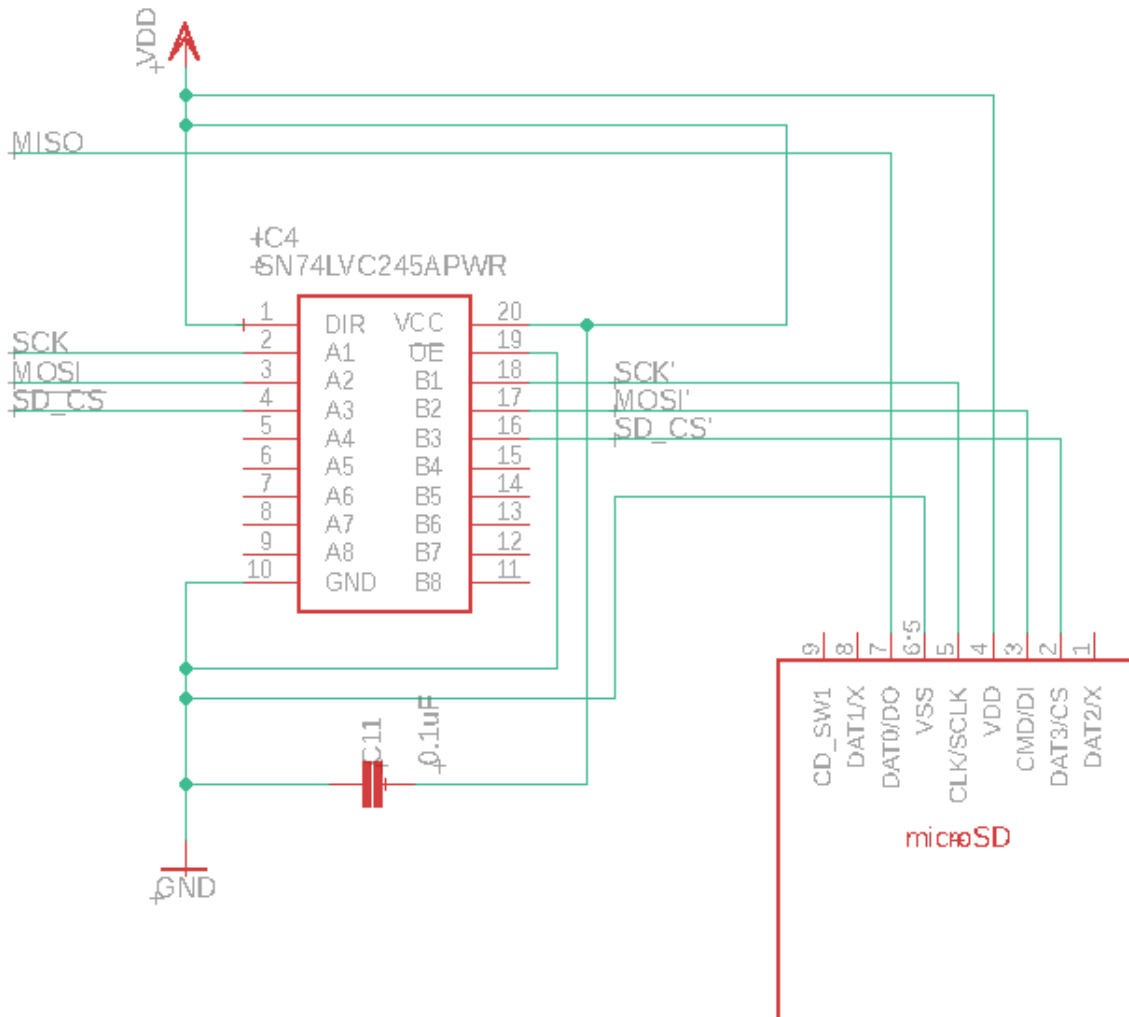
The Memory Unit

The songs are played directly from the SD card but as we will see in the software chapters it will actually be necessary to store part of a song in a separate memory unit which can be accessed more quickly.

The memory unit has two parallel buses, one for specifying an **address** and the other bus for conveying the **data** byte stored at that address. The data bus is bidirectional so we can write to it and read from it. As such we will use its signals directly. The address bus is unidirectional. The address bus is put on the SPI bus with shift registers to reduce signal count.

The memory unit has 512 KB capacity, which requires 19 address signals SRA0-SRA18. The lowest 16 of these signals are attached to shift registers, while the upper 3 are used directly.
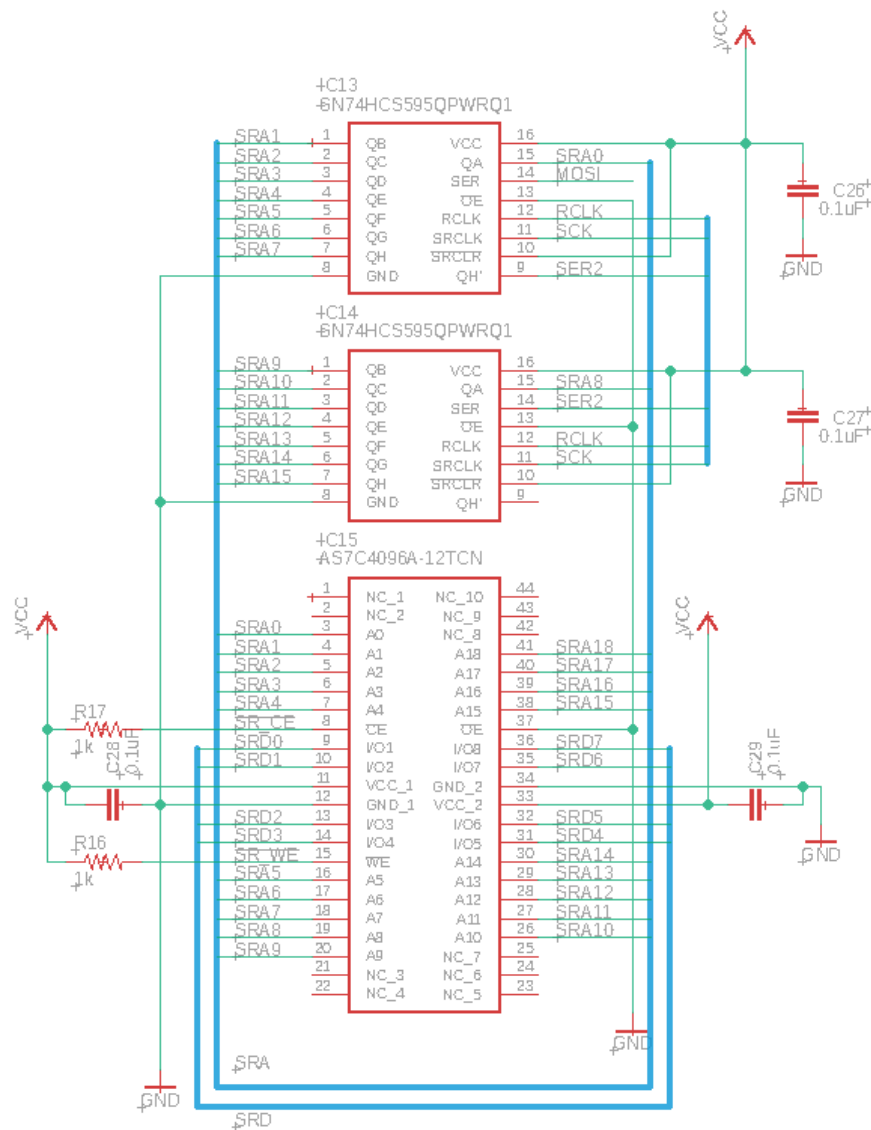
The resistors in this circuit are called **pull-up resistors** and they ensure the signal they're connected to is held high when it isn't in use, by providing a weak connection to VCC.

The Microcontroller

We've looked at most of the circuits required by the music player. Those that provide input or output to the player are on the SPI bus. Now it's time to introduce the SPI master. This is a **microcontroller** called the AVR ATmega644P. A microcontroller is a **processor** that has **peripherals**. These are internal hardware devices that are controlled using internal registers.

The microcontroller has internal **program memory** that we will be programming in the software chapters. We'll be writing a program that reads music from the SD card and plays it through the sound synthesis chips.
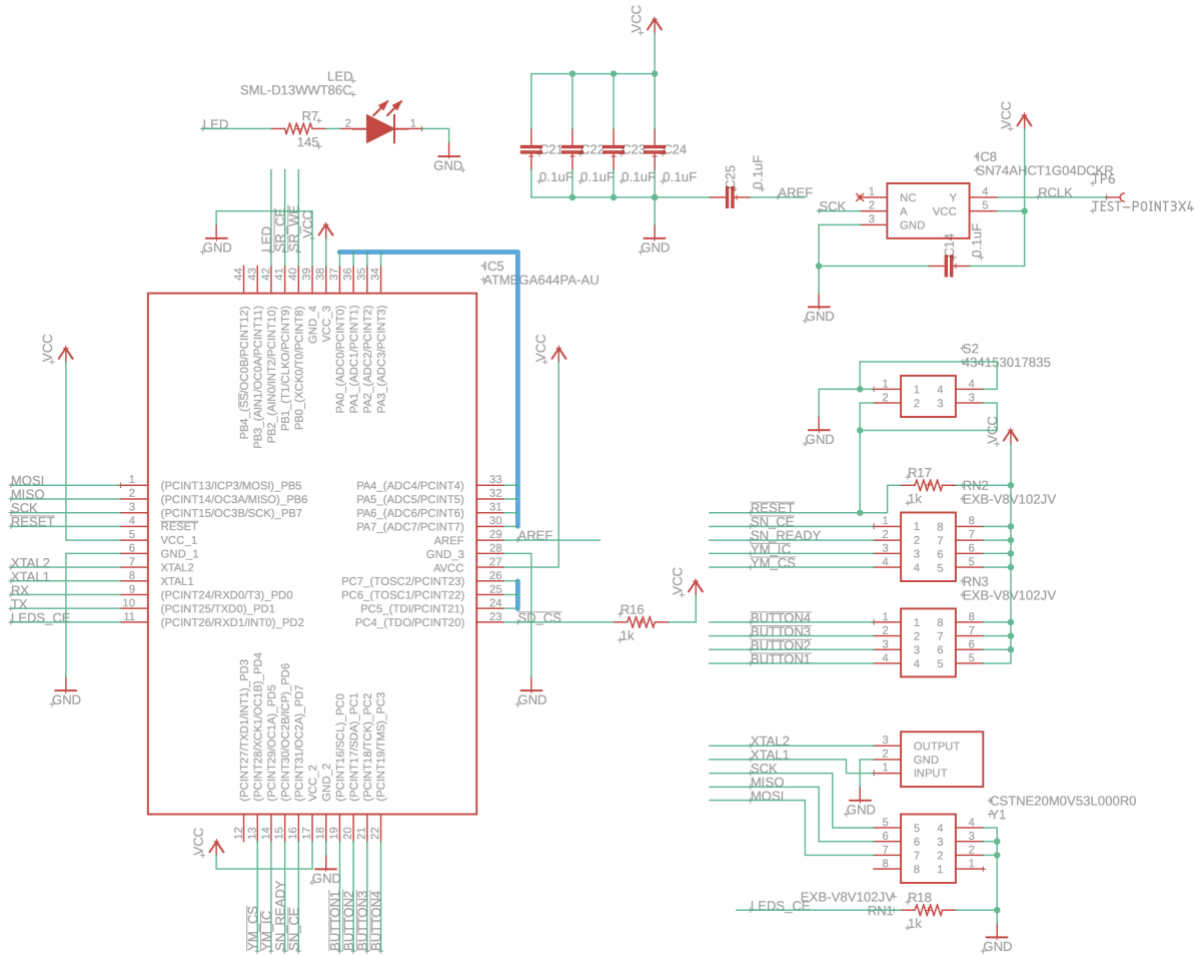
In the top left is an LED that is separate from the eight general-purpose LEDs.

The four capacitors shown in isolation are actually each connected across one VCC-GND pair on the microcontroller. They're shown that way to reduce clutter in the schematic.

In the top-right is the NOT gate mentioned earlier that converts SCK into RCLK.

Below that is a button that is used to temporarily pull the microcontroller's $\overline{\text{RESET}}$ signal low to reset the microcontroller.

Below the button we see a **pull-up resistor** that ensures $\overline{\text{RESET}}$ is normally high by providing a weak connection to VCC.

RN2 and RN3 are resistor networks. Each horizontal pair of terminals are the terminals of one resistor. RN2 provides pull-up resistors for control signals belonging to the YM2612 and SN76489. RN3 provides pull-up resistors for the on-board buttons.

Below RN2 and RN3 is a quartz crystal that drives the microcontroller's time-based behavior.

Below that is another resistor network RN1 of **pull-down resistors** that pulls the SPI bus low when it isn't being used. In addition to reducing noise on the bus, it ensures that SCK is brought low following the 8$^{\text{th}}$ bit transferred, which means RCLK will be brought high to transfer shift register contents to the storage register.

Below RN1 is a pull-down resistor for the chip enable signal for the general-purpose LEDs. Remember that this signal is high when active.

Pins 30-37 of the microcontroller are connected to the memory unit's data bus. Pins 24-26 are connected to the highest three address signals of the memory unit (there are 19 total and 16 are connected to shift registers on the SPI bus).

PC Communication

As mentioned earlier, the music player will connect to a computer to display information and receive commands. The device will connect over USB but will emulate a PC serial port for simplicity.
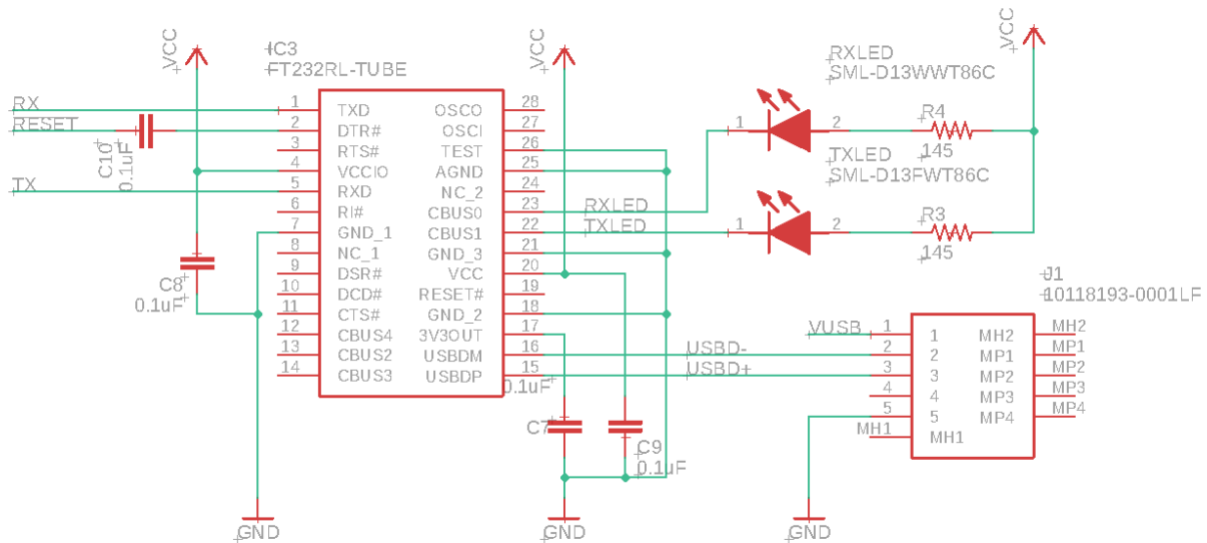
IC3 is the FTDI FT232RNL and it acts as a serial port attached to the computer over USB. In the lower right is shown the USB connection and in the upper left are the RX and TX connections to the serial port of the music player. Notice that the TX on the computer side is connected to the RX signal on the microcontroller and vice-versa.

The Buttons

The Chromasound has four general purpose buttons. Each button has a blue LED above it. Each button connects directly to its own input on the microcontroller.
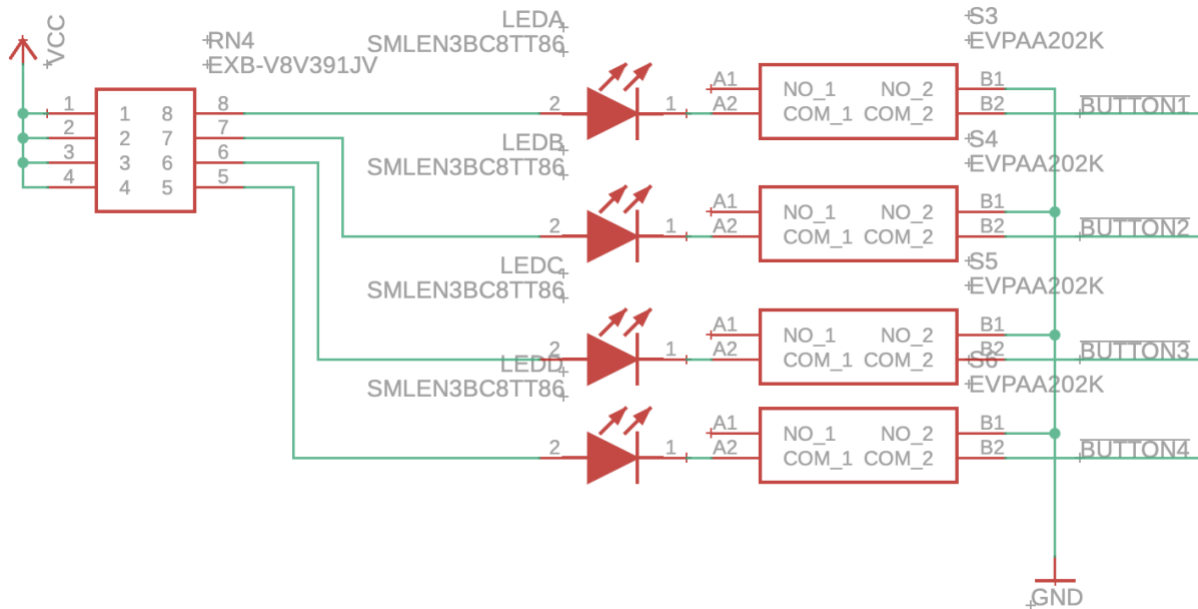
RN3 in the microcontroller schematic pulls the button signals up so they are normally high. When a button is pressed, its signal is shorted to GND. We can tell when a button is pressed by checking the level of the microcontroller input the button is connected to.

Each of the two horizontal pairs of pins on the buttons are internally connected. Pressing the button connects the pairs.

RN4 lowers VCC to a voltage and current that the LEDs are rated for. The negative pin of an LED (the **cathode**) is connected to the button signal. Normally, the button signal is high, so current does not flow through the LED. When the button is pressed, the button signal goes low, and current does flow through the LED, illuminating it.

The buttons are used to skip and pause the current song (as can be done over the USB connection). They are also used by a piece of software called the **bootloader** which we will see in the last chapter.

The Amplifier

Earlier we saw that the YM2612 outputs a stereo analog audio signal on MOL and MOR and the SN76489 outputs a mono analog audio signal on OUT. We need to mix these signals into a single stereo signal.

The YM2612 is also much quieter than the SN76489 and isn't really capable of driving a speaker. We need a device that can amplify the YM2612 and mix it with the SN76489.

The device used is called an **operational amplifier**.
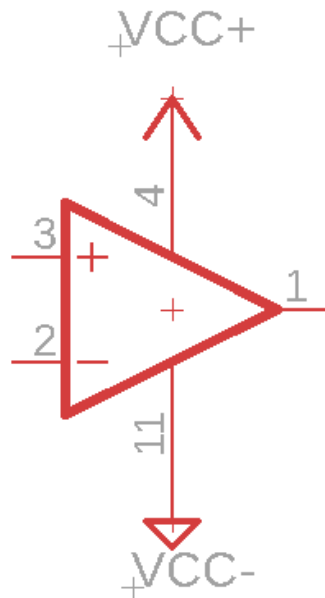
An operational amplifier has a pair of **differential inputs** on the left and one **single-ended output** on the right. It has an inherent amplification factor called its **open-loop gain** (A_OL). The op amp amplifies the difference between the voltages at the inputs. The bottom input (the **inverting input**) is subtracted from the top input (the **noninverting input**) and the difference is amplified. The resulting voltage is presented on the output.

$$V_{OUT} = A_{OL}(V_{in+} - V_{in-})$$

V_{OUT} is negative if V_{in+} is smaller than V_{in-}, which is why the op amp needs both a positive and negative supply.

The A_OL of the operational amplifier in the music player is just under 1778280. At first this seems impossibly high. If we connect a 1 mV signal to the noninverting input and connect the inverting input to ground, the output signal is 1778.3 V! However, VCC can't be higher than $\pm$ 40 V. The output is **clipped** to a value just under VCC. This is the maximum that the op amp can provide.

We want to amplify an audio signal. We want a much smaller amplification factor so that the output doesn't **saturate**, or reach the VCC boundary, given the input audio signal. The output should be a louder version of the input within the VCC+ to VCC- range.

Another consideration is that the **frequency response** of the operational amplifier is inherently low, so some frequencies in the audio signal may not be amplified.

As it turns out, we can trade some of the amplifier's open-loop gain for an increased frequency response (as well as a few other benefits). We can do this by employing **negative feedback**. In negative feedback, a portion of the output is *fed back into* the inverting (subtracted) input.
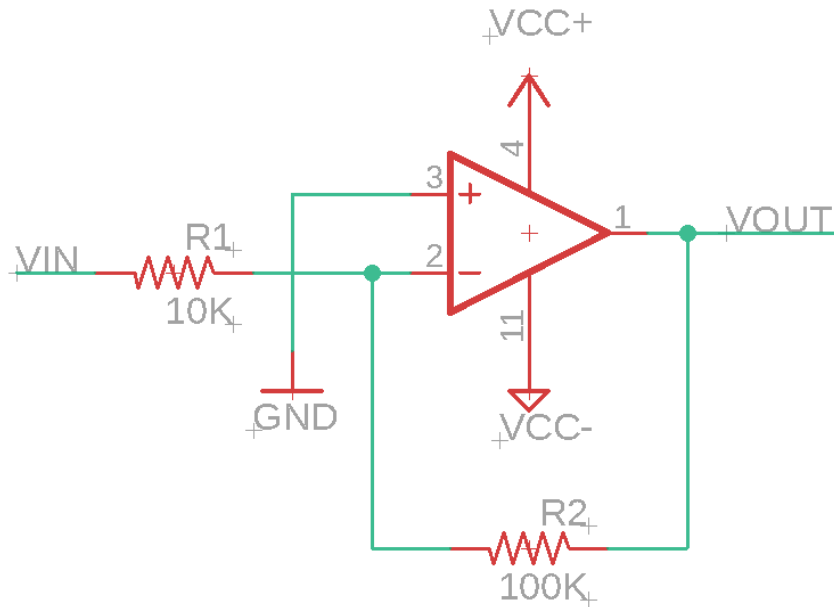
This circuit is called an **inverting voltage amplifier**. The op amp's gain in this configuration is referred to as its **closed-loop gain** (A_{CL}). The closed-loop gain is equal to the ratio of the resistor values R2 and R1. Because the input signal is on the inverting input, the amplified output is inverted.

$$V_{OUT} = -A_{CL}V_{IN}$$

$$V_{OUT} = -\frac{R2}{R1}V_{IN}$$

Using the values in the schematic above,

$$V_{OUT} = -10V_{IN}$$

An audio signal is not perceived differently if it is inverted. There is an effective amplification of 10 in this circuit. In addition to bringing the gain down to 10, the negative feedback ensures this gain for a band of frequencies that span the range of human hearing. In other words, the frequency response of the op amp is improved.

A single op-amp can be used to amplify and mix multiple analog signals.
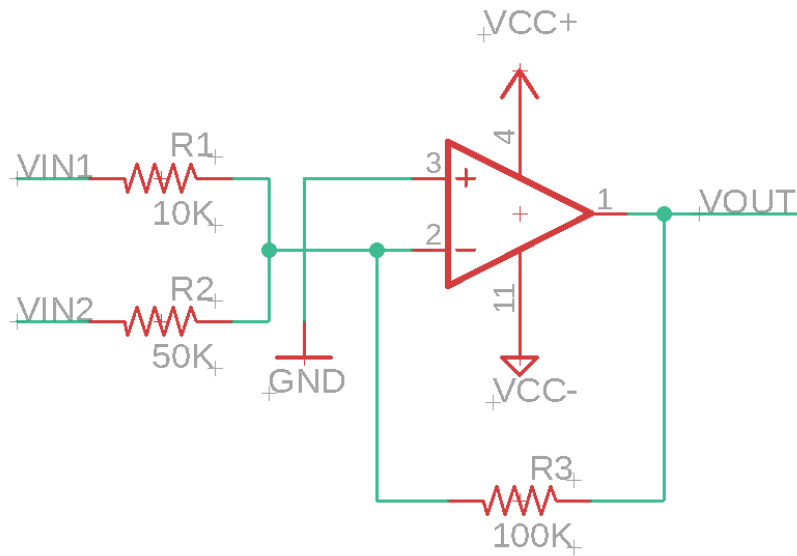
$$V_{OUT} = \left(-\frac{R3}{R1}V_{IN1}\right) + \left(-\frac{R3}{R2}V_{IN2}\right)$$

This is called a **summing amplifier**. Using the values in the above schematic,

$$V_{OUT} = -10V_{IN1} + -2V_{IN2}$$

This can be extended to any number of input signals.

It's possible that the circuit receiving V~OUT~ will have undesired effects on the op-amp. These effects are known as **loading effects**. To isolate the amplifier from loading effects, we use another op-amp in a configuration known as a **voltage follower** or **buffer**. In a voltage follower, there is unity gain (no amplification), so the signal is not modified. All load is placed on the op amp not performing any amplification.
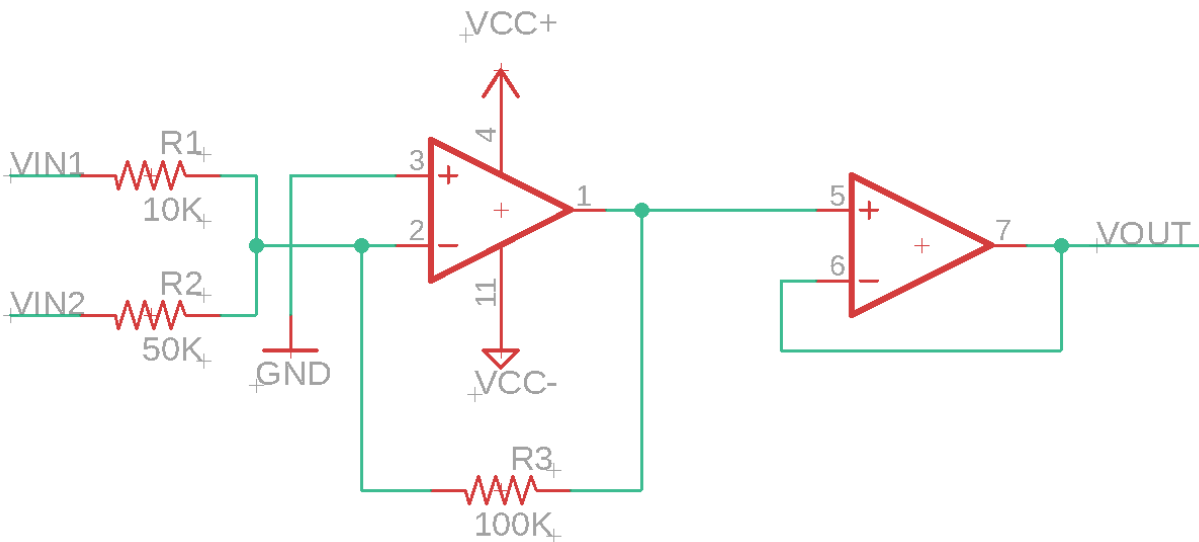
These two op amps are in a group of four that are contained in the same chip, which is why there is only one pair of supply lines, and why the pin numbers on the right op amp continue from those on the left.

The SN76489's OUT signal is moderately loud. Experimentally, the YM2612's volume is matched when amplified by a factor of 10. The approach I took was to amplify the left and right YM2612 signals by a factor of 10. Each signal is mixed with the SN76489's OUT signal, whose resistor is matched to the feedback resistor on the op amp. The SN76489's signal is therefore mixed in with unity gain (no amplification).
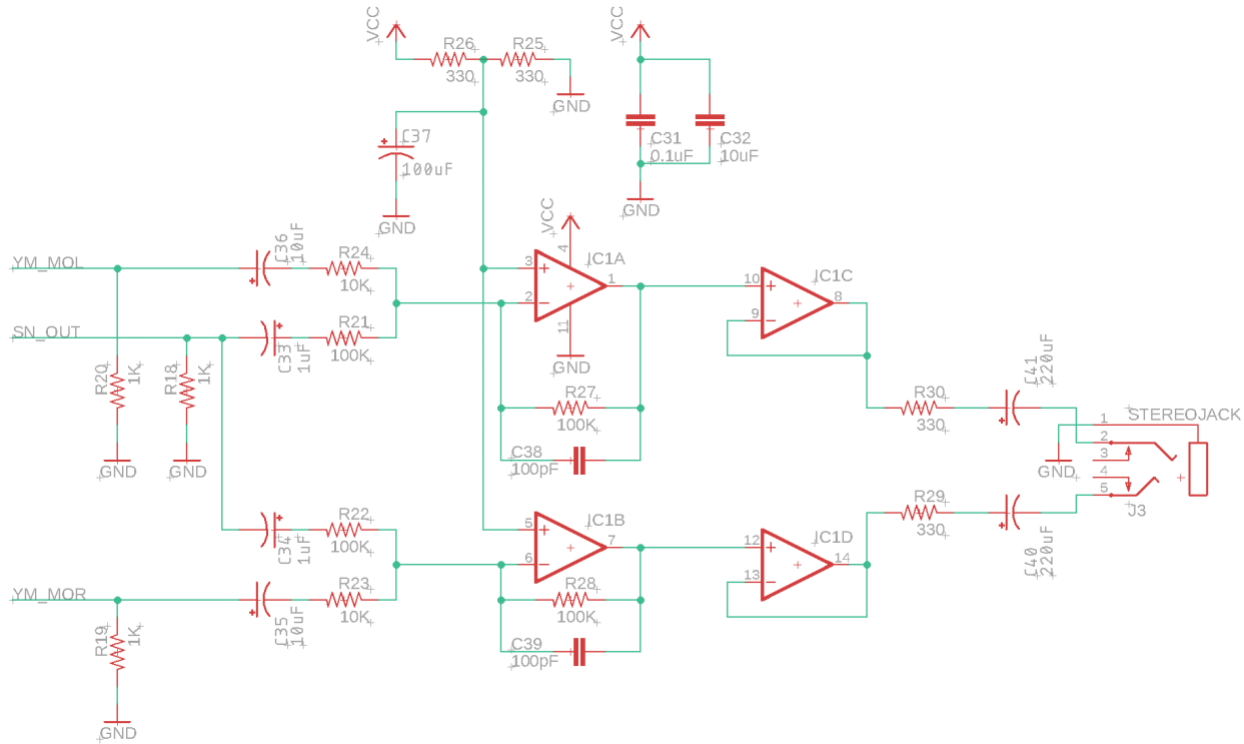
In the previous schematics there was a positive and negative supply line, and the analog inputs and outputs were relative to ground (0 V). The sound generators only have a positive supply line, so their analog outputs are relative to $\frac{VCC}{2}$. As such $\frac{VCC}{2}$ is provided to the left-hand op amps on the noninverting input instead of ground. In addition, the negative supply line of the op amps is connected to ground. We say the op amps are in **single-supply operation**.

C31 and C32 are decoupling capacitors on the supply lines. C37 improves the stability of the $\frac{VCC}{2}$ supply. The remaining capacitors are filters, allowing only some AC signals to pass based on capacitance value. DC signals are blocked by these capacitors. Without C40 and C41, **DC bias** on the left and right output signals will actually prevent a listener from hearing anything.

The three leftmost resistors are pull-down resistors that help reduce noise on the outputs of the sound generators.

There are two issues with the quad op-amp amplifier design as it stands. One is that in some installations a loud hiss can be heard amplified on the output.

Here's how we can address this problem. A hiss is essentially high-frequency noise. We can add a **low-pass filter** to the amplifier with a fixed cutoff frequency. The filter lets through all frequencies below the cutoff and attenuates (reduces the volume of or silences) frequencies above the cutoff. 10 kHz is a good cutoff value because most frequencies above this threshold are harder for the human ear to perceive. To add the filter, we replace the buffer op-amp with a different op-amp design called a **Single-supply Butterworth Sallen-Key Low-pass Filter**.
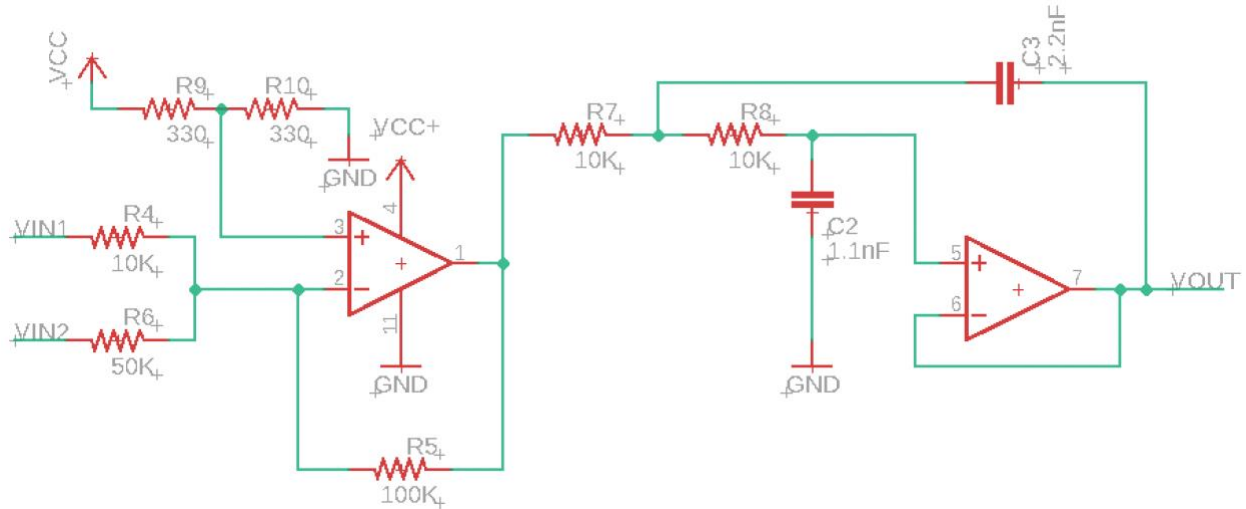
The resistors and capacitors after the first op-amp (R7, R8, C2, C3) configure the filter op-amp for a 10 kHz cutoff frequency.

The second design issue is that the amount of current the op-amp can provide on the output is small (26 mA). That's large enough to be able to drive headphones, but if we want to add even a small speaker to the system, we will need to increase the output current somehow. It's actually the **output power** that we want to increase, where power is the product of the signal voltage and the maximum current drawn by the speaker at that voltage.

The amplifier can only deliver up to 5 V * 26 mA = 130 mW of output power. A speaker that is a moderate size relative to the circuit board will require about 800 mW. To address the output power problem, I've added an audio amplifier chip that is designed to provide up to 1 W of output power.

The first op-amp in the amplifier is now a **preamplifier** whose job is to mix audio signals and bring them up to **line level**. The second op-amp is a **low-pass filter** on the output of the preamplifier. The power amplifier chip backs the filtered audio signal with more current and can be used to further amplify the audio beyond line level.
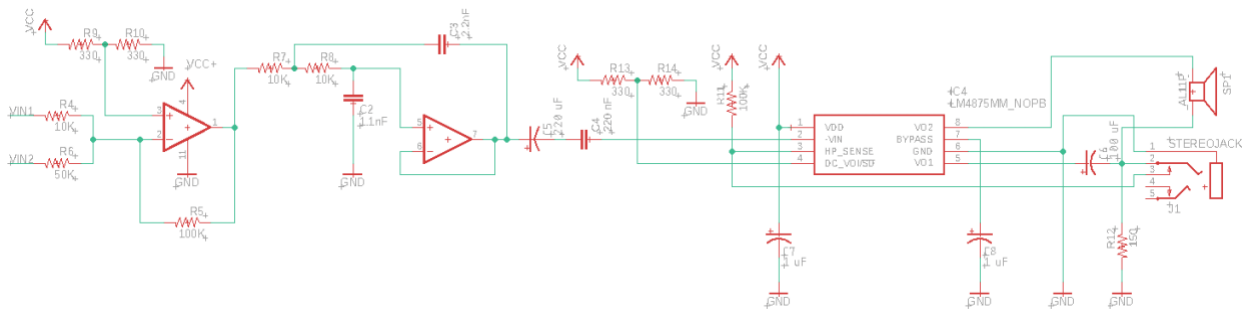
The power amplifier chip is a Texas Instruments LM4875MM. The HP_SENSE pin detects when headphones have been inserted into the stereo jack, using a resistor voltage divider (R11 and R12). When headphones are not present, the HP_SENSE pin is low and LM4875MM provides a powerful audio output to the speaker. When headphones are present, the HP_SENSE pin is high and the LM4875MM disables the speaker and provides audio output to the headphones. The DC_VOL/SD pin controls the volume of the output. At $\frac{VCC}{2}$ the amp is configured for unity gain (0 dB). The R13-R14 voltage divider can be replaced with a volume knob that wipes between 0.8 V and 3.6 V.

Check out the datasheet for the power amplifier if you want to learn more about it. Something interesting is that the VO2 pin is 180° out of phase with VO1. That means that for a given signal voltage on VO1, the voltage across the speaker is double, resulting in four times the output power when compared to headphone mode.

The schematic is doubled (except for the speaker and stereo jack) to handle both left and right audio signals.

Chromasound Pro
Nir Jacobson • 2024
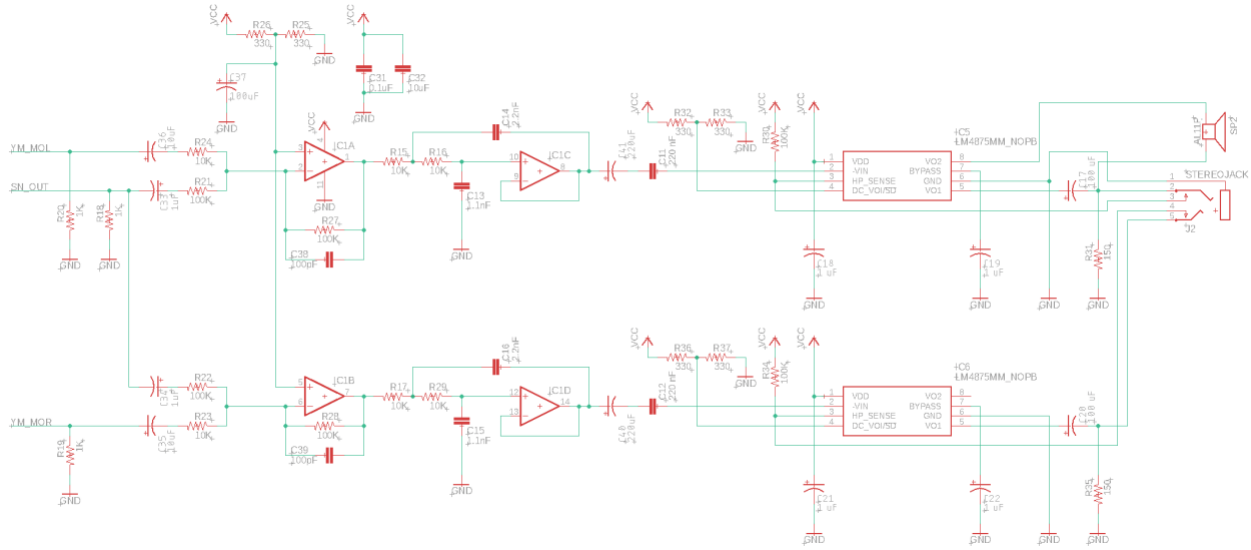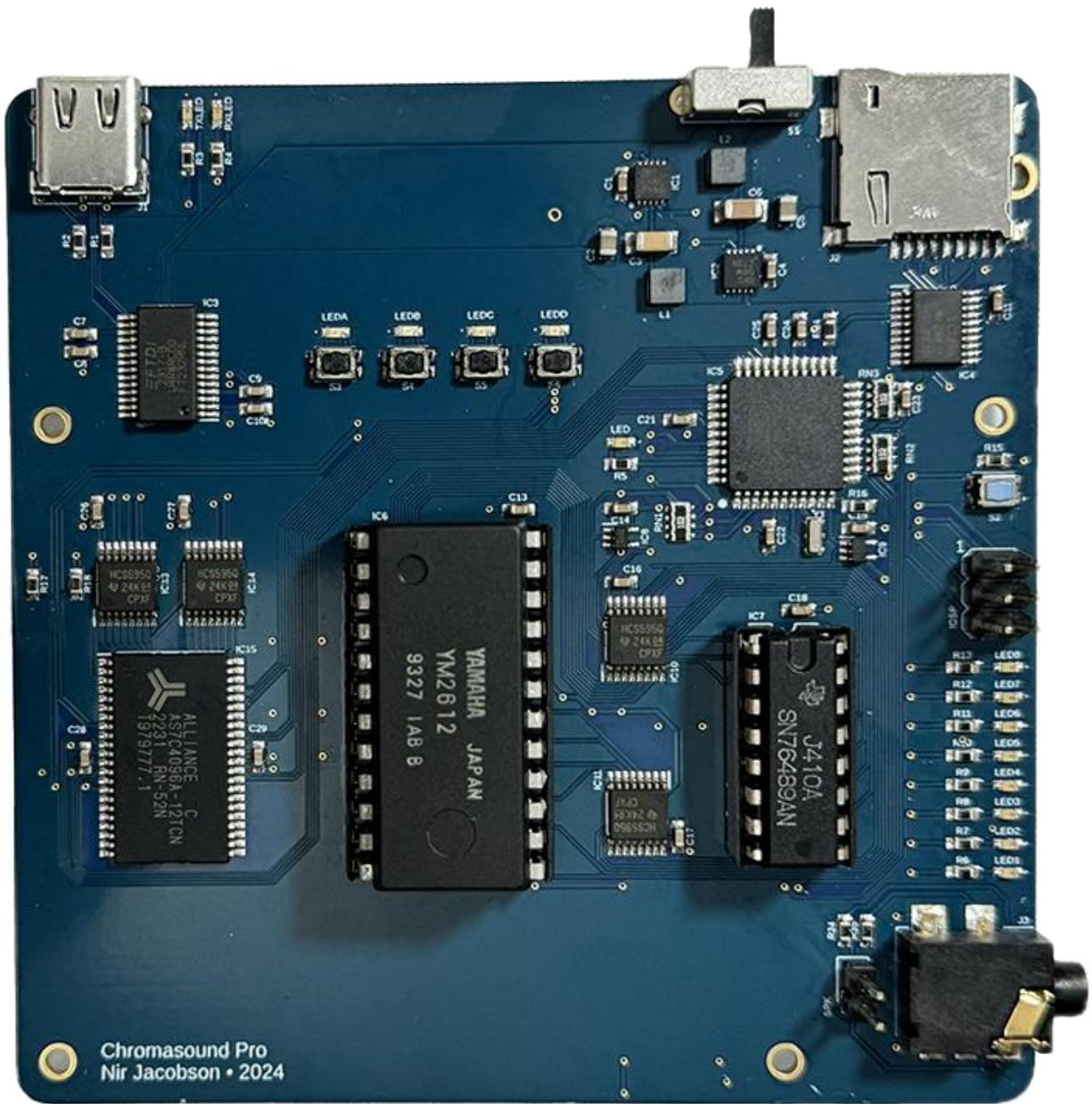
The Software

With the device built we are now almost ready to program the microcontroller. First, we need to discuss one more type of numeric encoding.

Hexadecimal Encoding

Earlier we saw how to convert the number 132 into binary (base 2).

10^2	10^1	10^0
1	3	2

In binary, each digit is a multiple of a power of 2. We say that the number is written in base 2.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	0	1	0	0

When a number is encoded in base n, each digit can only be as high as $n - 1$.

In software, we often want to represent a number in a shorthand format, so we use base 16. Here's the number 132 in base 16:

16^1	16^0
8	4

This is called **hexadecimal notation**. Each digit can be as high as 15. Instead of writing the numbers 10-15 where the digit goes, we use the letters A-F, so there is still one symbol per digit. Here's the number 161 in hexadecimal:

16^1	16^0
A	1

To distinguish numbers in base 10 from binary and hexadecimal numbers, we prepend "0b" to binary numbers and "0x" to hexadecimal numbers. So, 132 can also be written as 0b10000100 and 0x84.

Blinking an LED

The first program we write will just blink the isolated LED. After we write this program and upload it to the music player our environment will be set up to write additional programs.

In order to compile and upload programs a number of tools need to be installed on your system:

- **An AVR toolchain**

  Upon installation you should have the programs **avr-gcc** and **avr-objcopy** on your system.

- **avrdude**

  Upon installation you should have the **avrdude** program on your system.

- **make**

  Upon installation you should have the **make** program on your system.

You can check if you have these tools using the **which** command.

```
$ which avr-gcc
/usr/bin/avr-gcc
```

Switch to a branch **00-blinkled**. Create **main.c**.

```c
#include "global.h"

#include <stdbool.h>
#include <util/delay.h>

#define LED     0xB2

int main() {
    DIR(LED, 1);

    while (true) {
        OUT(LED, 1);
        _delay_ms(500);
        OUT(LED, 0);
        _delay_ms(500);
    }
}
```

This source file includes the other source file, **global.h**, and two standard libraries.

It also defines a constant named LED. Have a look at the schematic for the microcontroller. We can see that an LED is connected to a signal on the microcontroller named PB2. The global library provides functions for manipulating signals by name, written as a hexadecimal number.

main() is the **entry point** to the program. The logic enclosed in the braces is run by the microcontroller after it has been programmed and rebooted.

We start by setting the direction of PB2 as "output" using the value 1.

Then we enter an endless loop. The PB2 signal is set to output 1. Every 500 ms, the signal value is flipped. Since the LED is attached to PB2, this will turn the LED on and off.

Run **make** with no arguments to compile the program.

```
$ make
```

Attach your AVR programmer to your system and music player. Power on the music player.

Run **make upload** to upload the program to the music player.

```
# make upload
```

The Makefile

Converting main.c into a format the microcontroller can understand requires several steps. What actually happens when we run **make**? Let's take a look at the Makefile.

```
MMCU        = atmega644p
DEVICE      = m644p
SERIAL_PORT = /dev/ttyUSB0
PROGRAMMER  = avrisp2

MODULES   = main
OBJECTS   = $(foreach MODULE, ${MODULES}, build/${MODULE}.o)
CFLAGS    = -Wall -O2 -std=c17 --param=min-pagesize=0
LDFLAGS   =
EXEC      = cspro
EXEC_HEX = ${EXEC}.hex

${EXEC_HEX}: ${EXEC}
        avr-objcopy -j .text -j .data -O ihex $< $@

${EXEC}: ${OBJECTS}
        avr-gcc -mmcu=${MMCU} $^ -o $@ ${LDFLAGS}

build/:
        mkdir -p build

build/%.o: build/ src/%.c
        avr-gcc -mmcu=${MMCU} -c $(word 2, $^) -o $@ ${CFLAGS}

upload: ${EXEC_HEX}
        avrdude -P ${SERIAL_PORT} -p ${DEVICE} -c ${PROGRAMMER} -F -e -U flash:w:${EXEC_HEX}

format:
        astyle -rnNCS *.{c,h}

clean:
        rm -rf build
        rm ${EXEC}
        rm ${EXEC_HEX}
```

When no target is specified, make tries to build the first target which above is "cspro.hex" (targets are dark blue). There is a dependency on the file "cspro", so that target (the second one) must be built first. That target in turn depends on the object file of each of the MODULES. The fourth target specifies how to build one object file in the build/ directory from the corresponding source file.

When we run **make**, the following commands are executed in order:

```
mkdir -p build
avr-gcc -mmcu=atmega644p -c src/main.c -o build/main.o -Wall -O2 -std=c17 --param=min-
pagesize=0
avr-gcc -mmcu=atmega644p build/main.o -o cspro
avr-objcopy -j .text -j .data -O ihex cspro cspro.hex
```

Target 1 calls target 2 calls target 4 calls target 3 through the dependency chain, so these targets' commands are run in reverse of this order.

The music player application is organized into different modules. The modules are combined into a program by target 2. As we develop the application, we'll add modules to the MODULES list. When we do that, we'll be telling make to build an object file from the source file with the module's name, and to include that object when producing the program executable.

SPI

We'll need a function to transmit data on the SPI bus using the microcontroller's SPI peripheral. To house this and supporting functions we'll add a SPI module.

All modules except the main module have two files, a **header** and **source**. The header contains the names of functions in the module, and the source contains their implementation. The header can also contain things like enumerations and type definitions that are logically associated with the module.

To use a module, we include its header file. Here is the header for the SPI module.

```c
#ifndef SPI_H
#define SPI_H

#define DDR_SPI    DDRB
#define DD_MOSI    DDB5
#define DD_SCK     DDB7
#define DD_SS      DDB4

#include <avr/io.h>
#include <stdint.h>

enum spi_mode {
    SPI_128,
    SPI_64,
    SPI_32,
    SPI_16,
    SPI_8,
    SPI_4,
    SPI_2
};

typedef enum spi_mode spi_mode;

void spi_init(spi_mode mode);
void spi_set_mode(spi_mode mode);
spi_mode spi_get_mode();
uint8_t spi_transmit(uint8_t c);

#endif // SPI_H
```

The source file contains the implementations of the functions in the header.

```c
#include "spi.h"

spi_mode current_mode;

void spi_init(spi_mode mode) {
    DDR_SPI |= (1 << DD_MOSI) | (1 << DD_SCK) | (1 << DD_SS);
    SPCR = (1 << SPE) | (1 << MSTR);
    spi_set_mode(mode);
}

void spi_set_mode(spi_mode mode) {
    if (mode == SPI_128) {
        SPCR |= (1 << SPR1) | (1 << SPR0);
    } else if (mode == SPI_64) {
        SPCR |= (1 << SPR1);
    } else if (mode == SPI_32) {
        SPCR |= (1 << SPR1);
        SPSR |= (1 << SPI2X);
    } else if (mode == SPI_16) {
        SPCR |= (1 << SPR0);
    } else if (mode == SPI_8) {
        SPCR |= (1 << SPR0);
        SPSR |= (1 << SPI2X);
    } else if (mode == SPI_4) {
        // SPR0 and SPR1 are 0 and SPI2X is not set
    } else if (mode == SPI_2) {
        SPSR |= (1 << SPI2X);
    }

    current_mode = mode;
}

spi_mode spi_get_mode() {
    return current_mode;
}

uint8_t spi_transmit(uint8_t data) {
    SPDR = data;
    while (!(SPSR & (1 << SPIF))) ;

    return SPDR;
}
```

The spi_init() function must be called before other functions in the module can be used.

The function starts by setting the direction of the SPI signals by writing to an internal register. Then it enables the SPI peripheral in Master mode using a different register.

spi_set_mode() sets the frequency of the SPI clock signal by writing to internal registers. The number in the mode name (e.g. 128 in SPI_128) indicates the fraction of the microcontroller's clock source to use as the SPI speed. The clock source is a fixed 20 MHz, so SPI_128 results in a SPI clock speed of 20 MHz / 128 = 156250 Hz.

After spi_init() has been called, spi_transmit() can be used to simultaneously send one byte on MOSI and receive one byte on MISO.

With the SPI interface, we can now blink the eight general-purpose LEDs.

Let's update the main module (main.c).

```c
#include "global.h"

#include <stdbool.h>
#include <util/delay.h>

#include "spi.h"

#define LEDS_CE 0xD2

int main() {
    DIR(LEDS_CE, 1);
    OUT(LEDS_CE, 1);

    spi_init(SPI_2);

    while(true) {
        spi_transmit(0b10101010);
        _delay_ms(500);
        spi_transmit(0b01010101);
        _delay_ms(500);
    }
}
```

We start by setting the LEDs' chip enable signal as output and setting the signal value to 1 to select the LEDs as the current SPI slave.

Then we initialize the SPI peripheral using the fastest speed, 20 MHz / 2 = 10 MHz.

Finally, we enter an endless loop which blinks alternate LEDs in the set of eight.

Add the SPI module to the list of modules in the Makefile, then compile and run the program.

Sound

Now that we can perform SPI transfers let's test out the sound generators.

We introduce a new module for each.

The YM2612

```c
#ifndef YM2612_H
#define YM2612_H

#include "global.h"

#include <avr/cpufunc.h>

#include "spi.h"

#define YM2612_CS       0xD4
#define YM2612_IC       0xD5

#define YM2612_WR       7
#define YM2612_RD       6
#define YM2612_A0       5
#define YM2612_A1       4

#define YM2612_DAC      0x2A

void ym2612_init();
void ym2612_write(unsigned int part, unsigned char address, unsigned char data);
void ym2612_test();

#endif // YM2612_H
```

```c
#include "ym2612.h"

void ym2612_init() {
    DIR(YM2612_CS, 1);
    OUT(YM2612_CS, 1);

    DIR(YM2612_IC, 1);
    OUT(YM2612_IC, 1);

    OUT(YM2612_IC, 0);
    _NOP();
    _NOP();
    _NOP();
    _NOP();
    OUT(YM2612_IC, 1);
    _NOP();
    _NOP();
    _NOP();
    _NOP();
}

void ym2612_write(unsigned int part, unsigned char address, unsigned char data) {
    unsigned char controlByte;

    // Select register
    controlByte = (1 << YM2612_RD);

    if (part == 2)
        controlByte |= (1 << YM2612_A1);

    spi_transmit(controlByte);
    spi_transmit(address);

    OUT(YM2612_CS, 0);
    _NOP();
    _NOP();
    OUT(YM2612_CS, 1);

    // Write register
    controlByte = (1 << YM2612_RD) | (1 << YM2612_A0);

    if (part == 2)
        controlByte |= (1 << YM2612_A1);

    spi_transmit(controlByte);
    spi_transmit(data);

    OUT(YM2612_CS, 0);
    _NOP();
    _NOP();
    OUT(YM2612_CS, 1);
}
```

In ym2612_init(), two of the YM2612's control signals are configured. $\overline{\text{YM\_IC}}$, which is the YM2612's reset signal, is brought low and back high. The NOP instruction tells the microcontroller to do nothing for one clock cycle. Here it ensures $\overline{\text{YM\_IC}}$ is not toggled too quickly for the YM2612 to notice.

ym2612_write() is used to write one data byte to a register within a given part of the chip (1 or 2) and with the given 8-bit address.

Remember that the YM2612 is connected to two shift registers that form a single 16-bit register. The first register is connected to the YM2612's parallel data bus, and the second register is connected to four of the YM2612's control signals. If we send two bytes over SPI, the first one will be shifted into the first register. As the second byte is shifted into that register, the previous byte is shifted out and into the second register. So, we have to send the control signals byte first, then the byte for the data bus. Then we toggle chip enable on the YM2612 when all 12 signals are ready.

The parallel data bus is 8 bits wide. We have to send the address byte first, then send the data byte, for a total of four SPI transfers.

The SN76489

```
#ifndef SN76489_H
#define SN76489_H

#define SN76489_CLK       3579545

#define SN76489_CE        0xD7
#define SN76489_READY     0xD6

#define C4                262
#define E4                330
#define G4                392
#define n(note)           (SN76489_CLK / (32 * note))
#define d(note)           (((n(note) & 0xF) << 8) | ((n(note) >> 4) & 0x3F))

#include "global.h"

#include <avr/cpufunc.h>
#include <util/delay.h>

#include "spi.h"

void sn76489_init();
void sn76489_write(uint8_t data);
void sn76489_test();


#endif // SN76489_H
```

```c
#include "sn76489.h"

void sn76489_init() {
    DIR(SN76489_CE, 1);
    OUT(SN76489_CE, 1);

    DIR(SN76489_READY, 0);

    while (!IN(SN76489_READY)) ;

    // Tone 1 frequency
    sn76489_write(((0x8 | 0) << 4) | 0x0);
    sn76489_write(0x00);

    // Tone 1 attenuation
    sn76489_write(((0x8 | 1) << 4) | 0xF);

    // Tone 2 frequency
    sn76489_write(((0x8 | 2) << 4) | 0x0);
    sn76489_write(0x00);

    // Tone 2 attenuation
    sn76489_write(((0x8 | 3) << 4) | 0xF);

    // Tone 3 frequency
    sn76489_write(((0x8 | 4) << 4) | 0x0);
    sn76489_write(0x00);

    // Tone 3 attenuation
    sn76489_write(((0x8 | 5) << 4) | 0xF);

    // Noise control
    sn76489_write(((0x8 | 6) << 4) | 0x0);

    // Noise attenuation
    sn76489_write(((0x8 | 7) << 4) | 0xF);
}

void sn76489_write(uint8_t data) {
    spi_transmit(data);

    OUT(SN76489_CE, 0);

    _NOP();
    _NOP();
    _NOP();
     NOP();

    while (!IN(SN76489_READY)) ;

    OUT(SN76489_CE, 1);
}
```

sn76489_init() configures the SN76489's two control signals and waits until the chip indicates it's ready.

Remember that the SN76489 has an 8-bit data bus as well that is attached to the first shift register the YM2612 is attached to. To write to the SN76489, we transmit a byte over SPI, bring chip enable low, wait, wait again until the chip is ready, and finally bring chip enable back high.

sn76489_init() writes a couple of registers to silence the chip as the registers contain random values on power-on.

Registers with frequency values require two writes.

Here is main.c updated to perform a sound test.

The YM2612 test will play one note (C4) in a "piano" voice and pause for one second.

The SN76489 test will play three notes (C4, E4, G4) one over the other and one per second.

```c
#include "global.h"

#include <stdbool.h>
#include <util/delay.h>

#include "spi.h"
#include "ym2612.h"
#include "sn76489.h"

int main() {
    spi_init(SPI_2);

    sn76489_init();
    ym2612_init();

    ym2612_test();
    sn76489_test();
}
```

Add the YM2612 and SN76489 modules to the list of modules in the Makefile, then compile and run the program.

USART

We can exchange data between the music player and a PC over the USB cable at a low speed. This will be good for displaying information and receiving commands. Let's introduce a module to provide an interface to the USART (Universal Synchronous Asynchronous Receiver Transceiver) peripheral, which is the serial port we attached to the FTDI FT232RNL earlier to provide the USB connection.

```
#ifndef USART_H
#define USART_H

#include "global.h"

#include <avr/io.h>
#include <stdbool.h>
#include <string.h>
#include <stdint.h>

void usart_init(uint16_t baud);
void usart_send(char c);
char usart_receive(bool echo);
bool usart_try_receive(bool echo, char* c);
void usart_send_dec(long value);
void usart_send_str(char* str);
void usart_send_line(char* str);
void usart_receive_str(char* str, bool echo);

#endif // USART_H
```

The module follows the same pattern of initialization before use. Since there is no clock signal with this peripheral the USART is initialized with a data bitrate (baud rate) instead of a clock scaling factor. Ultimately, however, clock scaling is performed.

```
#include "usart.h"

#define UBRR(baud)     (F_CPU / 16 / baud - 1)

void usart_init(uint16_t baud) {
    UBRR0 = UBRR(baud);

    UCSR0B = (1 << RXEN0) | (1 << TXEN0);
    UCSR0C = (1 << USBS0) | (3 << UCSZ00);
}

void usart_send(char c) {
    while (!(UCSR0A & (1 << UDRE0))) ;

    UDR0 = c;
}

char usart_receive(bool echo) {
    while (!(UCSR0A & (1 << RXC0))) ;

    const char c = UDR0;

    if (echo)
        usart_send(c);

    return c;
}
```

Let's update the main module to test the serial connection.

```c
#define BAUD      57600
#define ECHO      true

#include "global.h"

#include <string.h>
#include <stdio.h>
#include <util/delay.h>

#include "usart.h"

int main() {
    char name[20];
    char message[50];

    usart_init(BAUD);

    usart_send_str("Hello! What is your name? ");

    usart_receive_str(name, ECHO);
    usart_send_line(NULL);

    sprintf(message, "Hello, %s!", name);

    usart_send_line(message);
}
```

First, we set aside some space to store two strings. Then we initialize the USART peripheral.

We prompt the user for their name. As they type their name, we echo back the characters so the user can see them.

After the user hits Enter, a new line is sent followed by a greeting that includes the user's name.

Add the USART module to the list of modules in the Makefile, then compile and upload the program.

Open a connection to the USB serial port using 57600 as the baud rate. On my system, I use the **screen** utility. Replace **ttyUSB0** with the name your system assigns to the USB serial port.

```
# screen /dev/ttyUSB0 57600
```

Then press the reset button on the player to restart the program.

SD Card

The music player plays songs stored on an SD card. A song file is a stream of instructions to write data to the two sound generators. These instructions are separated by instructions to wait which gives musical notes their duration.

The SD card itself does not have a concept of files. It stores consecutive **blocks** of 512 bytes. Later we'll see how a filesystem is implemented on top of these blocks.

Let's add a module for the SD card.

```c
#ifndef SD_H
#define SD_H

#define SD_CS    0xC4

…

#include "global.h"

#include <avr/io.h>
#include <stdbool.h>
#include <limits.h>
#include <stdint.h>

#include "spi.h"

typedef struct {
    uint32_t lba[4];
    uint32_t sectors[4];
} MBR;

typedef struct {
    uint32_t lba;
    uint8_t data[SD_BLOCK_SIZE];
} SD_Block_Cache;

bool sd_init();
uint8_t sd_command(uint8_t command, uint32_t argument, uint8_t crc);
uint8_t sd_command_ocr(uint8_t command, uint32_t argument, uint8_t crc, uint32_t* ocr);
uint8_t sd_transmit(uint8_t command, uint32_t argument, uint8_t crc);
bool sd_read_block(uint32_t address, uint8_t* data);
bool sd_read_long(uint32_t lba, uint8_t la, uint32_t* value);
bool sd_write_block(uint32_t address, const uint8_t* data);
bool sd_read_mbr(MBR* mbr);
bool sd_block_cache_load( SD_Block_Cache* cache, uint32_t lba);


#endif // SD_H
```

The blocks of an SD card can be divided into up to four **partitions**. In this use case, there is only one partition that takes up the entire size of the card. The Master Boot Record (MBR) can be read from an SD card to determine the large block address (LBA), or starting block number, of a partition, and its size in blocks (sectors).

SD_Block_Cache is a struct that can be used to store the contents of a block with a given LBA. sd_block_cache_load() will only perform a block read if the cache is storing a block with a different LBA.

```
#include "sd.h"

bool sd_init() {
    uint8_t result;
    uint32_t ocr;

    DIR(SD_CS, 1);
    OUT(SD_CS, 1);

    for (uint8_t i = 0; i < 10; i++)
        spi_transmit(0xFF);

    while (sd_command(SD_CMD0, SD_ARG_NONE, SD_CRC_CMD0) != SD_IDLE) ;

    result = sd_command_ocr(SD_CMD8, SD_ARG_CMD8, SD_CRC_CMD8, &ocr);

    if (result & SD_ILLEGAL_COMMAND) {
        if ((result = sd_command_ocr(SD_CMD58, SD_ARG_NONE, SD_CRC_CMD58, &ocr)) != SD_IDLE)
            return false;
    }

    if (!(ocr & SD_VOLTAGE_OK))
        return false;

    while (true) {
        if ((result = sd_command(SD_CMD55, SD_ARG_NONE, SD_CRC_NONE)) != SD_IDLE)
            return false;

        if ((result = sd_command(SD_CMD41, SD_ARG_CMD41, SD_CRC_NONE)) == SD_READY)
            break;
    }

    return sd_command(SD_CMD16, SD_BLOCK_SIZE, SD_CRC_NONE) == SD_READY;
}

uint8_t sd_command(uint8_t command, uint32_t argument, uint8_t crc) {
    OUT(SD_CS, 0);
    uint8_t result = sd_transmit(command, argument, crc);
    OUT(SD_CS, 1);
    return result;
}

uint8_t sd_command_ocr(uint8_t command, uint32_t argument, uint8_t crc, uint32_t* ocr) {
    uint32_t ocrResult = 0;

    OUT(SD_CS, 0);
    uint8_t result = sd_transmit(command, argument, crc);
    for (uint8_t i = 0; i < sizeof(ocrResult); i++) {
        ocrResult = (ocrResult << 8) | spi_transmit(0xFF);
    }
    OUT(SD_CS, 1);

    *ocr = ocrResult;
    return result;
}

uint8_t sd_transmit(uint8_t command, uint32_t argument, uint8_t crc) {
    uint8_t result;

    spi_transmit(command | 0x40);
    spi_transmit(argument >> 24);
    spi_transmit(argument >> 16);
    spi_transmit(argument >> 8);
    spi_transmit(argument);
    spi_transmit(crc);

    while ((result = spi_transmit(0xFF)) & (1 << 7)) ;

    return result;
}
```

```
bool sd_read_block(uint32_t address, uint8_t* data) {
    OUT(SD_CS, 0);

    if (sd_transmit(SD_CMD17, address, SD_CRC_NONE) != SD_READY) {
        OUT(SD_CS, 1);
        return false;
    }

    while (spi_transmit(0xFF) != SD_DATA_BLOCK_TOKEN) ;

    for (uint16_t i = 0; i < SD_BLOCK_SIZE; i++) {
        data[i] = spi_transmit(0xFF);
    }

    // CRC
    spi_transmit(0xFF);
    spi_transmit(0xFF);

    OUT(SD_CS, 1);

    return true;
}

bool sd_write_block(uint32_t address, const uint8_t* data) {
    OUT(SD CS, 0);

    if (sd_transmit(SD_CMD24, address, SD_CRC_NONE) != SD_READY) {
        OUT(SD_CS, 1);
        return false;
    }

    spi_transmit(SD_DATA_BLOCK_TOKEN);
    for (uint16_t i = 0; i < SD_BLOCK_SIZE; i++) {
        spi_transmit(data[i]);
    }

    while (!SD DATA ACCEPTED(spi transmit(0xFF))) ;

    while (spi_transmit(0xFF) != 0xFF) ;

    OUT(SD_CS, 1);

    return true;
}

bool sd_read_mbr(MBR* mbr) {
    uint8 t block[SD BLOCK SIZE];
    if (!sd_read_block(0, block))
        return false;

    uint16_t offset = 446 + 8;

    for (uint8_t i = 0; i < 4; i++) {
        mbr->lba[i] = *(uint32_t*)(block + offset + (i * 16));
        mbr->sectors[i] = *(uint32_t*)(block + offset + 4 + (i * 16));
    }

    return true;
}

bool sd_block_cache_load(SD_Block_Cache* cache, uint32_t lba) {
    if (cache->lba != lba) {
        bool retval = sd_read_block(lba, cache->data);
        cache->lba = lba;

        return retval;
    }

    return true;
}
```

Let's update the main module to test reading and writing blocks. This program prompts for a block address and some text. It writes the text to the block at the given address, and then reads it back. Notice that the SD card has to be initialized at a low speed before high speed can be used.

```c
#define   BAUD      57600

#include <stdio.h>
#include <avr/io.h>

#include "spi.h"
#include "sd.h"
#include "usart.h"

int main() {
    uint8_t block[SD_BLOCK_SIZE];

    usart_init(BAUD);

    usart_send_line("Hello world!");

    spi_init(SPI_128);
    if (!sd_init()) {
        usart_send_line("Failed to initialize SD card.");
        return 1;
    }
    spi_init(SPI_2);

    while (true) {
        char address_str[9];
        unsigned long address;
        usart_send_str("Enter a block address: 0x");
        usart_receive_str((char*)address_str, true);
        sscanf(address_str, "%lx", &address);

        usart_send_line(NULL);

        memset(block, 0, SD_BLOCK_SIZE);
        usart_send_str("Enter a block of text: ");
        usart_receive_str((char*)block, true);

        if (!sd_write_block(address, block)) {
            usart_send_line("Failed to write to SD card.");
            return 1;
        }
        memset(block, 0, SD_BLOCK_SIZE);
        if (!sd_read_block(address, block)) {
            usart_send_line("Failed to read from SD card.");
            return 1;
        }

        usart_send_line(NULL);

        usart_send_str("This text was written: ");
        usart_send_str((char*)block);

        usart_send_line(NULL);
        usart_send_line(NULL);
    }

    return 0;
}
```

FAT32 Filesystem

Before files can be copied to an SD card, the card has to be **formatted**. This sets up a partition on the card with a **filesystem** that is typically FAT32.

A file's data can be spread across multiple, nonconsecutive blocks. Some of the blocks in a partition are used to describe which blocks belong to which files and what those files' names are.

I'm not going to go into the FAT32 implementation detail as that is best explained by existing documentation. Instead, I will focus on how the FAT32 module is used.

```c
#ifndef FAT32_H
#define FAT32_H
…

#include <stddef.h>
#include <string.h>
#include <stdio.h>
#include <stdint.h>

#include "sd.h"
…

struct FAT32_FileStream;

typedef void(*fat32_stream_callback)(struct FAT32_FileStream*, void*, size_t, void*);

typedef struct {
    uint32_t lba_fat;
    uint32_t lba_clusters;
    uint8_t sectors_per_cluster;
    uint32_t root_dir_first_cluster;
} FAT32_FS;

typedef struct {
    FAT32_FS* fs;
    char name[12];
    uint8_t attrib;
    uint32_t first_cluster;
    uint32_t size;
} FAT32_File;

typedef struct FAT32_FileStream {
    FAT32_File* file;
    uint32_t cluster;
    uint32_t position;
    uint8_t block_idx;
    fat32_stream_callback callback;

    SD_Block_Cache* block;
} FAT32_FileStream;

bool fat32_init(FAT32_FS* fs, uint32_t partition_lba);
void fat32_root(FAT32_File* file, FAT32_FS* fs);
void fat32_stream(FAT32_FileStream* stream, FAT32_File* file, SD_Block_Cache* block,
                  fat32_stream_callback callback);
bool fat32_stream_next(FAT32_FileStream* stream, void* userData);
void fat32_stream_advance(FAT32_FileStream* stream, size_t items);
void fat32_stream_set_position(FAT32_FileStream* stream, uint32_t position);
bool fat32_file_has_extension(FAT32_File* file, const char* extension);
uint32_t fat32_next_cluster(const FAT32_FS* fs, uint32_t cluster);

uint32_t fat32_cluster_number(const FAT32_File* file, uint32_t cluster_idx);
uint32_t fat32_cluster_lba(const FAT32_FS* fs, uint32_t cluster);


#endif // FAT32_H
```

First we initialize a FAT32_FS struct—which is a handle to the filesystem—using fat32_init().

Then we initialize a FAT32_File struct—which is a handle to a file or folder—using fat32_root(). This gives us a handle to the root directory of the filesystem.

Then we initialize a file stream for the root directory using fat32_stream(). The last argument to this function is a **callback** or **handler**. When fat32_stream_next() is called on a stream, the handler is called and passed the next item in the stream. If the stream is for a directory file, the items in the stream are files inside the directory. If the stream is for an ordinary file, the items in the stream are file data bytes. fat32_stream_next() returns whether there are more items in the stream.

The common pattern is to obtain a stream for the root directory, and in the callback create additional streams from the files obtained from the root stream.

Let's take a look at the main module for an example.

```c
#define  BAUD      57600

#include <stdio.h>
#include <avr/io.h>

#include "spi.h"
#include "sd.h"
#include "fat32.h"
#include "usart.h"

SD_Block_Cache block = { -1 };
MBR mbr;
FAT32_FS fs;
FAT32_File root_dir;
FAT32_FileStream root_stream;

void stream_directory(FAT32_FileStream* stream, void* file, size_t len, void* level) {
    FAT32_File* ffile = (FAT32_File*)file;

    if (ffile->name[0] == '.') {
        fat32_stream_advance(stream, 1);
        return;
    }

    // Print file info
    char format[20];
    char line[40];
    sprintf(format, "[%%c] %%10lu%%%ds%%s", ((int)level + 1) * 4);
    sprintf(line, format, (DIRECTORY(ffile) ? 'D' : 'F'), ffile->size, "", ffile->name);
    usart_send_line(line);

    if (DIRECTORY(ffile)) {
        // Traverse directory
        FAT32_FileStream dir_stream;
        fat32_stream(&dir_stream, ffile, &block, stream_directory);

        while (fat32_stream_next(&dir_stream, (void*)((int)level + 1))) ;
    }

    fat32_stream_advance(stream, 1);
}

int main() {
    usart_init(BAUD);
    usart_send_line("Hello world!");
    usart_send_line(NULL);

    spi_init(SPI_128);
    if (!sd_init()) {
        usart_send_line("Failed to initialize SD card.");
        return 1;
    }
    spi_init(SPI_2);

    if (!sd_read_mbr(&mbr)) {
        usart_send_line("Failed to read MBR.");
        return 1;
    }
    if (!fat32_init(&fs, mbr.lba[0])) {
        usart_send_line("Failed to read FAT32 filesystem.");
        return 1;
    }

    fat32_root(&root_dir, &fs);
    fat32_stream(&root_stream, &root_dir, &block, stream_directory);

    // continued on next page
```

```
    char line[50];
    sprintf(line, "%-4s%-10s    %-s", " T", "Size", "Filename");
    usart_send_line(line);
    usart_send_line("----------------------------------------");
    while (fat32_stream_next(&root_stream, 0)) ;

    return 0;
}
```

This program traverses the root directory depth-first, printing the names of files encountered along the way.

We go through the process of initializing the SD card, filesystem, and root directory stream.

Then we print a table header and divider. Finally, we call next() on the root stream until the return value indicates there are no more items (files).

Each time we call next(), the stream_directory() function gets called with the next file in the stream. This function prints a table row for the file. If the file is a directory, a new stream is created for that directory using stream_directory() as the callback and the new stream is exhausted before the callback returns on the old stream.

The last argument to a stream callback can be used to pass additional information to the callback. In this example, that argument (often called "user data" in some contexts) is used to track the current depth level into the root directory tree. This makes it possible to indent the filenames so that files in a directory appear indented in its listing.

SRAM

We've established that we will be streaming song files from the SD card that contain instructions to perform certain writes to each of the two sound chips. In the stream can be series of very rapid write instructions that all reference contiguous data somewhere else in the file. There are three approaches to handling this that can be taken using the tools we have amassed so far.

1. Use separate file streams on the same file for instructions and data. This makes it possible to read sequentially from two different parts of the same file.

2. Same, but use a different block cache for each stream.

3. Preprocess the song file and inline the data with the instruction.

The first approach is not really viable because the device would have to switch too frequently between the block holding instructions and the block holding data.

The second approach works but requires two caches. The microcontroller only has 4K of internal SRAM, so this takes up one fourth of the memory.

The downside to the third approach (in addition to requiring preprocessing) is that it blows up the size of the file.

A disadvantage of all three options is that to get one byte of data during a series of rapid writes can require reading in all 512 bytes of a block on the SD card first. The rapid writes can happen as fast as 44100 times per second, which allows for $2.23 * 10^{-5}$ seconds per write. Reading in a block takes more than $4.10 * 10^{-4}$ seconds, which is close to 20 times that amount.

We can solve all of these problems with the external SRAM we added. We simply load the contiguous data that will be referenced ahead of time into the SRAM and retrieve data from the SRAM when it is referenced. This approach has the following advantages:

1. We still require two streams, but they are not used in tandem. There is an instruction stream and a temporary stream to load contiguous data into the SRAM.

2. Because the two streams are not used in tandem, the streams can use the same block cache.

3. The SRAM is accessed by individual byte, not by blocks, so we don't get any unwanted delays. The time it takes to read a data byte from the SRAM is dominated by the time it takes to write the address bits to the shift registers, which is about $1.6 * 10^{-6}$ seconds.

```
#ifndef SRAM_H
#define SRAM_H

#include "global.h"

#include <avr/io.h>
#include <avr/cpufunc.h>

#include "spi.h"

#define SRAM_WE          0xB0
#define SRAM_CE          0xB1
#define SRAM_DATA_PORT   0xA
#define SRAM_ADDR_PORT   0xC

void sram_init();
uint8_t sram_read(uint32_t address);
void sram_write(uint32_t address, uint8_t value);

#endif // SRAM_H
```

The header for this module defines constants for the control signals WE and CE.

It also defines the **port** of the data signals (A0-A7) and the port on which the highest three address signals (C7-C5) reside, instead of defining the individual signals.

```c
#include "sram.h"

void sram_init() {
    PORT_DIR(SRAM_ADDR_PORT) |= 0b111 << 5;

    DIR(SRAM_WE, 1);
    OUT(SRAM_WE, 1);

    DIR(SRAM_CE, 1);
    OUT(SRAM_CE, 1);
}

uint8_t sram_read(uint32_t address) {
    PORT_DIR(SRAM_DATA_PORT) = 0x00;

    spi_transmit((address >> 8) & 0xFF);
    spi_transmit(address & 0xFF);
    PORT_OUT(SRAM_ADDR_PORT) = (PORT_OUT(SRAM_ADDR_PORT) & ~(0b111 << 5)) |
                              (((address >> 16) & 0b111) << 5);

    OUT(SRAM_CE, 0);

    _NOP();
    _NOP();

    uint8_t value = PORT_IN(SRAM_DATA_PORT);

    OUT(SRAM_CE, 1);

    return value;
}

void sram_write(uint32_t address, uint8_t value) {
    PORT_DIR(SRAM_DATA_PORT) = 0xFF;
    PORT_OUT(SRAM_DATA_PORT) = value;

    spi_transmit((address >> 8) & 0xFF);
    spi_transmit(address & 0xFF);
    PORT_OUT(SRAM_ADDR_PORT) = (PORT_OUT(SRAM_ADDR_PORT) & ~(0b111 << 5)) |
                              (((address >> 16) & 0b111) << 5);

    OUT(SRAM_WE, 0);
    OUT(SRAM_CE, 0);

    _NOP();
    _NOP();

    OUT(SRAM_CE, 1);
    OUT(SRAM_WE, 1);
}
```

```c
#define BAUD       57600

#define BLOCK_SIZE  256

#include <stdio.h>
#include <stdint.h>
#include <avr/io.h>

#include "usart.h"
#include "sram.h"

uint8_t block[BLOCK_SIZE];

int main() {
    usart_init(BAUD);

    usart_send_line("Hello world!");

    spi_init(SPI_2);

    sram_init();

    while (true) {
        char address_str[9];
        uint32_t address;
        usart_send_str("Enter a byte address: 0x");
        usart_receive_str(address_str, true);
        sscanf(address_str, "%lx", &address);

        usart_send_line(NULL);

        memset(block, 0, BLOCK_SIZE);
        usart_send_str("Enter a block of text: ");
        usart_receive_str((char*)block, true);

        for (uint32_t i = 0; i < BLOCK_SIZE; i++) {
            sram_write(address + i, block[i]);
        }

        memset(block, 0, BLOCK_SIZE);

        for (uint32_t i = 0; i < BLOCK_SIZE; i++) {
            block[i] = sram_read(address + i);
        }

        usart_send_line(NULL);

        usart_send_str("This text was written: ");
        usart_send_str((char*)block);

        usart_send_line(NULL);
    }

    return 0;
}
```

The main module is very similar to the one demonstrating the SD card. This program prompts for a byte address and some text. It writes the text to the SRAM at the given address, and then reads it back.

VGM

The song files on the SD card are in a format called VGM. A VGM file contains a stream of instructions to perform certain writes to each of the two sound chips, or to wait in order to give notes their duration.

We already have the ability to get a stream of contiguous data of a file. It would be great if we could get a stream of the VGM commands those bytes comprise. Instead of the stream callback getting called once per data block, the callback of a VGM stream would get called once per VGM command in the data block. In the callback, we perform the current VGM command.

The VGM module provides a VGM stream construct that wraps a FAT32 file stream.

```c
#ifndef VGM_H
#define VGM_H

#define VGM_HEADER_LOOP_OFFSET       0x1C
#define VGM_HEADER_DATA_OFFSET       0x34

#define VGM_COMMAND_GAME_GEAR_WRITE 0x4F
#define VGM_COMMAND_SN76489_WRITE    0x50
#define VGM_COMMAND_YM2612_WRITE1    0x52
#define VGM_COMMAND_YM2612_WRITE2    0x53
#define VGM_COMMAND_WAITN            0x61
#define VGM_COMMAND_WAIT_735         0x62
#define VGM_COMMAND_WAIT_882         0x63
#define VGM_COMMAND_END_OF_SOUND     0x66
#define VGM_COMMAND_DATA_BLOCK       0x67
#define VGM_COMMAND_WAITN1           0x70
#define VGM_COMMAND_YM2612_WRITED    0x80
#define VGM_COMMAND_SEEK             0xE0

#include "fat32.h"
#include "usart.h"

struct VGM_Stream;

typedef void(*vgm_stream_callback)(struct VGM_Stream*, uint8_t*, uint8_t, void*);

typedef struct VGM_Stream {
    FAT32_FileStream fileStream;
    uint32_t loopOffset;
    uint8_t loopCount;
    uint8_t command[16];
    uint8_t commandLen;
    vgm_stream_callback callback;

    uint8_t* buffer;
    uint16_t buffer_index;
    uint32_t buffer_position;
    size_t buffer_size;
} VGM_Stream;

void vgm_stream(VGM_Stream* vgmStream, FAT32_File* file, SD_Block_Cache* block,
                vgm_stream_callback callback);
bool vgm_stream_next(VGM_Stream* stream, void* userData);
void vgm_stream_next_command(VGM_Stream* stream);

void vgm_stream_file(FAT32_FileStream* fileStream, void* data, size_t len, void* vgmStreamPtr);

size_t vgm_stream_position(VGM_Stream* stream);

void vgm_stream_debug(VGM_Stream* stream);

#endif // VGM_H
```

```c
#include "vgm.h"

void vgm_stream_file(FAT32_FileStream* fileStream, void* data, size_t len, void* vgmStreamPtr)
{
    VGM_Stream* vgmStream = (VGM_Stream*)vgmStreamPtr;
    vgmStream->buffer = (uint8_t*)data;
    vgmStream->buffer_index = 0;
    vgmStream->buffer_size = len;
    vgmStream->buffer_position = fileStream->position;

    fat32_stream_advance(fileStream, len);
}

void vgm_stream(VGM_Stream* vgmStream, FAT32_File* file, SD_Block_Cache* block,
                vgm_stream_callback callback) {
    fat32_stream(&vgmStream->fileStream, file, block, vgm_stream_file);
    vgmStream->loopOffset = 0;
    vgmStream->loopCount = 1;
    vgmStream->callback = callback;
}

bool vgm_stream_next(VGM_Stream* stream, void* userData) {
    if (stream->loopOffset == 0) {
        fat32_stream_next(&stream->fileStream, (void*)stream);

        stream->loopOffset = *(uint32_t*)&stream->buffer[VGM_HEADER_LOOP_OFFSET] +
                                VGM_HEADER_LOOP_OFFSET;

        if (stream->loopOffset == VGM_HEADER_LOOP_OFFSET) {
            stream->loopCount = 0;
        }

        uint32_t dataOffset = *(uint32_t*)&stream->buffer[VGM_HEADER_DATA_OFFSET] +
                                VGM_HEADER_DATA_OFFSET;

        if (dataOffset == VGM_HEADER_DATA_OFFSET) {
            dataOffset = 0x40;
        }

        while ((stream->buffer_position + stream->buffer_size) <= dataOffset) {
            fat32_stream_next(&stream->fileStream, (void*)stream);
        }

        stream->buffer_index = dataOffset - stream->buffer_position;
    }

    if (stream->command[0] == VGM COMMAND DATA BLOCK) {
        uint32_t* dataSize = (uint32_t*)&stream->command[3];

        fat32_stream_set_position(&stream->fileStream,
                                    vgm_stream_position(stream) + *dataSize);
        fat32_stream_next(&stream->fileStream, (void*)stream);
    }

    vgm_stream_next_command(stream);

    if (stream->command[0] == VGM_COMMAND_END_OF_SOUND) {
        if (stream->loopCount > 0) {
            fat32_stream_set_position(&stream->fileStream, stream->loopOffset);
            fat32_stream_next(&stream->fileStream, (void*)stream);
            vgm_stream_next_command(stream);
            stream->loopCount--;
        }
    }

    stream->callback(stream, stream->command, stream->commandLen, userData);

    return !(stream->command[0] == VGM_COMMAND_END_OF_SOUND && stream->loopCount == 0);
}
```

```c
void vgm_stream_next_command(VGM_Stream* stream) {
    if (stream->buffer_index == stream->buffer_size) {
        fat32_stream_next(&stream->fileStream, (void*)stream);
    }

    uint8_t command = stream->buffer[stream->buffer_index];

    size_t commandLength = 1;

    if (command == VGM_COMMAND_YM2612_WRITE1 ||
        command == VGM_COMMAND_YM2612_WRITE2 ||
        command == VGM_COMMAND_WAITN) {
        commandLength = 3;
    } else if (command == VGM_COMMAND_SN76489_WRITE ||
               command == VGM_COMMAND_GAME_GEAR_WRITE) {
        commandLength = 2;
    } else if (command == VGM_COMMAND_SEEK) {
        commandLength = 5;
    } else if (command == VGM_COMMAND_DATA_BLOCK) {
        commandLength = 7;
    }

    if ((command & 0xF0) == VGM_COMMAND_YM2612_WRITED) {
        commandLength = 0;
        for (uint8_t i = 0; i < sizeof(stream->command)
                            && (stream->buffer_index + i) < stream->buffer_size
                            && (stream->buffer[stream->buffer_index + i] & 0xF0) ==
                                VGM_COMMAND_YM2612_WRITED;
            i++) {
            commandLength++;
        }
        memcpy(stream->command, stream->buffer + stream->buffer_index, commandLength);
        stream->buffer_index += commandLength;
    } else {
        if (stream->buffer_index + commandLength > stream->buffer_size) {
            size_t firstSize = stream->buffer_size - stream->buffer_index;
            size_t restSize = commandLength - firstSize;
            memcpy(stream->command, stream->buffer + stream->buffer_index, firstSize);
            fat32_stream_next(&stream->fileStream, (void*)stream);
            memcpy(stream->command + firstSize, stream->buffer, restSize);
            stream->buffer_index = restSize;
        } else {
            memcpy(stream->command, stream->buffer + stream->buffer_index, commandLength);
            stream->buffer_index += commandLength;
        }
    }

    stream->commandLen = commandLength;
}

size_t vgm_stream_position(VGM_Stream* stream) {
    return stream->buffer_position + stream->buffer_index;
}

void vgm_stream_debug(VGM_Stream* stream) {
    usart_send_dec(vgm_stream_position(stream));
    usart_send_str("    ");
    for (size_t i = 0; i < sizeof(stream->command); i++) {
        char str[4];
        sprintf(str, "%02X ", stream->command[i]);
        usart_send_str(str);
    }
    usart_send_line(NULL);
}
```

Calling vgm_stream() creates a VGM stream from a file, and gives the stream an internal FAT32 file stream that uses the vgm_stream_file() callback. vgm_stream() accepts a callback to use for each command in the VGM stream.

In the FAT32 callback, we save the data pointer and some information about it before advancing the file stream. Note that we do not copy any data in this callback.

With a VGM stream created, we can now call vgm_stream_next() which will invoke the callback on the first VGM command in the stream.

The VGM file contains a header before the instruction stream with some information we need. We obtain the **loop offset**, which is the offset into the file where playing resumes after the end of the file is reached, as well as the **data offset**, which is the offset into the file where the instruction stream begins. We obtain this information only when we don't have it, which happens once at the beginning of the file stream.

When vgm_stream_next() is called, the previous command may have been a marker of the start of a contiguous data block whose data will be referenced later in the stream. We'll see how that command got handled a bit later. What's important is that in this function, we skip over that data block in the file stream.

Then we try to get the next VGM command using the stream's internal buffer provided by the FAT32 file stream. Once we have the command, we determine its length in bytes. We copy that many bytes from the internal FAT32 buffer into the VGM stream's VGM command buffer.

If the command is YM2612_WRITED, the command is only one byte long. Typically many of them will appear in a contiguous series; this is the command to write data by reference. We handle this special case by copying as many of the commands in the series as we can into the stream's VGM command buffer.

If the command is not YM2612_WRITED, the command might be several bytes long. It may span the boundary between buffers returned by the FAT32 stream. If it does, we copy bytes up to the end of the buffer, call fat32_stream_next(), then copy the remaining bytes from the beginning of the new buffer into the rest of the VGM command buffer.

Once we have the VGM command buffer, we check if the command indicates we reached the end of the song. If it does, and we haven't used up the number of loops in the song, we jump to the loop offset and try to get the next VGM command.

Once we have the buffer containing the VGM command that we want to act on, we call the VGM stream callback passing it the command buffer.

Like fat32_stream_next(), vgm_stream_next() returns whether the end of the stream has been reached.

PCM

The blocks of referenced data that appear in a VGM file are blocks of PCM sound data. When we receive the YM2612_WRITED command, we write one byte from the data block to the YM2612 and increment the position in the block from which we will write the next time. Because the data is always read in sequence, we can create a PCM stream construct similar to the VGM stream.

```
#ifndef PCM_H
#define PCM_H

#include "fat32.h"
#include "sram.h"

typedef struct {
    FAT32_FileStream fileStream;

    uint32_t write_ptr;
    uint32_t read_ptr;
    uint32_t copy_bytes_remaining;
} PCM_Stream;

void pcm_stream(PCM_Stream* pcmStream, FAT32_File* file, SD_Block_Cache* block);
void pcm_stream_set_data(PCM_Stream* pcmStream, uint32_t offset, uint32_t size);

uint8_t pcm_stream_next(PCM_Stream* pcmStream);
void pcm_stream_seek(PCM_Stream* pcmStream, uint32_t offset);

void pcm_stream_file(FAT32_FileStream* fileStream, void* data, size_t len, void* pcmStreamPtr);

#endif // PCM_H
```

A PCM stream wraps a file stream similar to VGM stream.

When we play a song, we create a VGM stream and a PCM stream on the same file.

When we receive a VGM command marking the start of a block of PCM data, we will know the offset into the song file where the data begins and how many bytes of data there are. We call pcm_stream_set_data() with this information. This copies the data from the file stream into the PCM stream.

When we receive the YM2612_WRITED command, we call pcm_stream_next() to get the data byte that needs to be written from the stream. Unlike the other next() functions we've seen, this function returns the next PCM byte directly and does not use a callback.

To facilitate setting data, the PCM stream has a write pointer. This gets incremented by the size of the data each time data is set, so PCM data is always stored consecutively even if it wasn't in the VGM stream.

To facilitate reading the next byte in the stream, the PCM stream has a read pointer. This gets incremented by one each time pcm_stream_next() is called.

When data is added to the stream, it's written to the external SRAM at the write pointer.

When data is read from the stream, it's read from the external SRAM at the read pointer.

When a seek is performed, the read pointer is updated.

```c
#include "pcm.h"

void pcm_stream_file(FAT32_FileStream* fileStream, void* data, size_t len, void* pcmStreamPtr)
{
    PCM_Stream* pcmStream = (PCM_Stream*)pcmStreamPtr;
    uint8_t* cdata = (uint8_t*)data;
    size_t length = pcmStream->copy_bytes_remaining < len
                      ? pcmStream->copy_bytes_remaining
                      : len;

    for (size_t i = 0; i < length; i++) {
        sram_write(pcmStream->write_ptr++, cdata[i]);
    }

    pcmStream->copy_bytes_remaining -= length;

    fat32_stream_advance(fileStream, length);
}

void pcm_stream(PCM_Stream* pcmStream, FAT32_File* file, SD_Block_Cache* block) {
    fat32_stream(&pcmStream->fileStream, file, block, pcm_stream_file);

    pcmStream->write_ptr = 0;
    pcmStream->read_ptr = 0;
    pcmStream->copy_bytes_remaining = 0;
}

void pcm_stream_set_data(PCM_Stream* pcmStream, uint32_t offset, uint32_t size) {
    pcmStream->copy_bytes_remaining = size;
    fat32_stream_set_position(&pcmStream->fileStream, offset);

    while (pcmStream->copy_bytes_remaining > 0 &&
           fat32_stream_next(&pcmStream->fileStream, pcmStream)) ;
}

uint8_t pcm_stream_next(PCM_Stream* pcmStream) {
    return sram_read(pcmStream->read_ptr++);
}

void pcm_stream_seek(PCM_Stream* pcmStream, uint32_t offset) {
    pcmStream->read_ptr = offset;
}
```

Player

A VGM stream contains two kinds of instructions: instructions to write data to the sound chips and instructions to wait. By instructing the player to wait for a certain amount of time in between writes, musical notes can be held by the sound chips for a specified duration.

We need an object that is capable of taking a FAT32 file, generating a VGM and PCM stream from that file, and processing the commands in the VGM stream (some of which depend on the PCM stream).

For this, we introduce the player module.

```c
#ifndef VGMPLAYER_H
#define VGMPLAYER_H

#include "global.h"

#include <avr/interrupt.h>
#include <util/delay.h>

#include "sd.h"
#include "fat32.h"
#include "vgm.h"
#include "pcm.h"
#include "ym2612.h"
#include "sn76489.h"
#include "usart.h"

struct VGMPlayer;

typedef bool(*vgm_input_callback)(char byte);

typedef struct VGMPlayer {
    VGM_Stream vgm_stream;
    PCM_Stream pcm_stream;
    SD_Block_Cache* block;
} VGMPlayer;

void vgm_player_init(VGMPlayer* player, SD_Block_Cache* block);
void vgm_player_play(VGMPlayer* player, FAT32_File* file, vgm_input_callback input_callback);
void vgm_player_reset(VGMPlayer* player);

#endif // VGMPLAYER_H
```

A vgm_input_callback is a function that responds to a keypress over the USB serial connection. This can be used to pause playback and skip to the next song.

```c
#include "vgmplayer.h"

volatile uint16_t timer = 0;

ISR (TIMER1_COMPA_vect)
{
    if (timer > 0) timer--;
}

void stream_vgm(VGM_Stream* stream, uint8_t* command, uint8_t len, void* player) {
    VGMPlayer* vgmPlayer = (VGMPlayer*)player;

    if (command[0] == VGM_COMMAND_YM2612_WRITE1) {
        while (timer > 0);
        ym2612_write(1, command[1], command[2]);
    } else if (command[0] == VGM_COMMAND_YM2612_WRITE2) {
        while (timer > 0);
        ym2612_write(2, command[1], command[2]);
    } else if (command[0] == VGM_COMMAND_WAITN) {
        timer += *(uint16_t*)&command[1];
    } else if (command[0] == VGM_COMMAND_WAIT_735) {
        timer += 735;
    } else if (command[0] == VGM_COMMAND_WAIT_882) {
        timer += 882;
    } else if ((command[0] & 0xF0) == VGM_COMMAND_WAITN1) {
        timer += (command[0] & 0x0F) + 1;
    } else if ((command[0] & 0xF0) == VGM_COMMAND_YM2612_WRITED) {
        for (uint8_t i = 0; i < len; i++) {
            uint8_t data = pcm_stream_next(&vgmPlayer->pcm_stream);
            while (timer > 0);
            ym2612_write(1, YM2612_DAC, data);
            timer += (command[i] & 0x0F);
        }
    } else if (command[0] == VGM_COMMAND_SN76489_WRITE) {
        while (timer > 0);
        sn76489_write(command[1]);
    } else if (command[0] == VGM_COMMAND_DATA_BLOCK) {
        uint32_t size = *(uint32_t*)&command[3];
        pcm_stream_set_data(&vgmPlayer->pcm_stream, vgm_stream_position(stream), size);
    } else if (command[0] == VGM_COMMAND_SEEK) {
        uint32_t offset = *(uint32_t*)&command[1];
        pcm_stream_seek(&vgmPlayer->pcm_stream, offset);
    } else if (command[0] == VGM_COMMAND_END_OF_SOUND) {
        // end song
    } else if (command[0] == VGM_COMMAND_GAME_GEAR_WRITE) {

    } else {
        vgm_stream_debug(stream);
        _delay_ms(500);
    }
}

void vgm_player_init(VGMPlayer* player, SD_Block_Cache* block) {
    player->block = block;

    // Timer 1
    OCR1A = F_CPU / 44100;                  // 1 sample length
    TIMSK1 = (1 << OCIE1A);                 // Timer interrupt

    TCNT1 = 0;                              // reset timer counter
    TCCR1B = (1 << WGM12) | (1 << CS10); // CTC, no prescaling

    sei();
}
```

```
void vgm_player_play(VGMPlayer* player, FAT32_File* file, vgm_input_callback input_callback) {
    vgm_stream(&player->vgm_stream, file, player->block, stream_vgm);
    pcm_stream(&player->pcm_stream, file, player->block);

    while (vgm_stream_next(&player->vgm_stream, player)) {
        char byte;
        if (usart_try_receive(false, &byte)) {
            if (input_callback(byte)) {
                break;
            }
        }
    }
}

void vgm_player_reset(VGMPlayer* player) {
    ym2612_init();
    sn76489_init();
}
```

To use the player, we start by calling the init() function. This sets up the microcontroller's timer peripheral to trigger an **interrupt** at a given interval. The VGM file measures time in samples with a 44100 Hz sample rate. The timer measures time in clock cycles of the microcontroller. We set the timer to the number of clock cycles in one audio sample. It triggers an **interrupt service routine** (ISR) every time the timer reaches this duration.

There are going to be VGM commands that instruct us to wait a certain number of samples. We create a module-scoped variable named "timer" that we can store this value in. In the ISR, we simply decrement "timer" if it isn't zero. When we need to wait for the timer to reach zero, we use an empty loop that checks its value until it's zero.

In the play() function, we create the VGM and PCM stream. We then go through the VGM commands as quickly as possible, checking for keyboard input on the USB serial connection in between commands.

stream_vgm() is called on every VGM command. Before we execute the command, we have to wait for the timer to reach zero. We can actually enable the player to perform processing in the background while waiting if we make an enhancement. Instead of waiting before *every* VGM command, we only wait for the timer to reach zero ahead of commands that write to the sound chips. This means that if a delay instruction is the last one in the VGM buffer, all of the SD card operations needed to get the next buffer can happen while that delay is taking place.

Let's update the main module to play our songs!

```c
#define  BAUD      57600

#include "global.h"

#include <stdint.h>
#include <avr/pgmspace.h>

#include "usart.h"
#include "spi.h"
#include "sd.h"
#include "fat32.h"
#include "ym2612.h"
#include "sn76489.h"
#include "vgmplayer.h"
#include "sram.h"

SD_Block_Cache block = { -1 };

MBR mbr;
FAT32_FS fs;
FAT32_File root_dir;
FAT32_FileStream dir_stream;

VGMPlayer vgm_player;

bool input_callback(char byte) {
    switch (byte) {
        case 'p':   // pause
            while (usart_receive(false) != 'p') ;
            break;
        case 'n':   // next
            return true;
    }

    return false;
}

void stream_directory(FAT32_FileStream* stream, void* file, size_t len, void* level) {
    FAT32_File* ffile = (FAT32_File*)file;

    if (ffile->name[0] == '.') {
        fat32_stream_advance(stream, 1);
        return;
    }

    // Print file info
    char format[20];
    char line[40];
    sprintf(format, "[%%c] %%10lu%%%ds%%s", ((int)level + 1) * 4);
    sprintf(line, format, (DIRECTORY(ffile) ? 'D' : 'F'), ffile->size, "", ffile->name);
    usart_send_line(line);

    if (DIRECTORY(ffile)) {
        // Traverse directory
        FAT32_FileStream dir_stream;
        fat32_stream(&dir_stream, ffile, &block, stream_directory);

        while (fat32_stream_next(&dir_stream, (void*)((int)level + 1))) ;
    } else if (fat32_file_has_extension(ffile, "VGM")) {
        // Play VGM
        vgm_player_reset(&vgm_player);
        vgm_player_play(&vgm_player, ffile, input_callback);
    }

    fat32_stream_advance(stream, 1);
}
```

```c
bool init() {
    // Serial
    usart_init(BAUD);
    usart_send_line("Hello world!");
    usart_send_line(NULL);

    // SPI Low speed
    spi_init(SPI_128);

    // Storage
    if (!sd_init()) {
        return false;
    }

    // SPI High speed
    spi_init(SPI_2);

    sram_init();

    // Synths
    ym2612_init();
    sn76489_init();

    // Filesystem
    if (!sd_read_mbr(&mbr)) {
        return false;
    }
    if (!fat32_init(&fs, mbr.lba[0])) {
        return false;
    }

    // VGM player
    vgm_player_init(&vgm_player, &block);

    return true;
}

int main() {
    if (!init()) return 1;

    // Print file list header
    usart_send_line(" T  Size          Filename");
    usart_send_line("-------------------------");

    // Walk the file system
    fat32_root(&root_dir, &fs);
    fat32_stream(&dir_stream, &root_dir, &block, stream_directory);
    while (fat32_stream_next(&dir_stream, 0)) ;

    return 0;
}
```

This example is similar to the FAT32 example. We traverse the root directory depth-first. This time, when we encounter a VGM file, we play it through the VGM player.

Buttons

We saw how the VGM player checks for keyboard input on the USB serial connection and passes the input to a callback function which can control the player. We can also check for input on the buttons.

Let's look at the buttons module.

```c
#ifndef BUTTONS_H
#define BUTTONS_H

#include "global.h"

#include <stdbool.h>
#include <stdint.h>

void buttons_init();
bool is_button_pressed(uint8_t button);
void set_button_led(uint8_t button, bool active);

#endif // BUTTONS_H
```

There is a function to check if a button is pressed. There is also a function to set the state of the LED. We saw that the LED is hard-wired to the output signal of the button. How can we control the state of the LED in software?

Recall that the LED is lit when the button signal is low. If we change the direction of the button input on the microcontroller so that it is an output, and then set that output low, the LED will also illuminate.

```c
#include "buttons.h"

void buttons_init() {
    DDRC |= 0x0F;
    PORTC |= 0x0F;
}

bool is_button_pressed(uint8_t button) {
    DIR((0xC0 | button), 0);
    bool result = !IN((0xC0 | button));
    DIR((0xC0 | button), 1);

    return result;
}

void set_button_led(uint8_t button, bool active) {
    DIR((0xC0 | button), 1);
    OUT((0xC0 | button), !active);
}
```

If you look closely, you'll notice that the module defaults the button signal to an output and only treats it as an input briefly when checking if the button is pressed. If the button is pressed, the LED will light because it is hard-wired to. If the button is not pressed, the LED will take on the state it is assigned by the software.

Let's look at an example main.c to see how this works. Without software control, pressing a button will light its LED and releasing it will turn the LED off. In software we can detect a button press, and then "stick" the on-state to the LED even after the button is released. In addition to showing when a button is pressed, the LEDs now show which button was *pressed last*.

```c
#include "global.h"

#include <util/delay.h>

#include "buttons.h"

int main() {
    set_button_led(0, 1);
    _delay_ms(500);
    set_button_led(0, 0);
    set_button_led(1, 1);
     delay ms(500);
    set_button_led(1, 0);
    set_button_led(2, 1);
    _delay_ms(500);
    set_button_led(2, 0);
    set_button_led(3, 1);
    _delay_ms(500);
    set_button_led(3, 0);

    int button = 0;

    while (true) {
        if (is button pressed(0)) {
            set_button_led(button, 0);
            set_button_led((button = 0), 1);
        }
        if (is_button_pressed(1)) {
            set_button_led(button, 0);
            set_button_led((button = 1), 1);
        }
        if (is_button_pressed(2)) {
            set_button_led(button, 0);
            set_button_led((button = 2), 1);
        }
        if (is_button_pressed(3)) {
            set_button_led(button, 0);
            set_button_led((button = 3), 1);
        }
    }
}
```

The program starts by strobing the button LEDs. It then enters a loop where it checks each button for a press. When a button is pressed, the last-pressed-button's LED is turned off, and the pressed button's LED is turned on (in software).

The buttons are all checked continuously and rapidly, and each check is very brief. This means two things:

1. It is very unlikely that a button press will ever be missed between checks.

2. The LEDs show their software-defined state almost all the time.

The effect is that the button signals appear to work in both directions at the same time.

We can now update the VGM player to interpret button presses and pass them to the input callback.

```
void vgm_player_play(VGMPlayer* player, FAT32_File* file, vgm_input_callback input_callback) {
    vgm_stream(&player->vgm_stream, file, player->block, stream_vgm);
    pcm_stream(&player->pcm_stream, file, player->block);

    while (vgm_stream_next(&player->vgm_stream, player)) {
        char byte;
        if (usart_try_receive(false, &byte)) {
            if (input_callback(byte)) {
                break;
            }
        }
        if (is_button_pressed(1)) {
            while (is_button_pressed(1)) ;
            if (input_callback('p')) {
                break;
            }
        }
        if (is_button_pressed(3)) {
            while (is_button_pressed(3)) ;
            if (input_callback('n')) {
                break;
            }
        }
    }
}
```

Recall that the VGM player goes through VGM commands as quickly as possible, only acting on delay commands when it absolutely must. In checking for button presses after each VGM command we are checking frequently enough that it is unlikely a button press will ever be missed. The same is true of keypresses but these are buffered while button presses are not.

The Bootloader

A **bootloader** is a program executed by the microcontroller when it powers on, before the main program. So far, we have not been using one and have been booting directly into the main program. A bootloader can do something that the main program cannot: it can write to the main program space.

Instead of using an ICSP programmer, we can use a bootloader to load a program from some source and then boot into it. The default behavior of the bootloader is to immediately boot into the main program. We can signal the bootloader by holding down a button when we power on the device that we want it to load a new program before booting.

Button A is used to load a new program using the **avrdude** program and the on-board USB port.

Button D will boot into a prompt on the USB port. If you type the name of a program or command and hit enter, the bootloader looks on the SD card for a **bin/** directory and a pre-compiled program with the given name inside. If found, this program is loaded into the program memory and begins to execute.

Hardware Considerations

When used, the bootloader is stored at the end of the main program memory. The microcontroller has **fuse bits** we can set with avrdude that configure the size of the bootloader space and establish that we want to use a bootloader.

The larger the bootloader region is, the less space we have for the main program. The default bootloader size setting is the largest: 8K of the total 64K of program memory. Since the VGM player can fit in 56K I do not change this setting. The only fuse bit that needs to be changed is the lowest bit in the "high fuse byte". Setting this bit causes the microcontroller to boot into the bootloader space on power-on.

There is one more bit we want to set and that is one of the **lock bits**. The fourth lock bit should be set to prevent the bootloader from overwriting itself.

We can add some **make** targets to update the high fuse byte to 0xD8 and the lock byte to 0xEF.

```
MMCU        = atmega644p
DEVICE.     = m644p
SERIAL_PORT = /dev/ttyUSB0
PROGRAMMER  = avrisp2
LFUSE.      = 0xE6
HFUSE       = 0xD8
LOCK        = 0xEF

…

all: build/ ${EXEC_HEX}

upload: ${EXEC_HEX}
        avrdude -P ${SERIAL_PORT} -p ${DEVICE} -c ${PROGRAMMER} -F -e -U flash:w:${EXEC_HEX}

lock:
        avrdude -P ${SERIAL_PORT} -p ${DEVICE} -c ${PROGRAMMER} -U lock:w:${LOCK}:m

hfuse:
        avrdude -P ${SERIAL_PORT} -p ${DEVICE} -c ${PROGRAMMER} -U hfuse:w:${HFUSE}:m
```

The Flash Module

The **flash** module provides functions for reading and writing the microcontroller's program memory. Both programming methods (USB and microSD) use this module to program the chip.

```
#ifndef FLASH_H
#define FLASH_H

#define FLASH_ARGS_PAGE 0xDE00

#define FLASH_PAGE_SIZE SPM_PAGESIZE
#define FLASH_SIZE      65536
#define FLASH_PAGES     (FLASH_SIZE / FLASH_PAGE_SIZE)

#include <stdint.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include <avr/boot.h>

#include "sd.h"
#include "fat32.h"

void flash_erase_chip();
void flash_write_page(uint32_t page, uint8_t* data, size_t len);
void flash_read_page(uint32_t page, uint8_t* data, size_t len);

void flash_program(FAT32_File* file);
void flash_parse_args(char* line, int* argc, char* argv[]);
void flash_args(int argc, char* argv[]);
void flash_read_args(int* argc, char* argv[]);

#endif // FLASH_H
```

The first three functions listed are used to flash the program memory using the USB port. For things to work the way avrdude expects, we need to be able to erase the whole chip and read and write pages that are 512 bytes in size.

The last four functions listed are used to flash the memory using a file on a microSD card. Recall that the microSD boot strategy presents a prompt on the USB port. A program name is given at the prompt along with command arguments. The whole command line is treated as space-separated command arguments. The first argument is the name of the program and is used to locate the file passed to flash_program().

flash_parse_args() will generate argc and argv from a command line. argc is the number of arguments given (including the program name) and argv contains those arguments.

flash_args() writes argc and argv in a simple binary format to the page in program memory that is just before the bootloader.

flash_read_args() generates argc and argv from the arguments page. It is used by the loaded program to retrieve the arguments the program was called with.

The FAT32 Module

The FAT32 module is the same as that from the VGM player program but doesn't include file streams. The reason for removing them from the bootloader is to reduce its size. When we read a file in the bootloader it's from beginning to end, page by page, with no random access, so the stream functionality is not needed.

Instead of file streams are two new functions shown below.

```c
#ifndef FAT32_H
#define FAT32_H

#define FAT32_NONEXISTENT_CLUSTER 0xFFFFFFFF
#define FAT32_DIRENTRY_SIZE       32
#define FAT32_ENTRY_SIZE           4
#define DIRECTORY_FLAG            (1 << 4)
#define DIRECTORY(f)             (f->attrib & DIRECTORY_FLAG)

#include <stddef.h>
#include <string.h>
#include <stdio.h>
#include <stdint.h>
#include "sd.h"

#define FAT32_ENTRY_DELETED(e)      (e[0] == 0x00)
#define FAT32_ENTRY_UNUSED(e)       (e[0] == 0xE5)
#define FAT32_ENTRY_CLUSTER_HIGH(e) (*(uint16_t*)(e + 0x14))
#define FAT32_ENTRY_CLUSTER_LOW(e)  (*(uint16_t*)(e + 0x1A))
#define FAT32_ENTRY_FILE_SIZE(e)    (*(uint32_t*)(e + 0x1C))
#define FAT32_ENTRY_LFN(e)          ((e[11] & 0xF) == 0xF)
#define FAT32_ENTRY_HIDDEN(e)       (e[11] & (1 << 1))
#define FAT32_ENTRY_VOLUMEID(e)     (e[11] & (1 << 3))

typedef struct {
    uint32_t lba_fat;
    uint32_t lba_clusters;
    uint8_t sectors_per_cluster;
    uint32_t root_dir_first_cluster;
} FAT32_FS;

typedef struct {
    FAT32_FS* fs;
    char name[12];
    uint8_t attrib;
    uint32_t first_cluster;
    uint32_t size;
} FAT32_File;

bool fat32_init(FAT32_FS* fs, uint32_t partition_lba);
void fat32_root(FAT32_File* file, FAT32_FS* fs);

…

// New functions

bool fat32_get_file_from_directory(FAT32_File* dir, FAT32_File* file, const char* name);
void fat32_read_block_from_file(FAT32_File* file, uint32_t block, uint8_t* data);

#endif // FAT32_H
```

fat32_get_file_from_directory() looks for a file named *name* in the directory *dir* and initializes *file* as a handle to that file if it is found.

fat32_read_block_from_file() is used to read the 512-byte block in *file* numbered by *block* into the buffer *data*.

USB Programming

When programming the microcontroller, avrdude uses a certain **protocol** to communicate with the chip programmer. It sends and receives messages in order to request to write data to a certain address and to read it back for verification. The protocol defines the message format and when certain messages are sent from each side.

To support programming via the on-board USB port and not an external programmer as we have been, we emulate an external programmer inside the bootloader. The protocol of the programmer we emulate is called STK500 (version 2). The bootloader receives messages from avrdude on the USART and performs the program memory operation requested in the message, then responds with its own message.

The STK500 internals are a bit complex. Let's focus on how the module is used. Here is an excerpt from the bootloader's main.c:

```c
void usb_program_init() {
    stk500v2_init(&stk500v2);
}

void usb_program() {
    usb_program_init();

    while (true) {
        STK500V2_Message message;
        STK500V2_Message answer;

        stk500v2_receive_message(&message);

        if (message.checksum == stk500v2_message_checksum(&message)) {
            stk500v2_answer_message(&stk500v2, &message, &answer);
            stk500v2_send_message(&answer);

            if (answer.body[0] == STK500V2_CMD_LEAVE_PROGMODE_ISP) {
                ((void(*)(void))0)(); // reset
            }
        }
    }

}
```

To program via USB, we listen for STK500 messages on the USART. If the message checksum is what we expect it to be, we perform the operation requested by the message and answer with a message of our own. We do this repeatedly until a message is received requesting to leave programming mode (meaning avrdude is done). When this message is received, we boot to program memory location 0, which is where the program was written to by the STK500.

Booting From the microSD

To boot from the microSD, we obtain a handle to the FAT32 filesystem, then a handle to the **bin/** directory in the root directory. After printing a prompt, we parse the command line we receive to determine the name of the file in the **bin/** directory that will be loaded. We flash this file, along with a binary representation of the command line, to the program memory.

```c
bool sd_boot_init() {
    // SPI Low speed
    spi_init(SPI_128);

    // Storage
    if (!sd_init()) {
        return false;
    }

    // SPI High speed
    spi_init(SPI_2);

    // Filesystem
    if (!sd_read_mbr(&mbr)) {
        return false;
    }
    if (!fat32_init(&fs, mbr.lba[0])) {
        return false;
    }

    fat32_root(&root, &fs);

    return true;
}

bool sd_boot() {
    if (!sd_boot_init()) return 1;

    FAT32_File bindir;
    bool found = fat32_get_file_from_directory(&root, &bindir, "BIN");

    if (found) {
        while (true) {
            usart_send_str("> ");
            char line[128];
            usart_receive_str(line, true);
            usart_send_line(NULL);

            char* argv[8];
            int argc;
            flash_parse_args(line, &argc, argv);

            FAT32_File bin;
            found = fat32_get_file_from_directory(&bindir, &bin, argv[0]);

            if (found && argc > 0) {
                flash_program(&bin);
                flash_args(argc, argv);
                ((void(*)(void))0)(); // reset
            }
        }
    }

    return 0;
}
```

Putting It All Together

The bootloader will defer to one of the two programming strategies presented based on whether button A or button D is pressed. If neither button is pressed the bootloader boots directly into the program last run.

```
#define  BAUD     57600

#include "global.h"

#include <stdint.h>
#include <avr/pgmspace.h>

#include "usart.h"
#include "buttons.h"
#include "spi.h"
#include "sd.h"
#include "fat32.h"
#include "flash.h"
#include "stk500v2.h"

STK500V2 stk500v2;

MBR mbr;
FAT32_FS fs;
FAT32_File root;

void common_init() {
    // Buttons
    buttons_init();

    // USART
    usart_init(BAUD);
}

…

int main() {
    common_init();

    if (is_button_pressed(0)) {
        usb_program();
    } else if (is_button_pressed(3)) {
        sd_boot();
    }

    ((void(*)(void))0)(); // reset
}
```

Obtaining Songs

Possibly the only source of music that utilizes the YM2612 and SN76489 is the catalog of SEGA Genesis/Mega Drive games released for the system. The soundtracks of many of these games can be downloaded from https://project2612.org/.

The soundtracks are in a compressed format called gzip. Once you have extracted an archive of music, you can run the following to decompress the tracks before copying them to your SD card.

```
$ for y in *.vgz; do mv "$y" "${y%.vgz}.vgm.gz"; gunzip *.gz; done
```

Some of my favorite soundtracks:

- Cosmic Carnage (Cyber Brawl)
- Sonic the Hedgehog 3
- Thunder Force IV
- Time Trax

Composing Songs

Shortly after I completed most of the development of the Chromasound I realized I wanted to compose VGMs of my own. It would require a graphical application. The Chromasound is not powerful enough to host one. I could develop a program for the computer, but the Chromasound's USB serial connection does not provide enough bandwidth for streaming song data. The bandwidth is only sufficient for player controls and information display.

What I opted to do was develop a Chromasound HAT for the Raspberry Pi. The HAT has almost all of the same circuits as the standalone. The Pi controls the HAT over the Pi's SPI interface.

To give you a sense of the speed difference of USB serial and SPI, USB serial can operate at a maximum of 115.200 kilobits per second. The SPI connection can operate at 5 megabits per second. SPI is about 43 times faster.

Let's take a look at the HAT, and how it differs from the standalone.
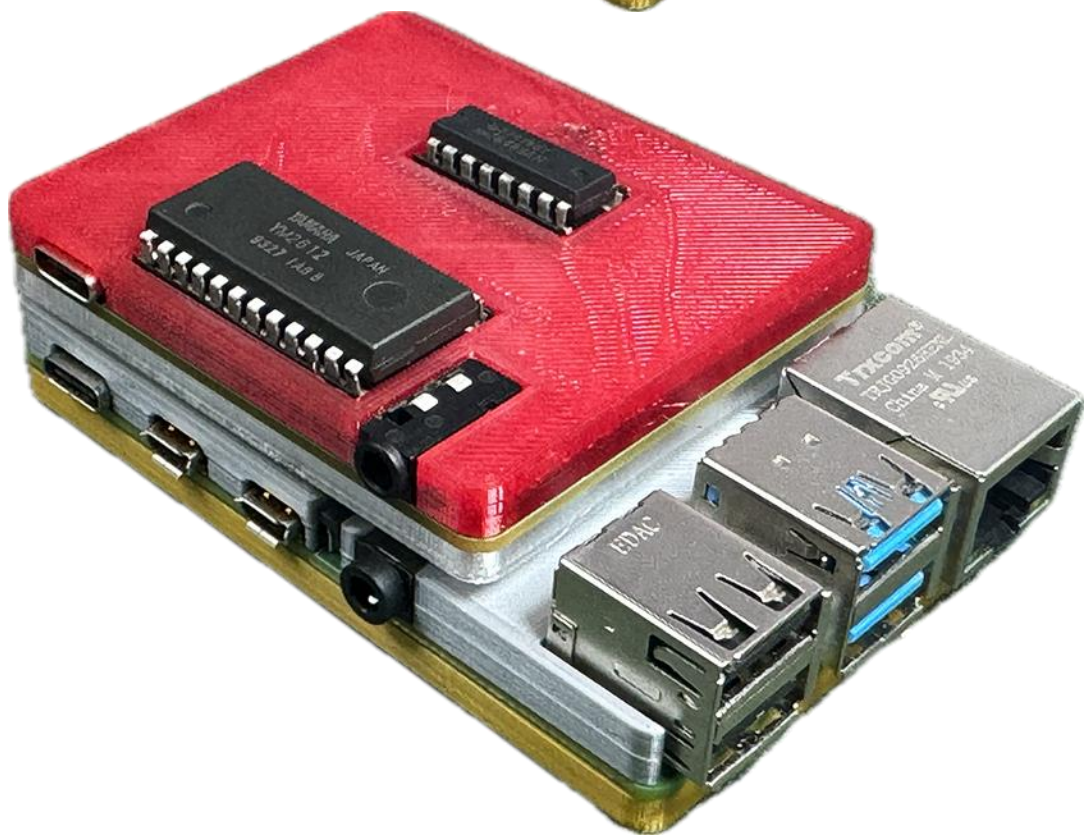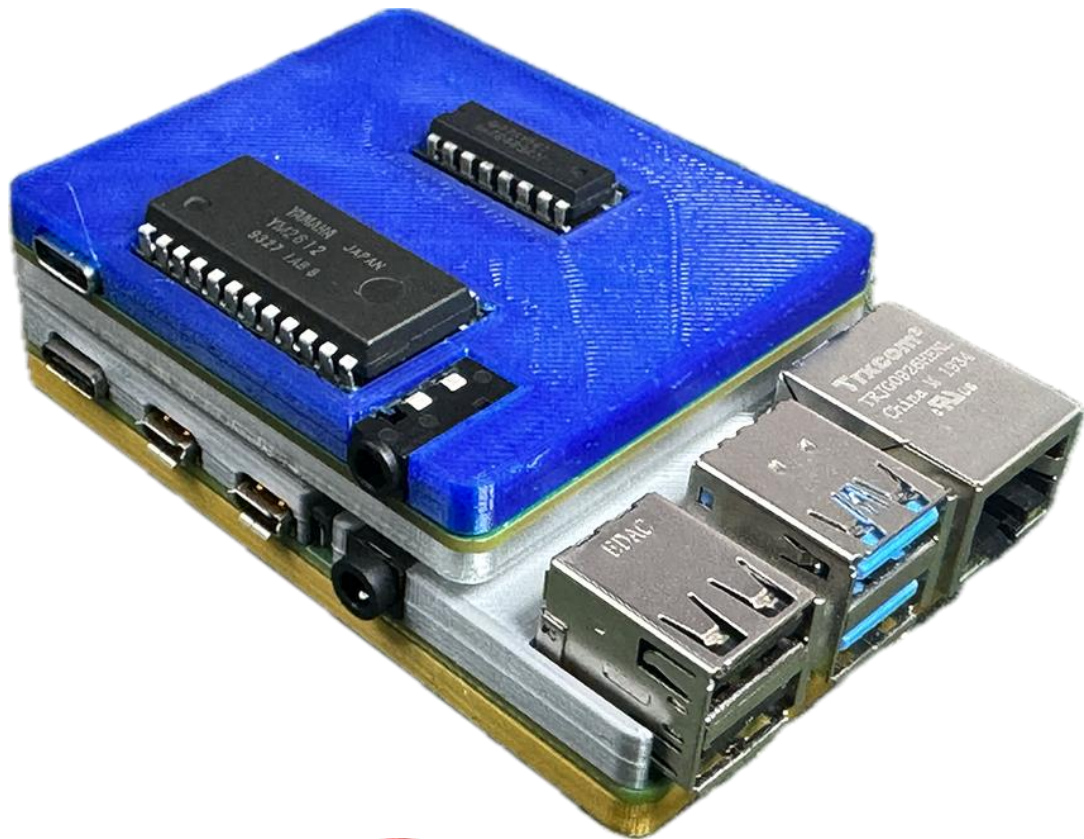
The HAT Hardware

The HAT has most of the same circuits as the standalone. It doesn't have the microSD slot, the buttons, or the LEDs. Also missing is the FTDI chip which provides a USB adapter to the microcontroller's serial port. The serial port is connected directly to the serial port of the Pi.

The HAT has a few extra bits of hardware that the standalone does not have. There is the Pi GPIO connector which provides the SPI and serial connection between the Pi and HAT. Because the Pi is a 3.3V device, and the Chromasound circuits run at 5V, there are two logic level translators on board that convert signals from one level to the other depending on the signal's direction.

The HAT makes use of a second serial port in the AVR microcontroller, which can operate in SPI mode. This SPI peripheral is what controls the onboard-hardware, and its use has been described in chapters prior. The microcontroller's primary SPI peripheral serves as the interface between the Chromasound HAT and the Pi. The HAT therefore runs on two SPI buses. There is the bus to control the on-board hardware, as has been described in chapters prior, and the bus that the Pi uses to send high-bandwidth data to the HAT.

We've seen that SPI connections are not exclusive. Many devices can be put on the same bus. So why use a separate dedicated bus for the interface to the Pi?

The reason is that the microcontroller operates in **SPI master** mode when it controls the on-board hardware. It is the initiator of all SPI transactions. When the microcontroller receives commands from the Pi, it is running as a **SPI slave**. The Pi is the initiator of all SPI transactions. In this way there is a sort of "chain of command" from the Pi to the devices on the Chromasound's SPI bus. It is not possible for a single device (such as the microcontroller) to operate as both master and slave on the same SPI bus.

Finally, the primary SPI interface on the microcontroller (between the Pi and the HAT) is the interface that an external programmer normally would connect to. The Pi can program the microcontroller using avrdude over this connection, so there is no need for an external programmer or data-capable USB port to program the HAT.

You might be wondering why we need a microcontroller at all. Why not just connect the sound generators directly to the Raspberry Pi?

The answer is that the Raspberry Pi runs programs in an **operating system** while the microcontroller does not. One of the best features of an operating system is its facilities for **multiprogramming**, that is, the running of several programs simultaneously. Because many programs are running in the operating system (including those you didn't start directly), time is shared between them. While any one program appears to be running continuously, the truth is no well-behaved program can provide timing guarantees on the nanosecond timescale. It can be interrupted at any time.

The microcontroller is only ever interrupted by hardware interrupts that we prepare for in advance. This means we can make sample-accurate delays between the instructions we send to the sound generators. Sending data to the sound generators cannot be buffered, but sending data to the microcontroller can be. And filling a buffer is generally a safe process to have interrupted for a few microseconds.

The HAT Software

The HAT and standalone share much of the same software logic, sans the routines that drive the hardware only present on the standalone. The **vgmplayer** module is modified to play VGM data contained in an internal buffer instead of a file on and SD card. There is a new **controller** module that operates the interface between the Pi and the HAT. One thing it can do is transfer data from the Pi into the VGM player's internal buffer.

To respond to a command from the Pi at any given time, the HAT makes use of another **interrupt service routine** (ISR). Recall we looked at an ISR that controls a software timer using a hardware interrupt. This second ISR responds when a byte is received on the slave SPI.

Finally, the HAT does not have (or need) a bootloader. The bootloader is used on the standalone to provide additional methods of programming the microcontroller. The Pi can program the microcontroller on the HAT using the SPI connection.

Controller

The **controller** module is responsible for receiving commands and data from the Pi and propagating them to the rest of the system. The slave SPI interface can only operate on one byte in each direction at a time. The controller is implemented as a state machine. When one byte is received from the Pi, the byte value determines how the next byte will be handled. This makes it possible to implement multi-byte messages over the single-byte SPI.

Specifically, the controller maintains a state variable called its **mode** or **command**. The first byte received from the Pi determines the new mode of the controller. The controller knows how many additional bytes it needs to receive in this mode, and how to interpret them. When all those bytes have been received, the controller switches back to "idle" mode. The index of the newly received byte within the current mode/command is tracked by a **step** variable.

```
#ifndef CONTROLLER_H
#define CONTROLLER_H

#define IDLE                0
#define REPORT_SPACE        1
#define RECEIVE_DATA        2
#define REPORT_TIME         3
#define PAUSE_RESUME        4
#define STOP                5
#define RESET               6
#define FILL_WITH_PCM       7
#define STOP_FILL_WITH_PCM 8

#include <stdint.h>
#include <avr/interrupt.h>

#include "vgmplayer.h"

#endif // CONTROLLER_H
```

The header for this module just defines the commands the controller supports. The implementation defines some state variables (mode and step) and module-level working variables in addition to the SPI slave ISR. That's it!

```c
#include "controller.h"

uint8_t mode = IDLE;
uint8_t step = 0;

uint32_t count = 0;
uint16_t space = 0;
uint32_t ptime = 0;

ISR(SPI_STC_vect) {
    if (mode == IDLE) {
        mode = SPDR;
    }

    switch (mode) {
        case REPORT_SPACE:
            if (step == 0) {
                space = vgm_player_buffer_space();
            }
            if (step < 2) {
                SPDR = (space >> (8*step)) & 0xFF;
                step++;
            } else {
                mode = IDLE;
                step = 0;
            }
            break;
        case RECEIVE_DATA:
            if (step == 0) {
                step++;
            } else if (step < 5) {
                ((uint8_t*)&count)[step - 1] = SPDR;
                step++;

                if (step == 5 && count == 0) {
                    mode = IDLE;
                    step = 0;
                }
            } else {
                vgm_player_buffer_write(SPDR);
                SPDR = vgm_player_is_playing_pcm();

                count--;

                if (count == 0) {
                    mode = IDLE;
                    step = 0;
                }
            }
            break;
        case REPORT_TIME:
            if (step == 0) {
                ptime = vgm_player_time();
            }
            if (step < 4) {
                SPDR = (ptime >> (8*step)) & 0xFF;
                step++;
            } else {
                mode = IDLE;
                step = 0;
            }
            break;
        case PAUSE_RESUME:
            vgm_player_pause_resume();
            mode = IDLE;
            step = 0;
            break;
        case STOP:
            vgm_player_stop();
            mode = IDLE;
```

```
                step = 0;
                break;
        case RESET:
                vgm_player_reset();
                mode = IDLE;
                step = 0;
                break;
        case FILL_WITH_PCM:
                vgm_player_fill_with_pcm(true);
                mode = IDLE;
                step = 0;
                break;
        case STOP_FILL_WITH_PCM:
                vgm_player_fill_with_pcm(false);
                mode = IDLE;
                step = 0;
                break;
        default:
                break;
    }
}
```

The first two controller commands are the most important. The first is REPORT_SPACE and it's used to report the minimum number of empty bytes in the VGM player's buffer. After determining how much space the player has left, the Pi sends at most that many bytes of data using the second command, RECEIVE_DATA. On the standalone player, the stream of VGM data comes from a file on the SD card. On the HAT, the stream comes directly from the Pi.

Let's take a look at the alterations to the **vgmplayer** module.

VGM Player

```c
#ifndef VGMPLAYER_H
#define VGMPLAYER_H

#define BUFFER_CAPACITY 512
#define INCREMENT_BUFFER_PTR(x)     x++; if (x == BUFFER_CAPACITY) x = 0;

#define VGM_COMMAND_GAME_GEAR_WRITE 0x4F
#define VGM_COMMAND_SN76489_WRITE   0x50
#define VGM_COMMAND_YM2612_WRITE1   0x52
#define VGM_COMMAND_YM2612_WRITE2   0x53
#define VGM_COMMAND_WAITN           0x61
#define VGM_COMMAND_WAIT_735        0x62
#define VGM_COMMAND_WAIT_882        0x63
#define VGM_COMMAND_END_OF_SOUND    0x66
#define VGM_COMMAND_DATA_BLOCK      0x67
#define VGM_COMMAND_WAITN1          0x70
#define VGM_COMMAND_YM2612_WRITED   0x80
#define VGM_COMMAND_YM2612_WRITEDN  0x96
#define VGM_COMMAND_PCM_SIZE        0xD0
#define VGM_COMMAND_PCM_SEEK        0xE0
#define VGM_COMMAND_PCM_ATTENUATION 0xF0

#include "global.h"

#include <stdint.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#include "ym2612.h"
#include "sn76489.h"
#include "pcm.h"
#include "usart.h"

void vgm_player_init();
uint16_t vgm_player_buffer_space();
uint16_t vgm_player_buffer_size();
uint32_t vgm_player_time();
void vgm_player_buffer_write(uint8_t val);
uint8_t vgm_command_length(uint8_t command);

void process_vgm_command();
void vgm_player_run();
void vgm_player_pause_resume();
void vgm_player_stop();
void vgm_player_reset();
void vgm_player_set_time(uint32_t time);
bool vgm_player_is_playing_pcm();

void vgm_player_fill_with_pcm(bool enable);

#endif // VGMPLAYER_H
```

```c
#include "vgmplayer.h"

volatile uint32_t timer = 0;

volatile uint8_t buffer[BUFFER_CAPACITY];
volatile uint16_t rp = 0;
volatile uint16_t wp = 0;

uint8_t command[10];

uint32_t dataLeft = 0;

volatile uint32_t time = 0;

volatile bool paused = false;
volatile bool stopped = false;

volatile bool playingPCM = false;

volatile bool fillWithPCM = false;

ISR (TIMER1_COMPA_vect)
{
    if (timer > 0) {
        timer--;
        time++;
    }
}

uint32_t vgm_player_time() {
    return time;
}

void vgm_player_init() {
    // Timer 1
    OCR1A = F_CPU / 44100;                 // 1 sample length
    TIMSK1 = (1 << OCIE1A);                // Timer interrupt

    TCNT1 = 0;                             // reset timer counter
    TCCR1B = (1 << WGM12) | (1 << CS10); // CTC, no prescaling

}

uint16_t vgm_player_buffer_space() {
    if (rp <= wp) {
        return BUFFER_CAPACITY - wp + rp - 1;
    } else {
        return rp - wp - 1;
    }
}

uint16_t vgm_player_buffer_size() {
    if (rp <= wp) {
        return wp - rp;
    } else {
        return BUFFER_CAPACITY - rp + wp;
    }
}

void vgm_player_buffer_write(uint8_t val) {
    buffer[wp] = val;
    INCREMENT_BUFFER_PTR(wp);
}

// Continued on next page
```

```
uint8_t vgm_command_length(uint8_t command) {
    switch (command & 0xF0) {
        case VGM_COMMAND_PCM_SIZE:
            if (pcm_is_long_channel(command & 0x0F)) {
                return 5;
            }
            return 3;
        case VGM_COMMAND_PCM_SEEK:
            if (pcm_is_long_channel(command & 0x0F)) {
                return 5;
            }
            return 3;
        case VGM_COMMAND_PCM_ATTENUATION:
            return 2;
        default:
            switch (command) {
                case VGM_COMMAND_SN76489_WRITE:
                case VGM_COMMAND_GAME_GEAR_WRITE:
                    return 2;
                case VGM_COMMAND_YM2612_WRITE1:
                case VGM_COMMAND_YM2612_WRITE2:
                case VGM_COMMAND_WAITN:
                    return 3;
                case VGM_COMMAND_YM2612_WRITEDN:
                    return 5;
                case VGM_COMMAND_DATA_BLOCK:
                    return 7;
                default:
                    return 1;
            }
    }
}

void process_vgm_command() {
    uint8_t channel;
    switch (command[0] & 0xF0) {
        case VGM_COMMAND_WAITN1:
            timer += (command[0] & 0x0F) + 1;
            break;
        case VGM_COMMAND_YM2612_WRITED:
            uint8_t data = pcm_read();
            while (timer > 0);
            ym2612_write(1, YM2612_DAC, data);
            timer += (command[0] & 0x0F);
            break;
        case VGM_COMMAND_PCM_SIZE:
            channel = (command[0] & 0x0F);
            pcm_set_size(channel, *(uint16_t*)&command[1]);
            break;
        case VGM_COMMAND_PCM_SEEK:
            channel = (command[0] & 0x0F);
            if (pcm_is_long_channel(channel)) {
                pcm_seek_ext(channel, *(uint32_t*)&command[1]);
            } else {
                pcm_seek(channel, *(uint16_t*)&command[1]);
            }
            break;
        case VGM_COMMAND_PCM_ATTENUATION:
            channel = (command[0] & 0x0F);
            pcm_set_attenuation(channel, command[1]);
            break;
        default:
            switch (command[0]) {
                case VGM_COMMAND_SN76489_WRITE:
                    while (timer > 0);
                    sn76489_write(command[1]);
                    break;
                case VGM_COMMAND_YM2612_WRITE1:
                    while (timer > 0);
                    ym2612_write(1, command[1], command[2]);
```

```
                    break;
                case VGM_COMMAND_YM2612_WRITE2:
                    while (timer > 0);
                    ym2612_write(2, command[1], command[2]);
                    break;
                case VGM_COMMAND_YM2612_WRITEDN:
                    playingPCM = true;
                    uint32_t samplesToWrite = *(uint32_t*)&command[1];
                    while (samplesToWrite > 0) {
                        while (timer > 0);
                        timer = 1;
                        ym2612_write(1, YM2612_DAC, pcm_read());
                        samplesToWrite--;
                    }
                    playingPCM = false;
                    break;
                case VGM_COMMAND_WAITN:
                    timer += *(uint16_t*)&command[1];
                    break;
                case VGM_COMMAND_WAIT_735:
                    timer += 735;
                    break;
                case VGM_COMMAND_WAIT_882:
                    timer += 882;
                    break;
                case VGM_COMMAND_DATA_BLOCK:
                    dataLeft = *(uint32_t*)&command[3];
                    break;
                case VGM_COMMAND_GAME_GEAR_WRITE:
                    break;
                case VGM_COMMAND_END_OF_SOUND:
                    while (timer > 0);
                    // end song
                    break;
                default:
                    usart_send_str("Unrecognized command 0x");
                    usart_send_hex(command[0]);
                    break;
            }
            break;
    }
}

void vgm_player_run() {
    while (true) {
        if ((rp == wp || stopped) && fillWithPCM) {
            uint8_t data = pcm_read();
            while (timer > 0);
            ym2612_write(1, YM2612_DAC, data);
            timer = 1;
        }
        while (rp != wp && !stopped) {
            if (dataLeft > 0) {
                pcm_write(buffer[rp]);
                INCREMENT_BUFFER_PTR(rp);
                dataLeft--;
            } else {
                while (paused) ;
                if (stopped) break;

                while (rp != wp && (buffer[rp] & 0xF0) == VGM_COMMAND_YM2612_WRITED) {
                    uint8_t data = pcm_read();
                    while (timer > 0);
                    timer += (buffer[rp] & 0x0F);
                    ym2612_write(1, YM2612_DAC, data);
                    INCREMENT_BUFFER_PTR(rp);
                }

                int len = vgm_command_length(buffer[rp]);
```

```
                if (vgm_player_buffer_size() >= len) {
                    uint16_t x = rp;
                    for (int i = 0; i < len; i++) {
                        command[i] = buffer[x++];
                        if (x == BUFFER_CAPACITY) x = 0;
                    }
                    process_vgm_command();
                    rp += len;
                    if (rp >= BUFFER_CAPACITY) rp -= BUFFER_CAPACITY;
                }
            }
        }
    }
}

void vgm_player_pause_resume() {
    paused = !paused;
}

void vgm_player_reset() {
    stopped = false;
    paused = false;
    timer = 0;
    time = 0;
    rp = 0;
    wp = 0;
    playingPCM = false;
}

void vgm_player_stop() {
    vgm_player_reset();
    stopped = true;
}

void vgm_player_set_time(uint32_t t) {
    time = t;
}

bool vgm_player_is_playing_pcm() {
    return playingPCM;
}

void vgm_player_fill_with_pcm(bool enable) {
    fillWithPCM = enable;
}
```

The VGM player's buffer is implemented as a **circular buffer**. A **read pointer** and **write pointer** are separately maintained (variables **rp** and **wp**). When data is added to the buffer, the write pointer is incremented for the next byte that will be written. When data is read from the buffer, the read pointer is incremented for the next byte that will be read. When either pointer falls off the end of the buffer, it wraps back around to the beginning. As the player reads data from the buffer, the space behind the read pointer becomes available, and the Pi can continue sending more data into that space.

When there is one more byte left in the buffer for writing, **wp** will be one less than **rp**. That means when the buffer is full, **wp** and **rp** will be equal. But they are also equal when the buffer is empty! To differentiate the empty and full state, we sacrifice one spot in the buffer, and artificially decide that the buffer is full when **wp = rp - 1**. Now we can determine the used space and free space in the buffer by inspecting the distance between the read pointer and write pointer.

```
uint16_t vgm_player_buffer_space() {
    if (rp <= wp) {
        return BUFFER_CAPACITY - wp + rp - 1;
    } else {
        return rp - wp - 1;
    }
}

uint16_t vgm_player_buffer_size() {
    if (rp <= wp) {
        return wp - rp;
    } else {
        return BUFFER_CAPACITY - rp + wp;
    }
}
```

The VGM player runs continuously and waits whenever it is stopped, or when the buffer is empty. If it receives a data block command, the **dataLeft** variable will be nonzero, and the bytes that follow the command need to be written to the PCM store (the SRAM).

Otherwise, the next byte to read in the buffer contains the ID of a VGM command. We determine how many more bytes comprise the command and do not act until at least that number of bytes is present in the buffer. We then copy the command into the **command** variable which is contiguous and does not wrap around. We then act on the VGM command received and increment the read pointer.

```c
void vgm_player_run() {
    while (true) {
        if ((rp == wp || stopped) && fillWithPCM) {
            uint8_t data = pcm_read();
            while (timer > 0);
            ym2612_write(1, YM2612_DAC, data);
            timer = 1;
        }
        while (rp != wp && !stopped) {
            if (dataLeft > 0) {
                pcm_write(buffer[rp]);
                INCREMENT_BUFFER_PTR(rp);
                dataLeft--;
            } else {
                while (paused) ;
                if (stopped) break;

                while (rp != wp && (buffer[rp] & 0xF0) == VGM_COMMAND_YM2612_WRITED) {
                    uint8_t data = pcm_read();
                    while (timer > 0);
                    timer += (buffer[rp] & 0x0F);
                    ym2612 write(1, YM2612 DAC, data);
                    INCREMENT_BUFFER_PTR(rp);
                }

                int len = vgm_command_length(buffer[rp]);

                if (vgm_player_buffer_size() >= len) {
                    uint16_t x = rp;
                    for (int i = 0; i < len; i++) {
                        command[i] = buffer[x++];
                        if (x == BUFFER_CAPACITY) x = 0;
                    }
                    process_vgm_command();
                    rp += len;
                    if (rp >= BUFFER_CAPACITY) rp -= BUFFER_CAPACITY;
                }
            }
        }
    }
}
```

Between VGM commands, we check if the player has been paused or stopped and wait until each condition is no longer true.

There is one more thing we do in this module that we do not do on the standalone. Take a look at the ISR for the sample timer:

```
ISR (TIMER1_COMPA_vect)
{
    if (timer > 0) {
        timer--;
        time++;
    }
}
```

Now when we decrement the software timer, we also increment a new **time** variable! This variable tracks how many samples have been waited while playing the whole song, which is effectively the current position into the song. In the VGM composition software we will later see, we read this time value from the HAT to determine the current position of the play cursor.

Chromasound Studio

If you've read my other project books, you might know that I named the demonstrative app for my 3D rendering engine "Sahara Studio". I like to name all of my demonstrative apps "Studio", prefixed with the name of the project. With Chromasound Studio it was particularly fitting, because the interface borrows heavily from a program called FL Studio released by Image-Line.
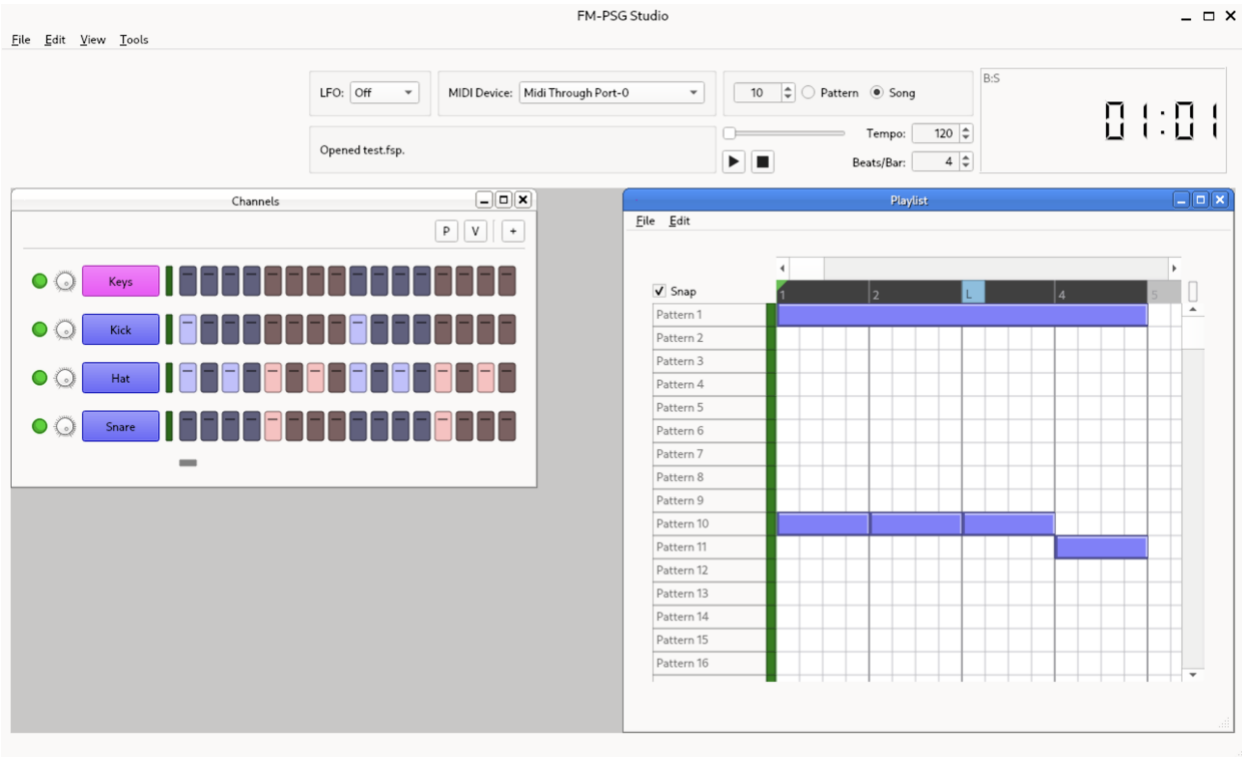
A song is made up of **patterns** in a **playlist**. A pattern can start at any time and overlap with other patterns. A **pattern** is a set of **tracks** for all or some of the project's **channels**. A **track** is a set of notes which can start at any time and overlap with other notes. All the notes in one track are played through a single **channel**.

Above, pattern 10 is the **front pattern** for the project which means its tracks are shown in the "channels" window for editing. This pattern has a track for three of the four channels shown. These tracks use the **step sequencer**.

Pattern 1 has one track that is more complex and was composed using the **piano roll**:

Clicking on the piano roll display brings up the track in the piano roll:
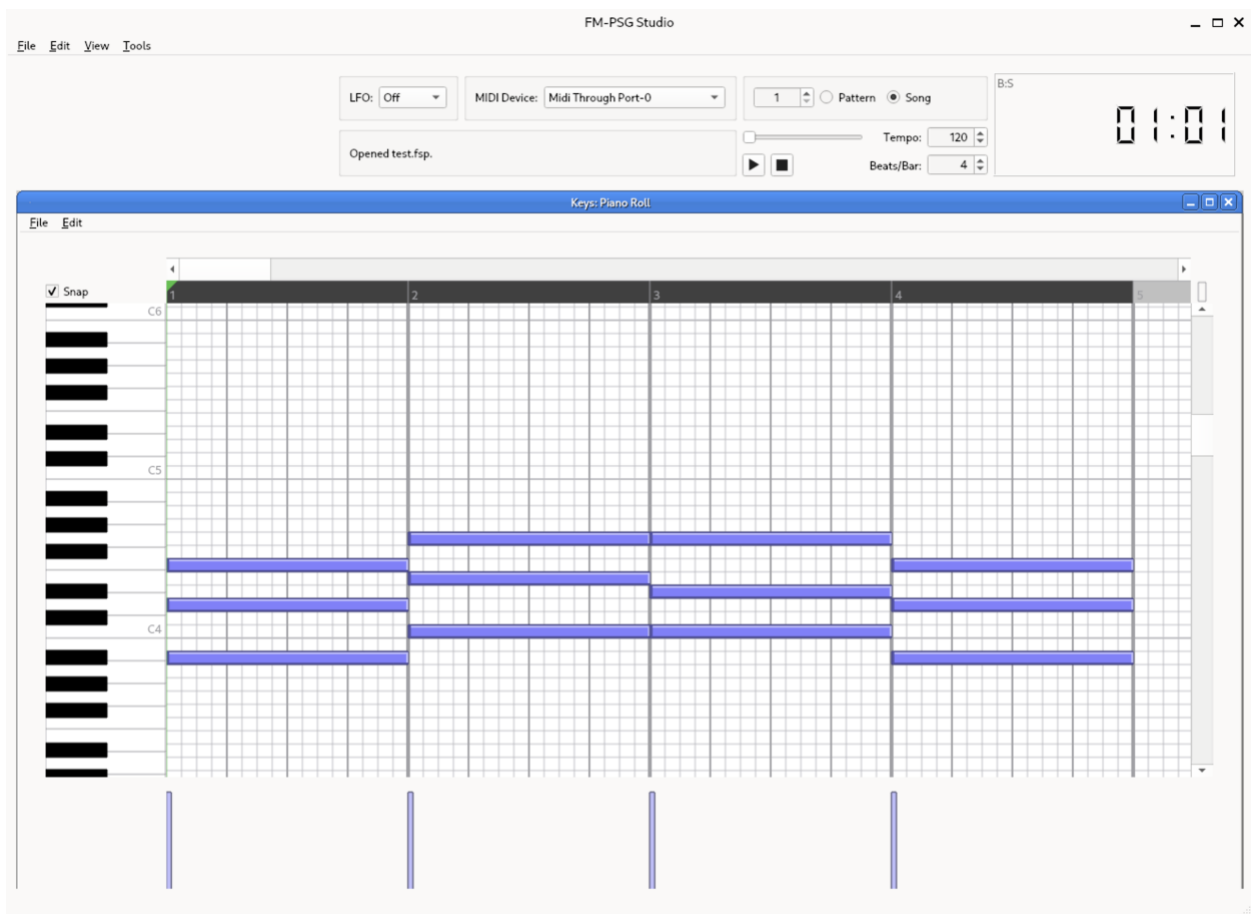
The piano roll is fairly self-explanatory. The bars on the bottom are used to show and set the velocities of the notes. When multiple notes occur at the same time, their velocity bars are sorted so the smaller bars appear on top. However, clicking the bars will only set the velocity for one of the notes. To set the velocity of any note in a chord, shift+right-click the note to expose a menu with a velocity option.

Note how similar in appearance the piano roll is to the playlist window. This is an important part of the architecture of the application that I'll dive into shortly.

Channels

I mentioned that a track is played through a single channel, and as you might have guessed, this is done to assign an instrument to a particular track. What it actually does is assign a track to a particular **source of sound** within the Chromasound as well as settings for the source. The Chromasound has two digital sound generators Yamaha YM2612 and Texas Instruments SN76489. Each provides two sources of sound for a total of four.

Texas Instruments SN76489

- Tone generator

- Noise generator

Yamaha YM2612

- FM synthesizer

- DAC (digital to analog converter)

The tone generator produces a simple tone at a specified frequency. The noise generator produces either white noise or periodic (tonal) noise with a parameter called a "shift rate" that affects the pitch of the noise. There are 3 shift rates to choose from. Once the noise type and shift rate are set, all notes played through the noise generator will sound the same.

The FM synthesizer takes a relatively large (but not too large) number of parameters to be able to reproduce a very wide variety of sounds. Once a sound is configured in the FM synth any note can be played with it.

While the FM synth can reproduce many sounds, it is less capable of reproducing percussion and drum sounds because of how the synth works and the nature of these instruments. To compensate for this the YM2612 has a DAC (digital to analog converter). It can convert digital samples into an analog signal, allowing it to reproduce *recorded* sound. Both sound sources have their strengths and limitations. The FM sounds generally have higher fidelity while the DAC can reproduce sounds that are difficult or impossible to synthesize.

Both chips have **hardware channels** that are independent instances of each sound source. The SN76489 has three tone channels and one noise channel. The YM2612 has six FM channels, the last of which can be swapped for a single DAC channel. All channels can be active simultaneously, but each of the tone and FM channels can only play a single note at a time.

In the screenshot on page 95, there is a polyphonic track assigned to a single channel. This is a **virtual channel** that gets mapped onto three **hardware channels** at runtime. The FM settings are stored in the virtual channel, and they define a particular instrument sound. These settings are copied into each of the physical channels when the song is played.

Virtualizing the hardware channels doesn't only allow for the expression of channel polyphony. It also allows the song writer to incorporate any number of instruments they like. They must only be sure that only six FM notes are ever active at once across all their virtual channels.

There is only one DAC. Does that mean we can only play one audio clip at a time?

PCM Channels

Let's review some of the earlier chapters. Audio clips in a song are stored in a data block that comes at the beginning of the song file, in a sample format called PCM (Pulse Code Modulation). When a song is played, that data block is copied into the Chromasound's SRAM memory unit first. Among the instructions the song file contains are instructions to seek to a particular address in the audio sample data block, and instructions to rapidly write audio samples from the data block—which we store on the SRAM—to the DAC on the YM2612.

What this means is that in the songs we obtain online, the multiple sound sources we may be hearing in the PCM portion have been **mixed** into a single 8-bit stream of samples. While we are limited to one stream of audio in the hardware, we can play multiple audio clips at a time by mixing them first.

Does that mean we need to preprocess the whole song to generate one long stream of mixed audio samples? Ordinarily, yes. However, if we're willing to diverge from the VGM specification a bit, there is something we can do to avoid this.

The PCM seek command starts with the byte 0xE0 which is followed by an audio memory address. It resets the PCM read pointer to a specific value. We can extend this command to support multiple read pointers. I use the lower four bits to specify a PCM channel number.

When pcm_read() is called on the Chromasound, the audio sample memory is read at the location of each read pointer, and the read samples are mixed. We retain the original functionality by only using the 0xE0 command. We make use of mixing by additionally using 0xE1, 0xE2, etc. I also introduced the 0xF0 (0xF1, 0xF2, etc.) command to set the attenuation (volume) of each audio stream.

Normally we would use the SRAM to store pre-mixed PCM clips that play exclusively at different times throughout the song. Now we can store each individual audio sample (e.g. kick, hat, snare, etc.) once in the SRAM, and trigger them over each other and at multiple times throughout the song, using the extended seek and attenuation commands.

What we now have is **firmware virtualization** of the DAC into multiple (four) PCM channels. In Chromasound Studio, we create virtual channels on top of these firmware-virtualized channels. We can have any number of audio samples or clips in our song, but only four can be playing at a time. This gives us audio track and drum machine capability!

Putting It All Together

On the hardware side, we have a VGM player that reads from a buffer that we have relatively high-speed access to on the Pi. On the software side, we have a program with which we can compose a song. When the "play" button is pressed in Chromasound Studio, the song is **compiled** into a VGM stream that the Chromasound can play directly. This process includes the mapping of the virtual channels onto physical channels provided by the hardware and firmware.

Chromasound Studio writes the VGM stream to the VGM player's buffer. In between sending chunks of this stream, Chromasound Studio occasionally reads back the "time" variable from the HAT. This is done at fairly long intervals to avoid using too much of the microcontroller's resources. In between these time readings Chromasound Studio runs a software timer, which "fills in" the time value so that a mostly accurate value is available whenever it is requested.

While the song is playing, Chromasound Studio redraws the window at about 30 frames per second. How each widget is rendered on the screen in each frame is a function of the value of the "time" variable. The playlist window draws a green cursor at the position of the player. The channels window and piano roll do the same when the corresponding pattern is playing. These windows also turn LEDs on and off in time with the activity of virtual channels and patterns in the song.

One More Extension

I mentioned that an extension to the VGM specification allows for more optimization and control over the DAC. There is one more extension I added in this vein.

There is just one VGM command for writing audio samples from the PCM block to the YM2612 DAC. It is 0x8n, and it specifies the player should write one audio sample and wait n samples. This means that while PCM audio is playing, every sample is the result of a 0x8n command, which results in very long sequences of 0x8n commands in the stream. One possible justification for this is that it synchronizes the player during PCM playback. However, for the Chromasound, it means there is significantly less time and bandwidth to perform the operation of reading samples from memory and writing them to the YM2612.

To solve this problem, I introduced a new VGM command 0x96 that is followed by a 32-bit number of samples, say s. This command is the equivalent of s consecutive 0x81 commands. The overhead of sending and receiving those 0x81 commands is removed.

Data Storage

Chromasound Studio saves and opens projects with a **.csp** extension. This is a file in BSON format, which itself is a binary form of JSON (JavaScript Object Notation). The piano roll and playlist window support copy and paste, and these operations put BSON data onto the operating system's clipboard. The FM channel window can save and open FM settings in a **.opn** patch file that is also BSON. I think BSON is great because it's binary but becomes human readable with the right inspection tool.

Chromasound Studio can render the project to a VGM file. The same VGM compilation code that is used to play the song is used to produce the VGM file. It can either do it using the extended VGM format I described or generate a bigger file that is compatible with all other players.

The piano roll can save and import MIDI files.

The PCM channel settings specify the path of an audio clip to be played. This file is in raw, unsigned byte format for which I use the **.pcm** extension. I use **ffmpeg** to convert audio files into this format:

```
$ ffmpeg -i input -f u8 -c:a pcm u8 -ar 44100 output.pcm
```

More About Piano Roll & Playlist

Earlier I pointed out the similarity in appearance of the playlist and piano roll. Both windows show items that occur at a certain time and end at a certain time, and which can overlap each other. This format can be used to describe events with duration, as well as generic sequences that can overlap and repeat. The chart of rectangles shown in each window is referred to as a **Gantt chart**.

I was essentially able to write the logic for these two windows once by implementing a Gantt chart. Both the Track and Playlist constructs in Chromasound Studio contain Item objects which implement a Gantt item interface. This means they can be supplied directly to a Gantt chart for graphical rendering.

A feature that is available in both the piano roll and playlist is the selection of part of a track or song to be played in a loop. This is an editing feature for reviewing part of the song. To use it, click and drag the mouse over the Gantt chart's header.

A feature that is only available in the playlist is the ability to set a loop point in the song. When the song reaches the end, it continues from the loop point if it is defined. Unlike selection looping, this loop is incorporated into the song. To set the loop point, right-click the Gantt chart's header at the loop point. Right-click the loop marker to remove it.

There is one more thing Gantt charts support and that is **markers**. We'll get to markers in a bit.

Understanding FM Synthesis

When you configure an FM channel in Chromasound Studio, this is the screen you are presented with:
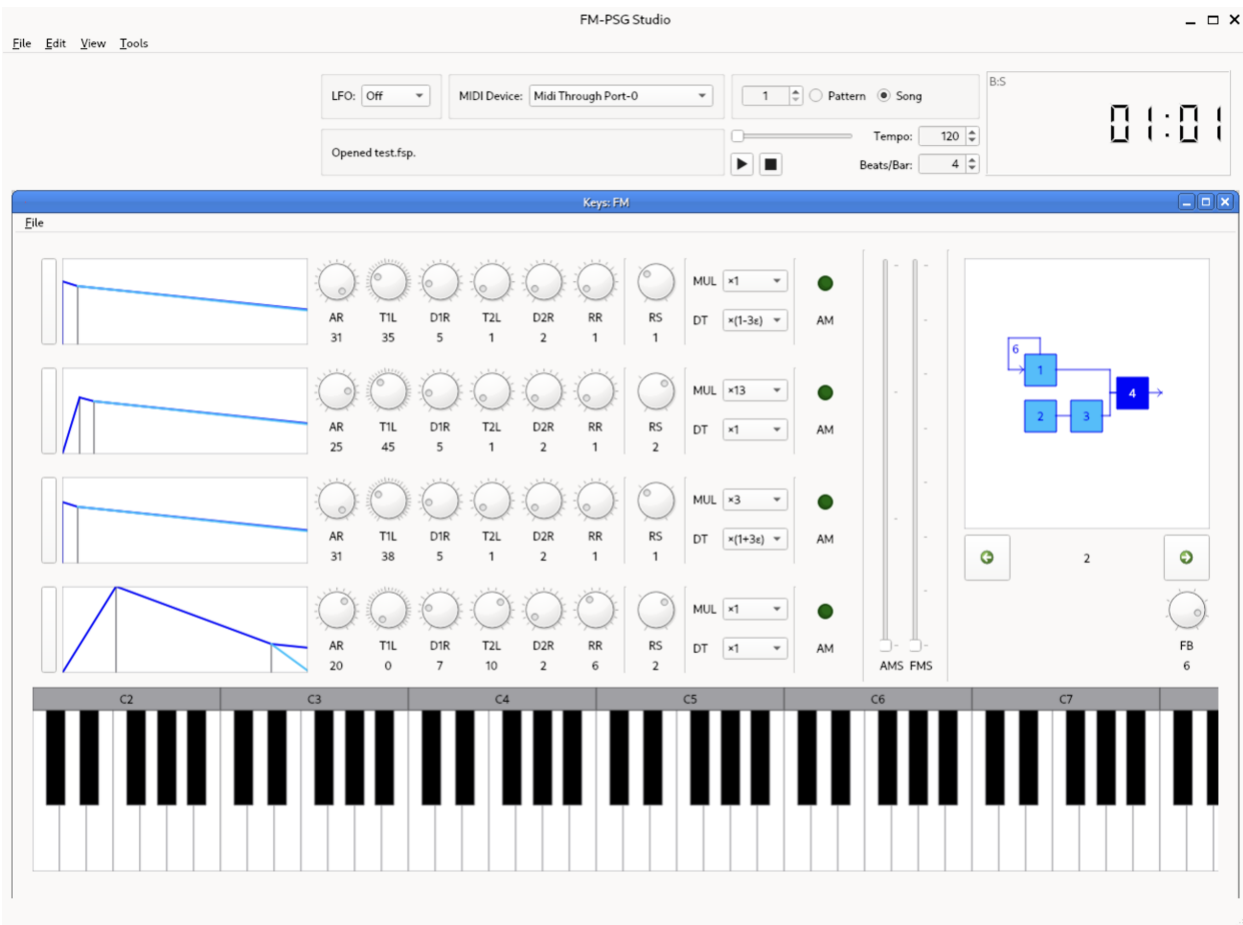
The number of parameters in the FM settings is many, but they are grouped so it becomes easier to understand. Let's start by talking about the parameters of one **operator**.
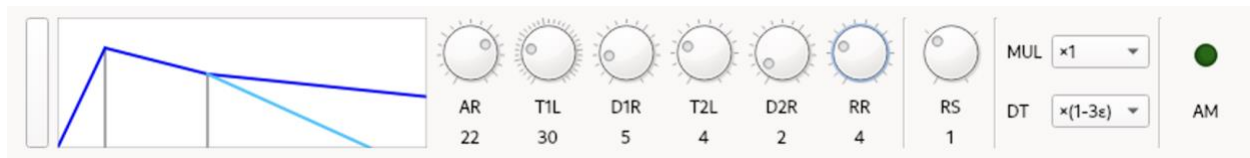
An operator produces a simple tone—a sine wave—and applies an **envelope** to it. An **envelope** describes a change in volume over time. This is what is shown in the display on the left. The shape of the envelope is controlled by the six dials shown to the right of the graph.

AR   Attack rate. This controls the angle of the leftmost slope in the graph.

T1L   The height of the highest point of the graph. As this parameter changes, the envelope keeps its shape but scales up or down in size. This parameter actually controls inverse volume (attenuation) so lower values produce a higher volume. T1 refers to the time of this inflection point in the graph, and L stands for level.

D1R   Decay 1 rate. This controls the angle of the second slope in the graph.

T2L   The height of the graph at the inflection point between the second and third (darker blue) slopes in the graph. This parameter is also in terms of attenuation, and its range is mapped to the range [0, **T1L**]. T2 refers to the time of this inflection point in the graph, and L stands for level.

D2R   Decay 2 rate. This controls the angle of the third (darker blue) slope in the graph.

RR   Release rate. When the key that triggered the envelope is released, the volume continues to decay but at the rate **RR** (light blue).

RS stands for rate scaling. It controls the degree to which the envelope becomes narrower as the frequency of the operator tone becomes higher.

Both **MUL** and **DT** relate the operator's output frequency to the tone frequency.

MUL multiplies the tone frequency, e.g. ×1/2, ×1, ×2, ... ×15.

DT gives small variations from the tone frequency × MUL.

DT values*
×(1-3ε)
×(1-2ε)
×(1-ε)
×1
×(1+ε)
×(1+2ε)
×(1+3ε)

* ε is a small number.

AM enables or disables amplitude modulation for the operator (more on this later).

The Algorithm

The algorithm of the channel defines how the operators are combined. There are eight algorithms to choose from. **FB** controls the amount of self-feedback in operator 1.
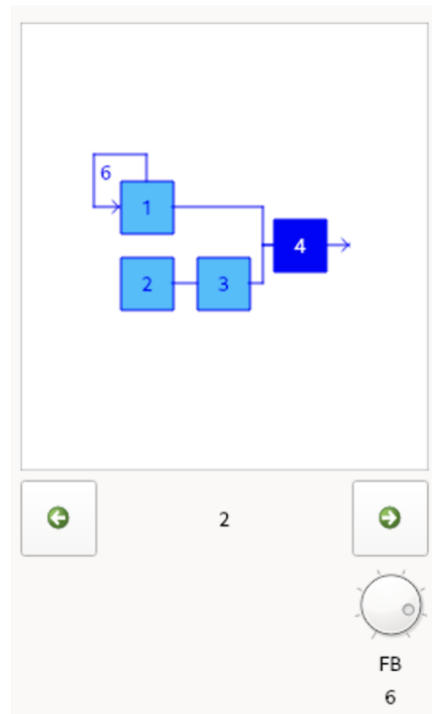
A dark blue operator is called a **slot**, and its output is mixed into the output of the channel. The output of a light blue operator is fed into the input of another operator.

An operator's output can normally be defined as:

$$F(t) = A(t)\sin(\omega t)$$

Where $A(t)$ is the envelope and ω defines the sine wave frequency.

When operator 2 feeds into operator 3, operator 3's output becomes:

$$F_3(t) = A_3(t)\sin(F_2(t) + \omega_3 t)$$
$$F_3(t) = A_3(t)\sin(A_2(t)\sin(\omega_2 t) + \omega_3 t)$$

This produces an oscillation of operator 3's frequency.

We say that operator 2 is **modulating the frequency** of operator 3.

Let's say A(t) = 1 for all t, and that ω = 1, for both operators 2 and 3.
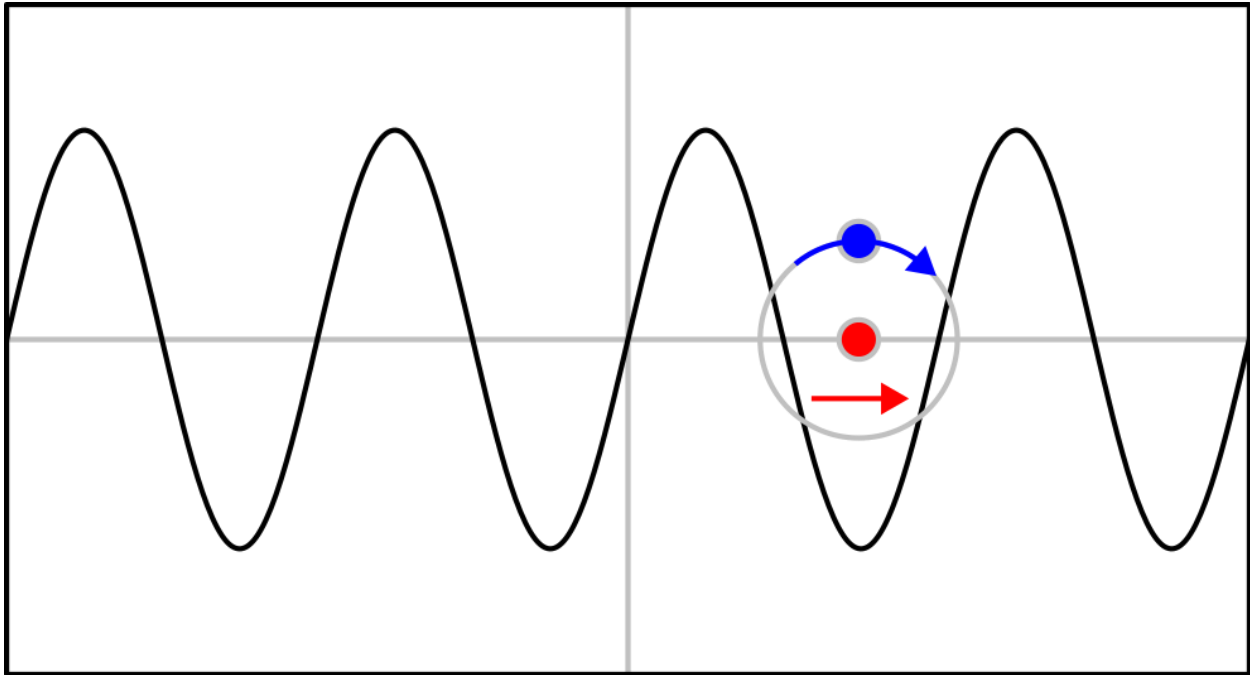
F$_2$(t) = sin(t)

The blue dot goes around the red dot as the red dot travels to the right. Both dots travel at constant speed. If at every position of the red dot, we feed the horizontal position of the blue dot into the sine function, we get a graph like the one below.

F$_3$(t) = sin(F$_2$(t) + t)
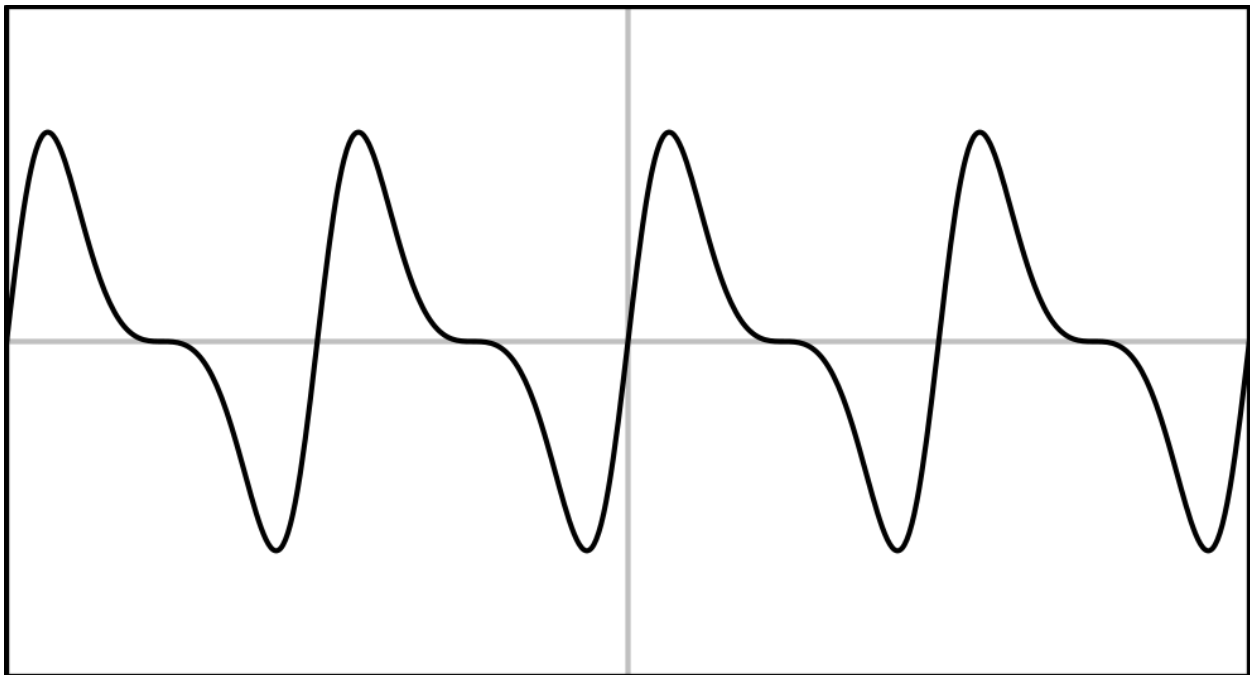
= sin(sin(t) + t)

The LFO

The Yamaha YM2612 has an LFO (low frequency oscillator) which can apply a sine wave of a selectable low frequency to all or some of the FM channels. The sine wave can be used to modulate both the amplitude and base frequency of an FM channel. This can be used to apply vibrato to a note and effect other undulations.

The last two controls in the FM settings are AMS and FMS. They control the amplitude modulation sensitivity and frequency modulation sensitivity of the channel to the output of the LFO.

The operator-level AM control enables or disables LFO amplitude modulation for a specific operator. Modulating the amplitude of the slots will modulate the amplitude of the channel output. Modulating the amplitude of the other operators will change the channel's flavor.

Chromasound Studio lets you change the LFO frequency and channels' LFO sensitivity throughout the course of the song. Let's take a look.

The LFO will start at the frequency selected in the OPN globals dialog. To change the LFO frequency at some point in the song, shift-click the header in the playlist to add an LFO marker. Shift-click the marker to change the LFO setting or remove the marker.
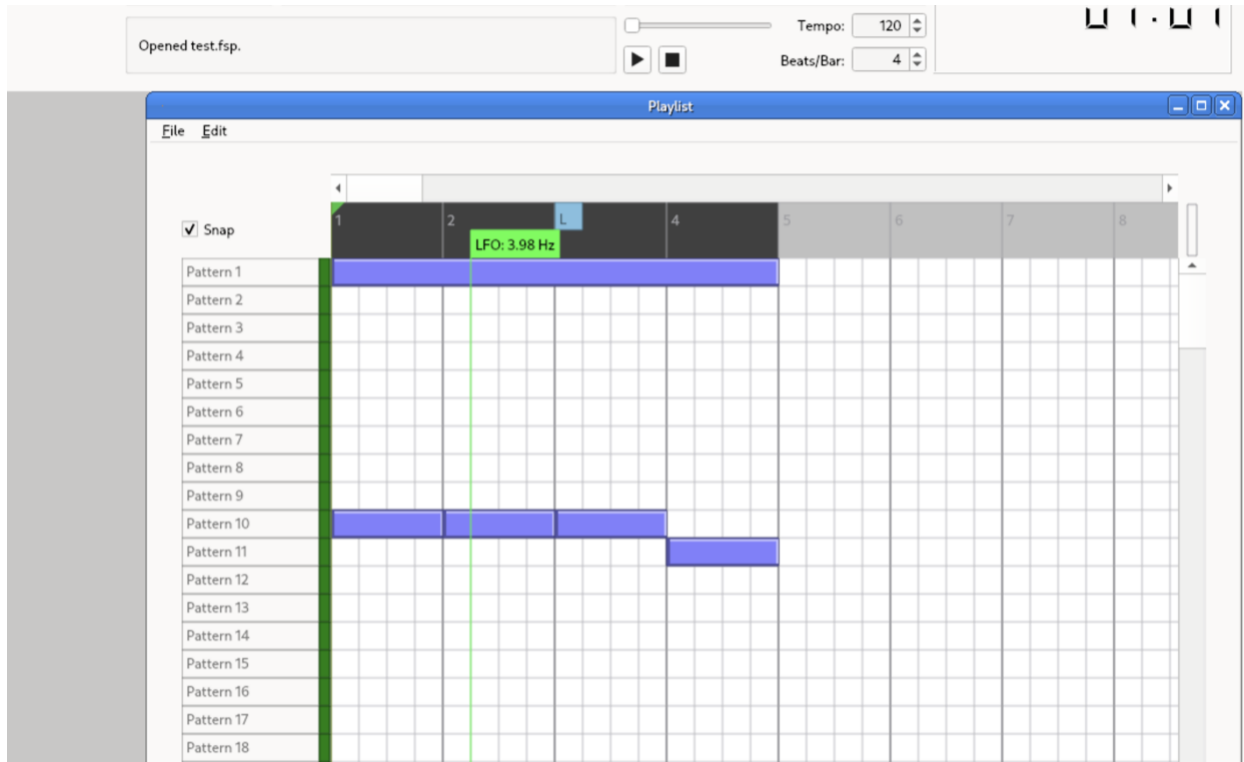
The LFO sensitivity of an FM channel can be modified in the middle of a track. To change the LFO sensitivity of an FM channel, shift-click the header in the piano roll to add a settings change marker. Shift-click the marker to change the LFO sensitivity (or any other FM setting!) or remove the marker.
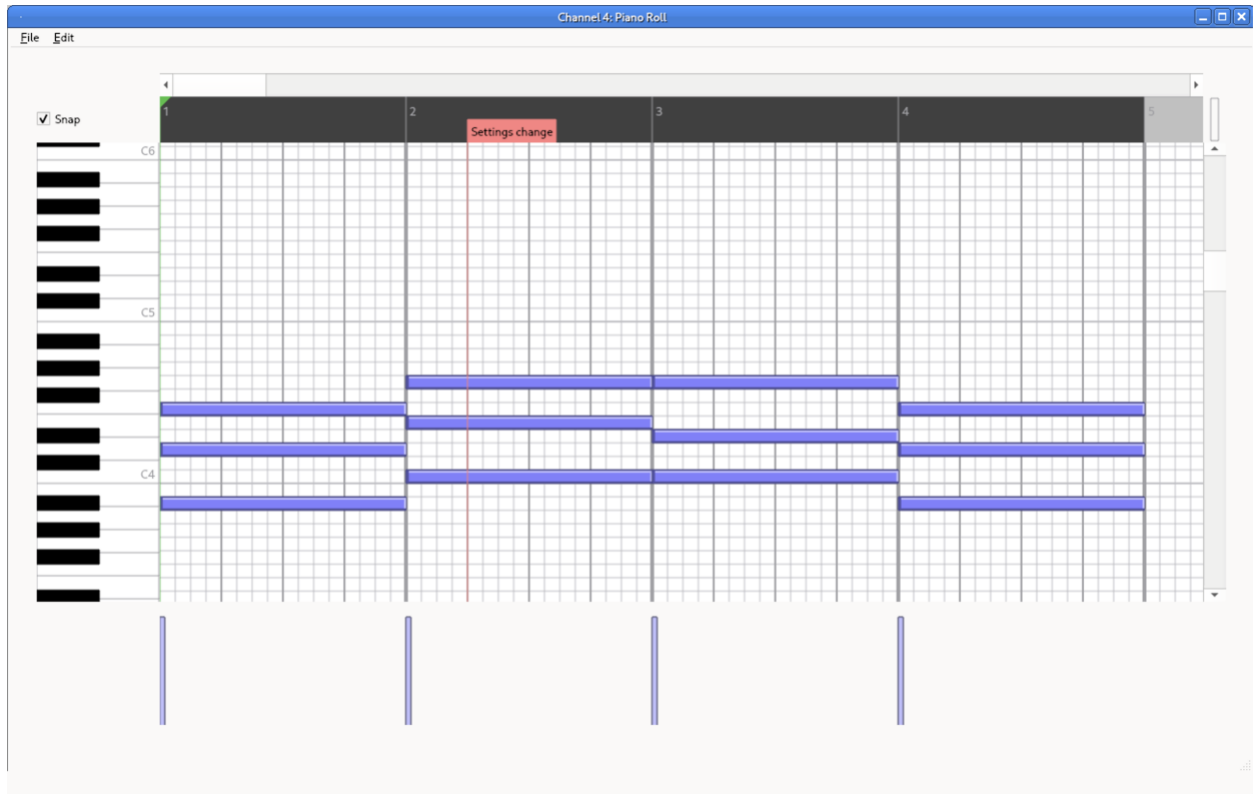
As a rule of thumb, I suggest favoring multiple channels with different settings over track settings changes whenever possible. The markers are mostly intended for use with the channel volume and LFO.

Obtaining FM Patches

It can be useful to have a starting point when authoring new patches. In the Tools menu, Chromasound Studio provides a utility for importing FM patches from VGM files.
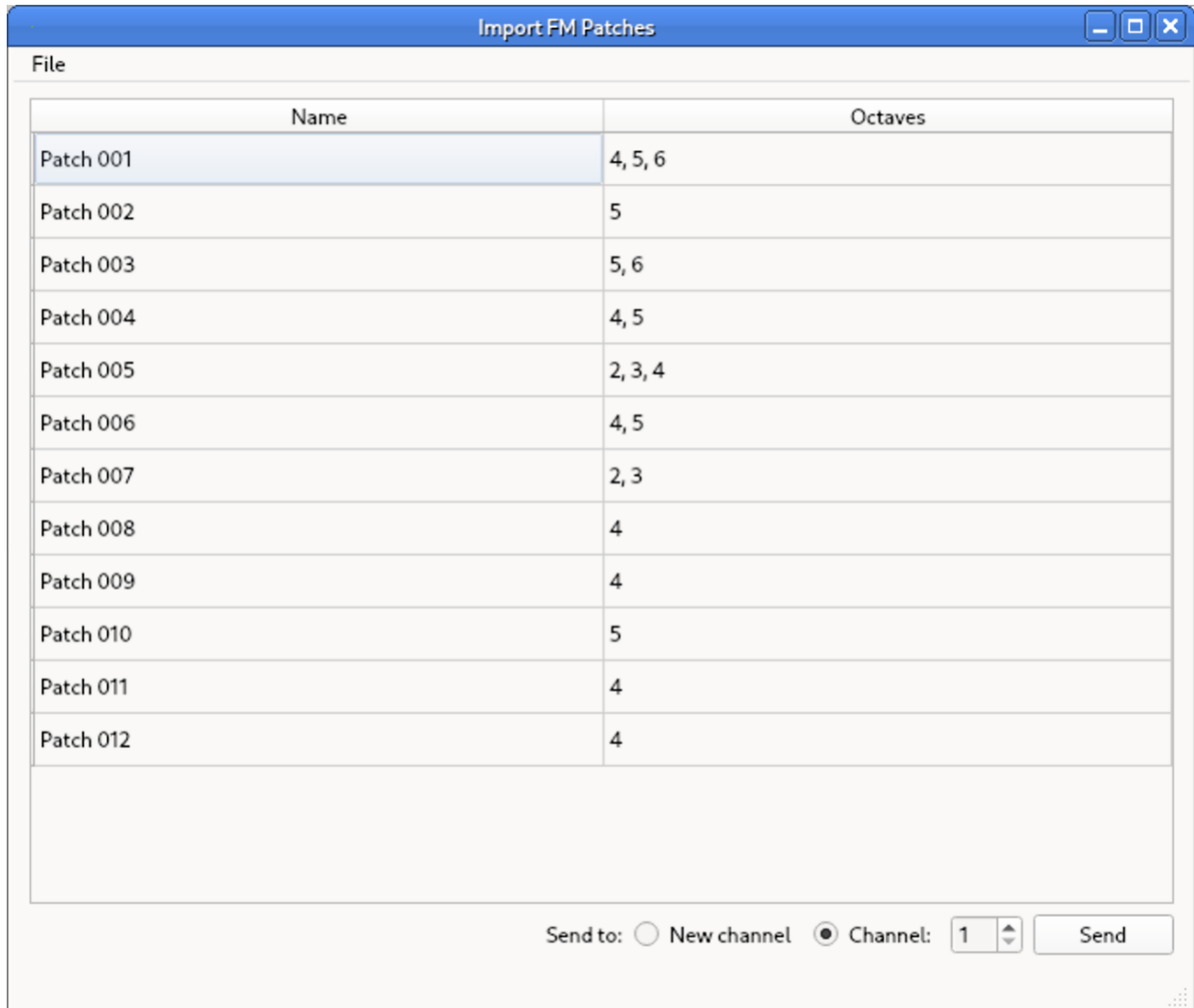
How does it work?

A VGM file is essentially a stream of timed writes to the sound generators' memory at different addresses. The utility reads the VGM file and every time a write to the YM2612 is encountered, the write is performed on internal memory. In this way, the most recent state of the YM2612 memory is kept. When a write signaling "key on" is encountered, the FM settings of the corresponding channel are read from the internal memory, as well as the octave of the note that was triggered. The unique patches that are found in this process are listed in the utility's dialog window along with the octaves they were used in.

At the bottom of the window are controls for sending a patch to one of the project's channels. After testing out a patch, you can rename it in the left column above. The File menu allows you to save one or all of the patches imported.

Going Beyond the Hardware

While developing this project I used the Audacious media player for Linux to listen to VGMs and compare the output with the Chromasound. This was made possible by a plugin for Audacious that supports VGMs. It works by **emulating** the two sound chips, that is, implementing them in software. With a bit of work, I was able to incorporate that plugin directly into Chromasound Studio. While the default mode of the program will expect a Chromasound HAT, the program can be configured to use the emulator instead.

The emulator is generally faster. It sounds very good, but it sounds a bit unlike the actual YM2612 and SN76489. As an example, decays seem shorter in the emulator. In addition, the PCM fidelity is unrealistically high, but that's not necessarily a bad thing.

Appendix A: FM Algorithms

Bass, distortion guitar, high hat
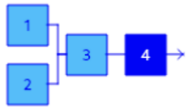
Harp, square tone
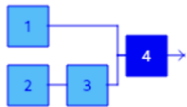
Bass, brass, electric guitar, piano, woodwinds
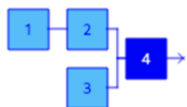
Chimes, guitar, strings
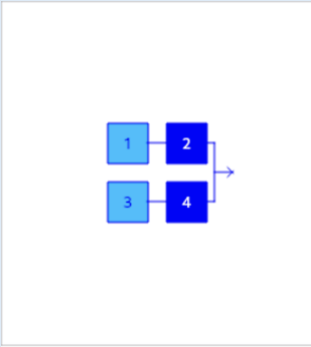
Bass drum, bells, chorus, flute, snare drum, tom-tom
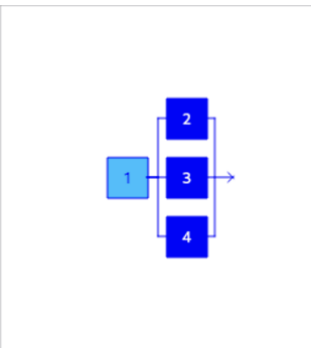
Brass, organ
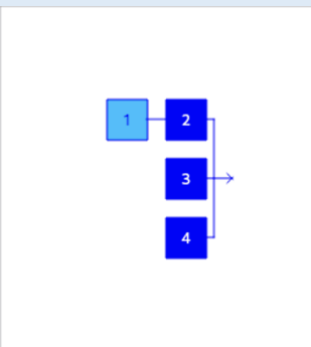
Bass drum, organ, snare drum, tom-tom, vibraphone, xylophone
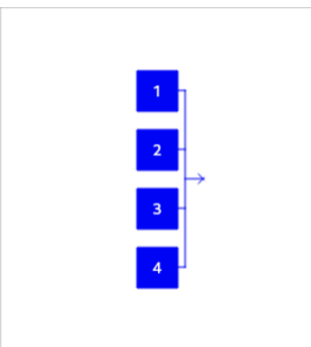
Pipe organ

Appendix B: CSS for Chromasound Studio

The Edit menu exposes a Styles dialog where CSS can be used to change the color and appearance of different parts of the GUI. Here is a listing of all of the stylable properties.

Channels


```
ChannelWidget LED {
    qproperty-color: rgb(0, 212, 0);
}
```
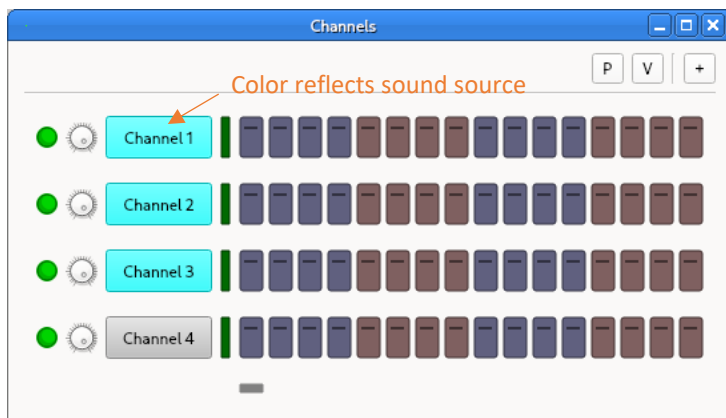


```
ChannelWidget {
    qproperty-toneColor: cyan;
    qproperty-noiseColor: lightGray;
    qproperty-fmColor: magenta;
    qproperty-pcmColor: rgb(128, 0, 255);    /* purple */
    qproperty-ssgColor: green;
    qproperty-melodyColor: yellow;
    qproperty-rhythmColor: rgb(255, 128, 0); /* orange */
    qproperty-romColor: rgb(0, 128, 255);    /* light blue */
}
```

LED (color when on and when selected)

```
RectLED {
    qproperty-color: rgb(0, 212, 0);
    qproperty-selectedColor: cyan;
}
```


Step sequencer

(step colors when on)

```
StepSequencerWidget {
    qproperty-stepColor: rgb(192, 192, 255);
    qproperty-otherStepColor: rgb(255, 192, 192);
    qproperty-activeStepLightColor: rgb(255, 192, 0);
    qproperty-stepRadius: 2;
}
```
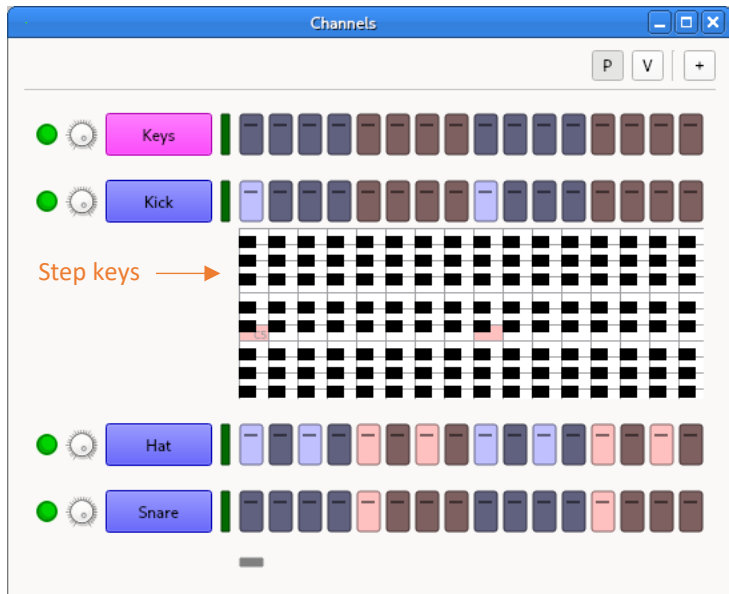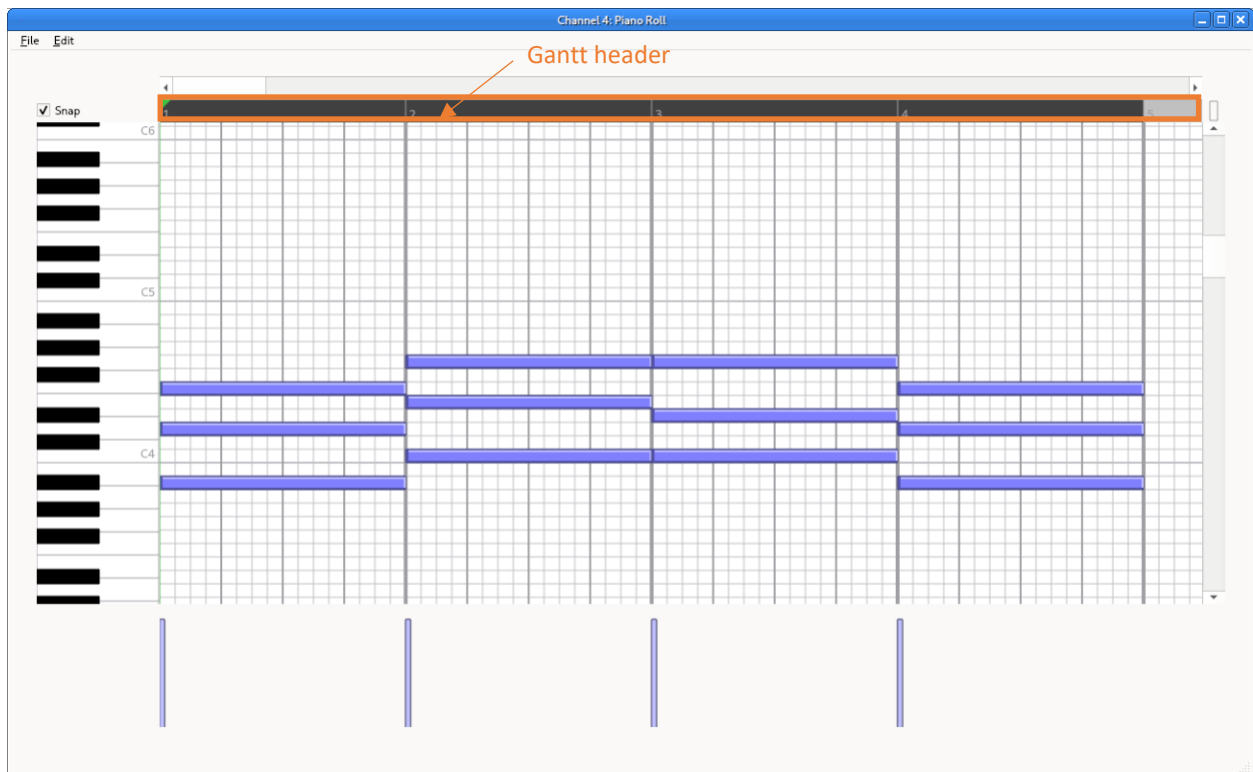
```
StepKeysWidget {
    qproperty-outlineColor: gray;
    qproperty-whiteKeyColor: white;
    qproperty-blackKeyColor: black;
    qproperty-activeKeyColor: rgb(255, 192, 192);
}
```
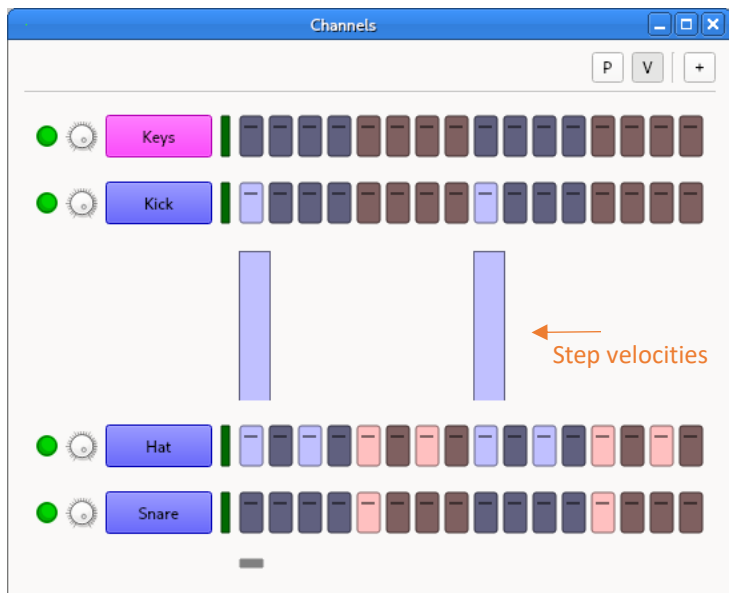


```
StepVelocitiesWidget {
    qproperty-barColor: rgb(192, 192, 255);
}
```
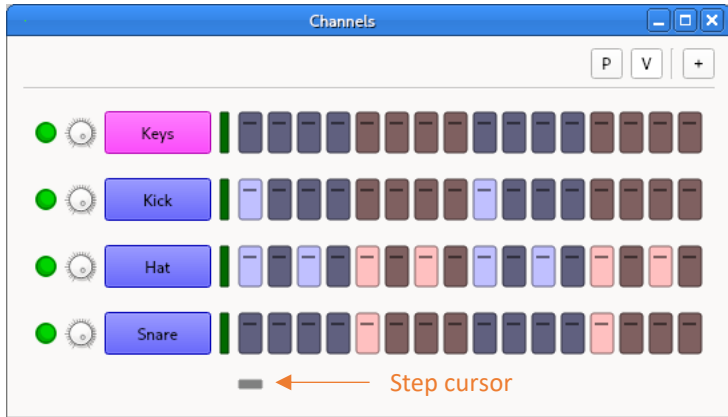
```
StepCursorWidget {
    qproperty-color: darkGray;
}
```
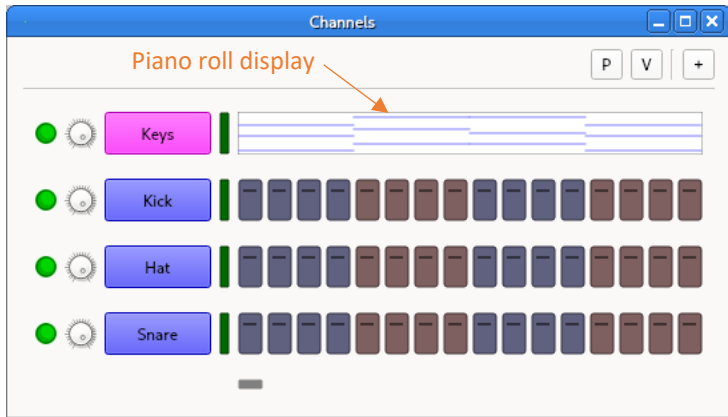


```
PRDisplayWidget {
    qproperty-borderColor: rgb(128, 128, 128);
    qproperty-backgroundColor: rgb(255, 255, 255);
    qproperty-cursorColor: rgb(64, 192, 64);
    qproperty-itemColor: rgb(128, 128, 255);
}
```

FM Settings
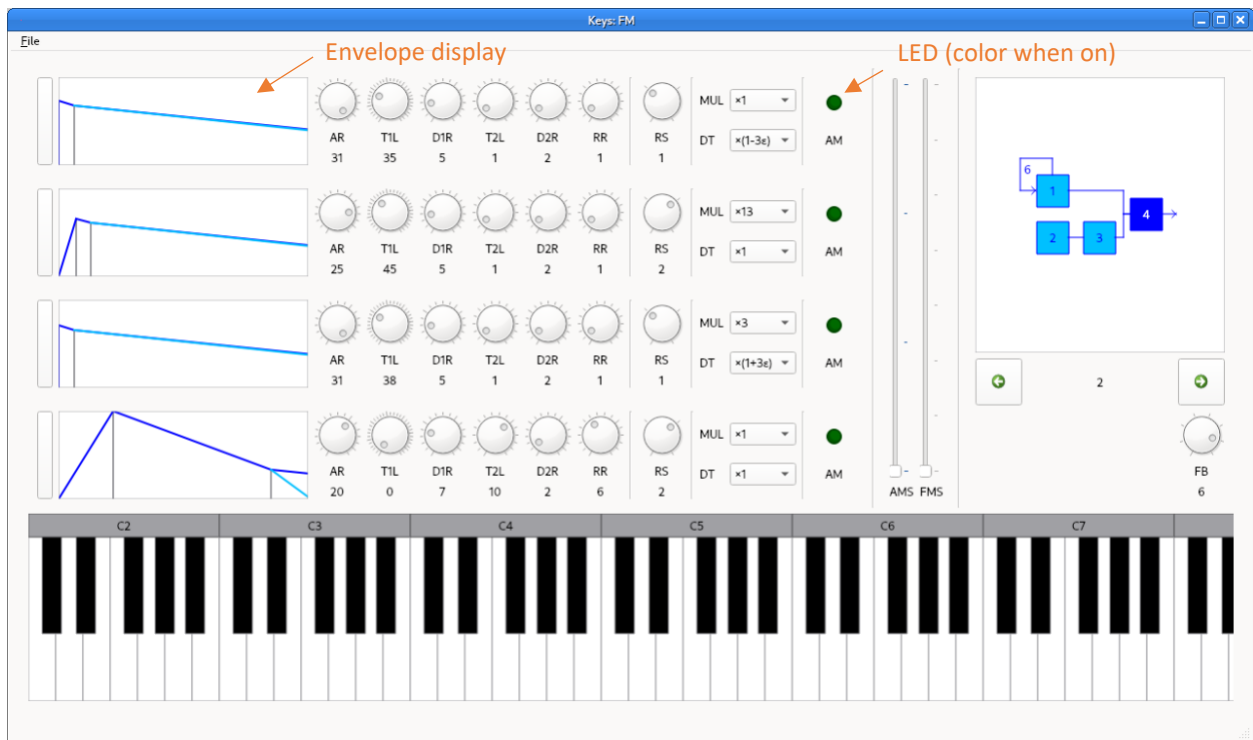

```
OPNEnvelopeDisplayWidget {
    qproperty-backgroundColor: white;
    qproperty-borderColor: gray;
    qproperty-envelopeColor: blue
    qproperty-levelColor: gray;
    qproperty-releaseColor: rgb(0, 192, 255);
}
```

```
OPNOperatorWidget LED {
    qproperty-color: rgb(0, 212, 0);
}
```
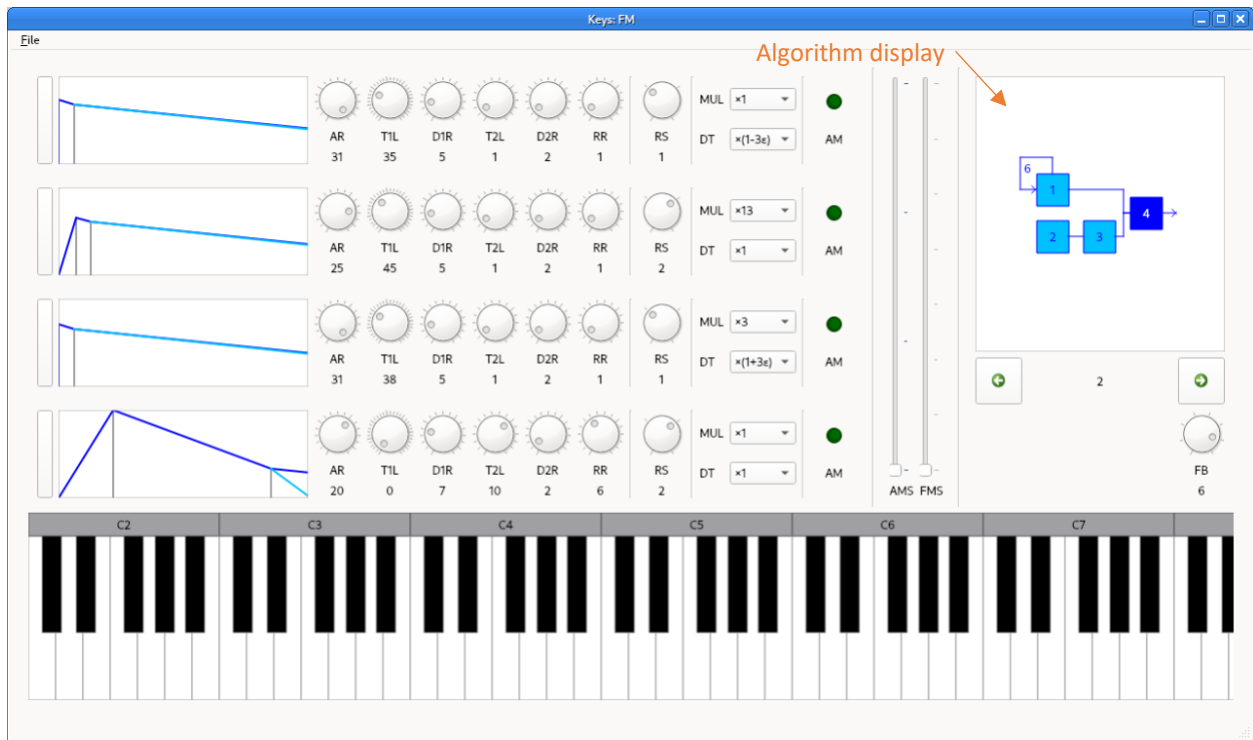
Algorithm display

```
AlgorithmDisplayWidget {
    qproperty-backgroundColor: white;
    qproperty-borderColor: gray;
    qproperty-operatorColor: rgb(0, 192, 255);
    qproperty-operatorTextColor: blue;
    qproperty-slotColor: blue;
    qproperty-slotTextColor: white;
    qproperty-edgeColor: blue;
}
```
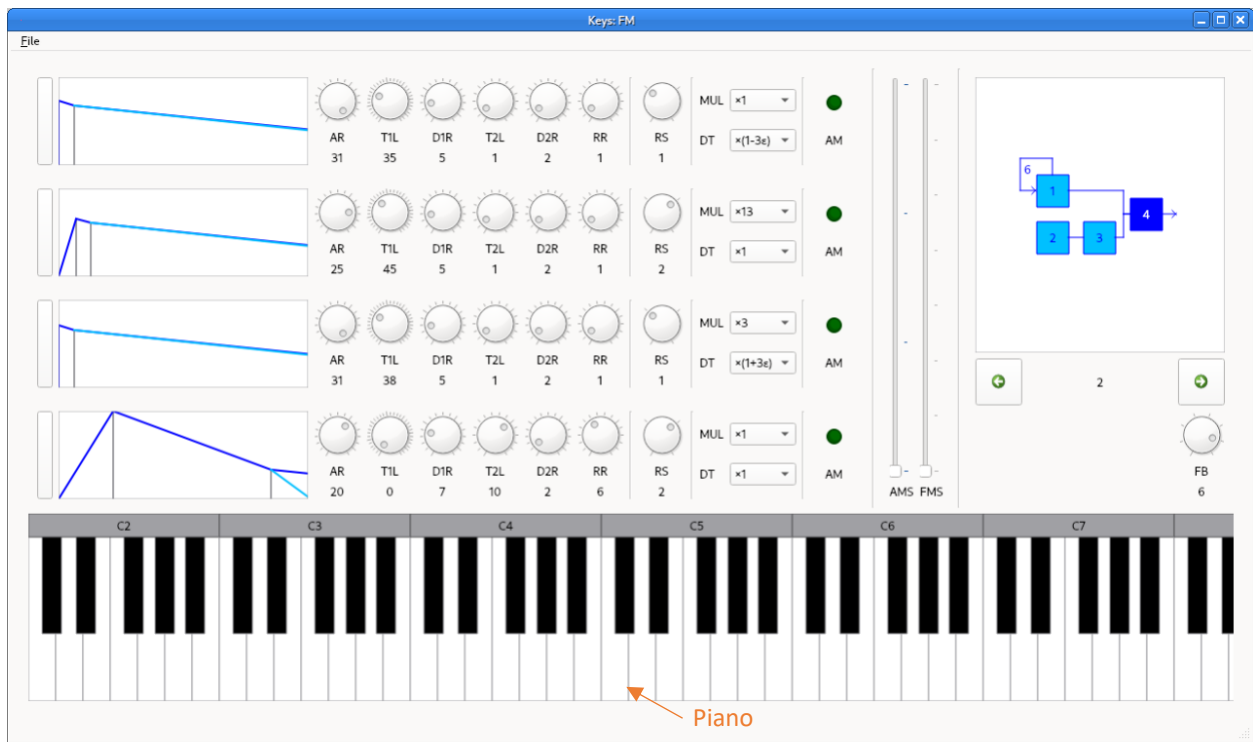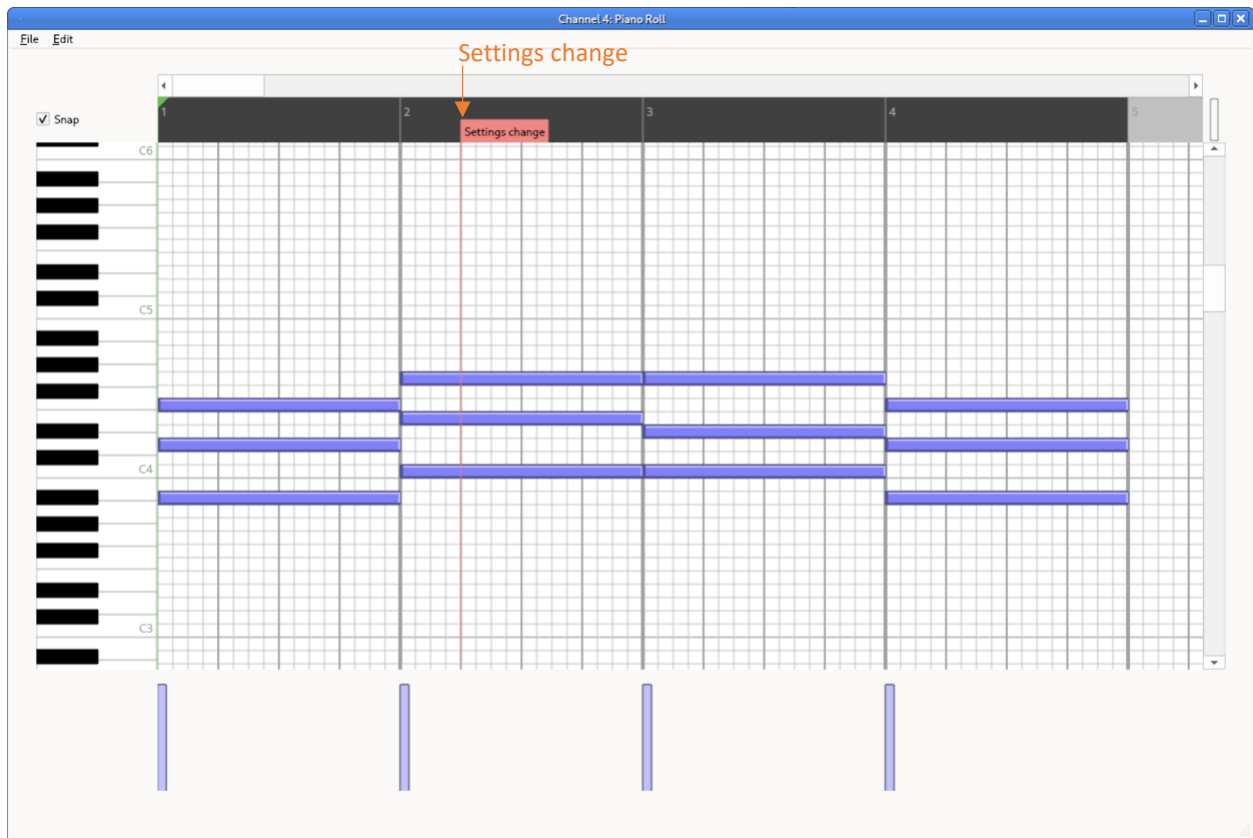
Piano

```
PianoWidget {
    qproperty-outlineColor: gray;
    qproperty-whiteKeyColor: white;
    qproperty-blackKyColor: black;
    qproperty-activeKeyColor: rgb(255, 192, 192);
    qproperty-headerColor: gray;
    qproperty-headerTextColor: black;
}
```

Gantt Chart

Gantt chart
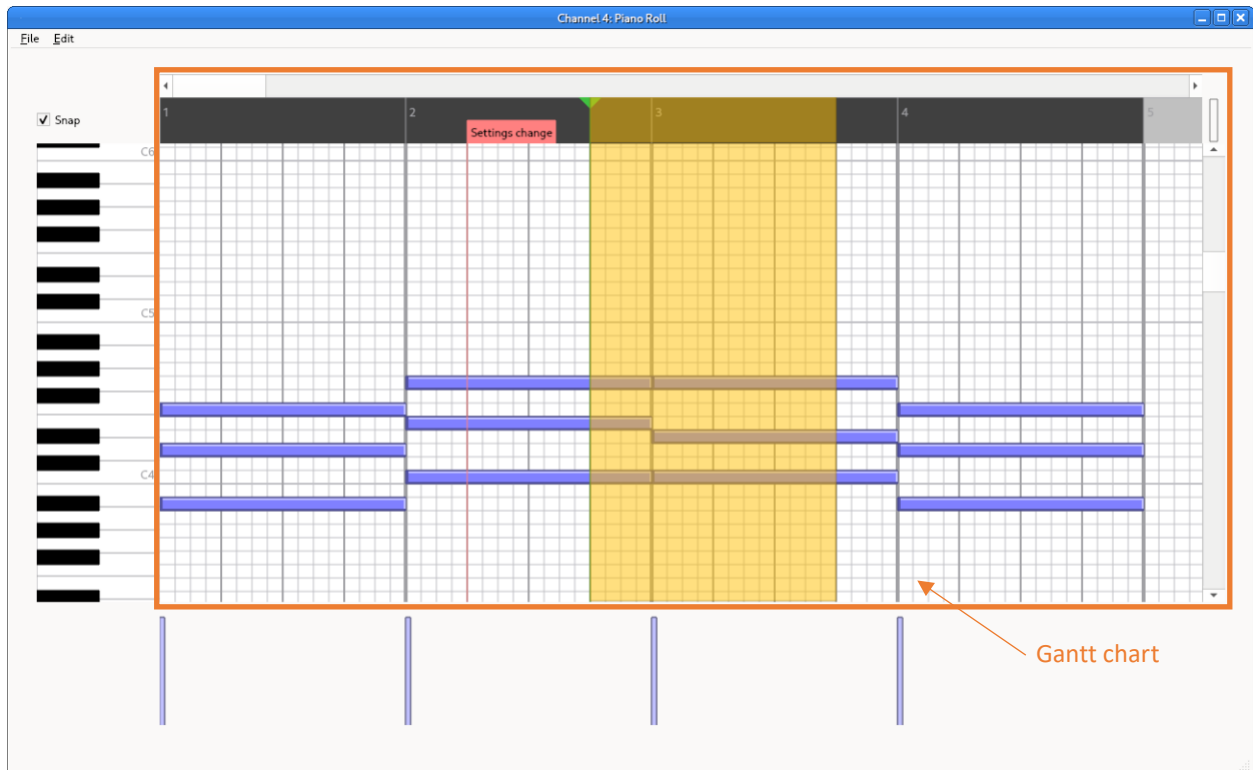
```
[PianoRollWidget|PlaylistWidget] GanttWidget {
    qproperty-cursorColor: rgb(64, 192, 64);
    qproperty-selectionColor: rgb(255, 192, 0);
}
```

```
[PianoRollWidget|PlaylistWidget] GanttHeaderWidget {
    qproperty-activeColor: rgb(64, 64, 64);
    qproperty-inactiveColor: lightGray;
    qproperty-activeForegroundColor: gray;
    qproperty-inactiveForegroundColor: gray;
}
```
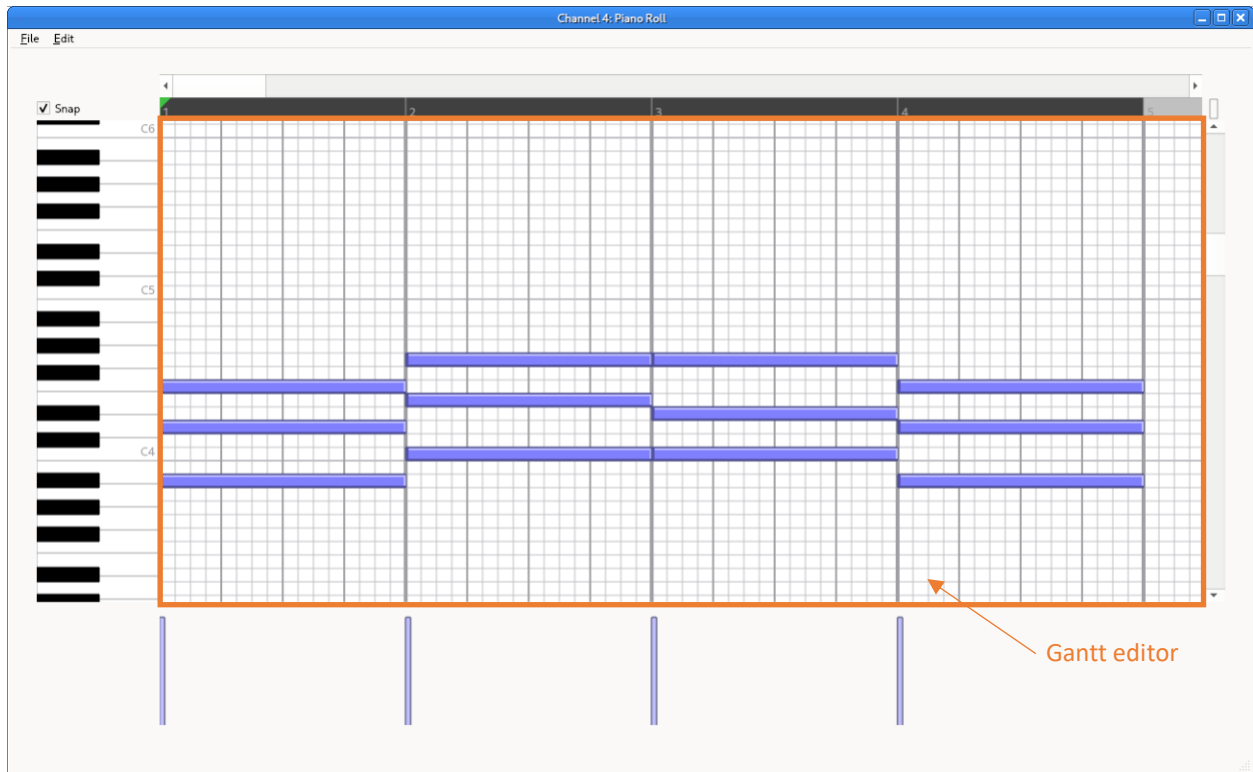
Gantt editor

```
[PianoRollWidget|PlaylistWidget] GanttEditorWidget {
    qproperty-backgroundColor: white;
    qproperty-borderColor: gray;
    qproperty-itemColor: rgb(128, 128, 255);
    qproperty-areaSelectionColor: rgb(192, 192, 255);
}
```
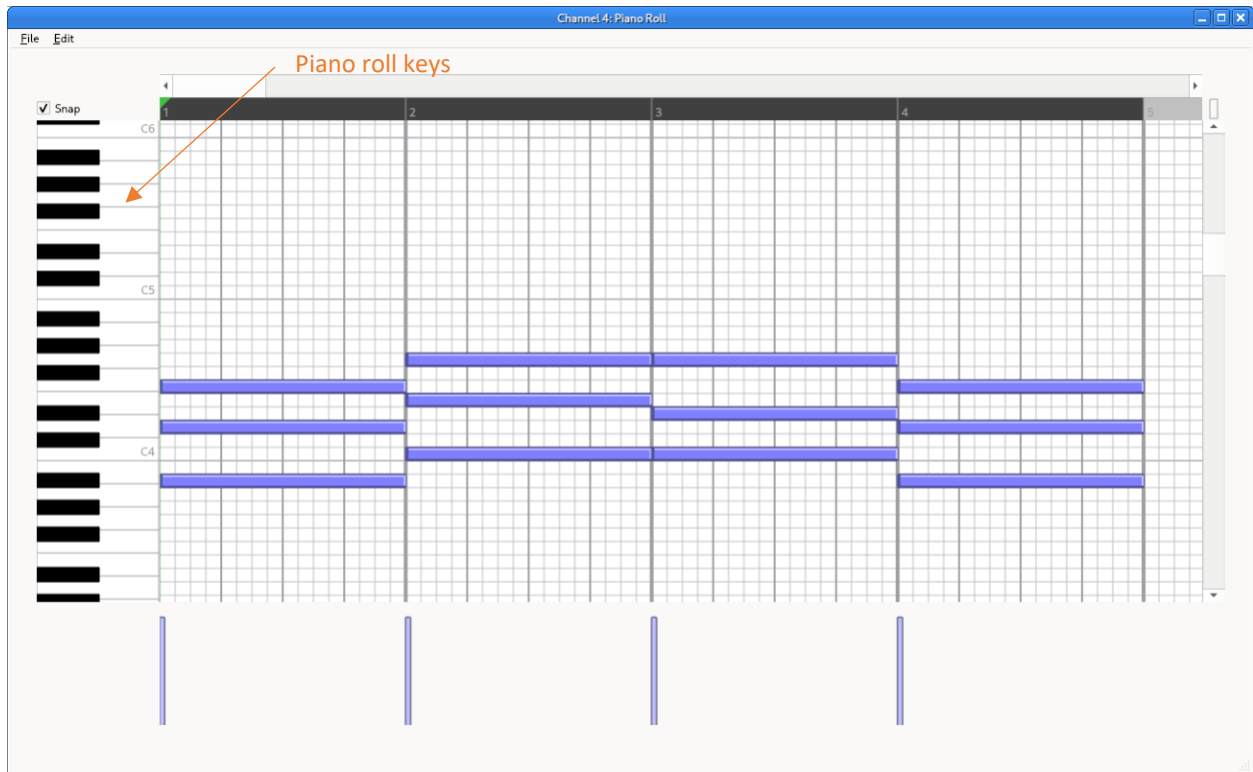
Piano Roll


```
PianoRollKeysWidget {
    qproperty-outlineColor: gray;
    qproperty-whiteKeyColor: white;
    qproperty-blackKeyColor: black;
    qproperty-activeKeyColor: rgb(255, 192, 192);
}
```

Piano roll velocities
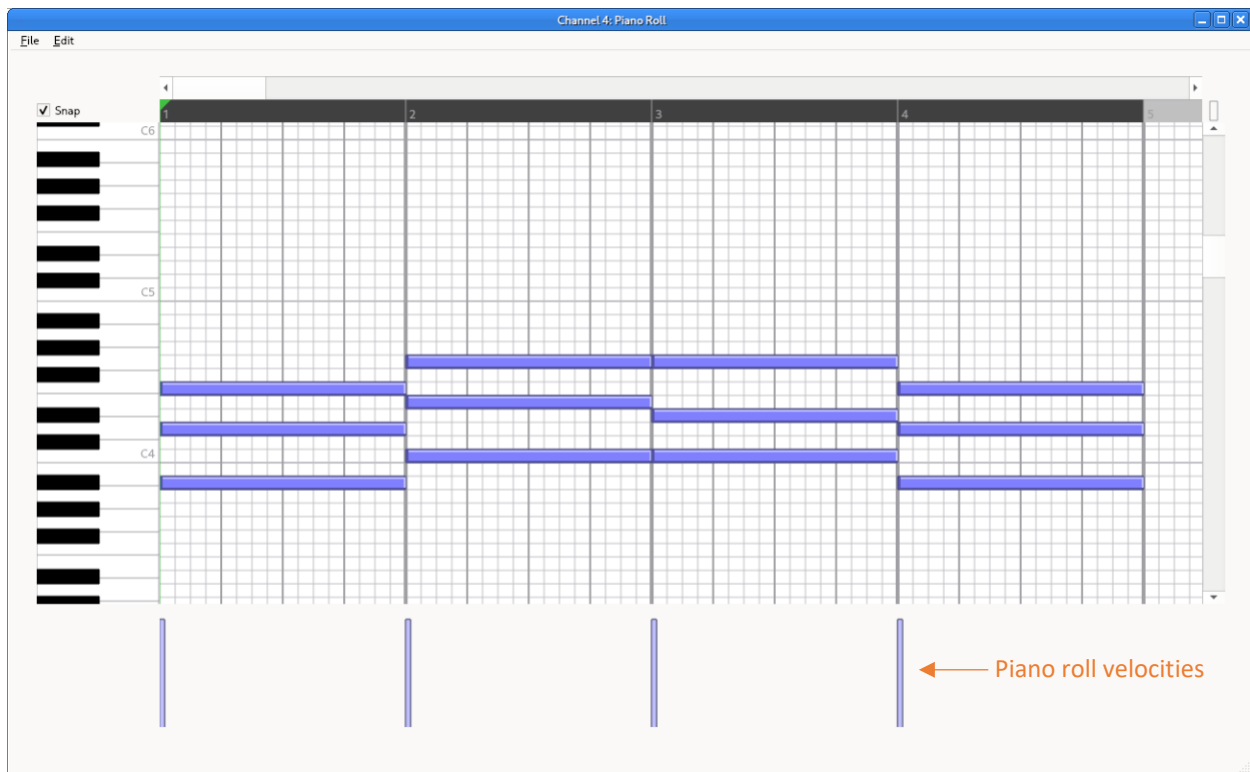
```
PianoRollVelocitiesWidget {
    qproperty-barColor: rgb(192, 192, 255);
}
```

```
PianoRollWidget {
    qproperty-settingsChangeColor: rgb(255, 128, 128);
}
```

Playlist
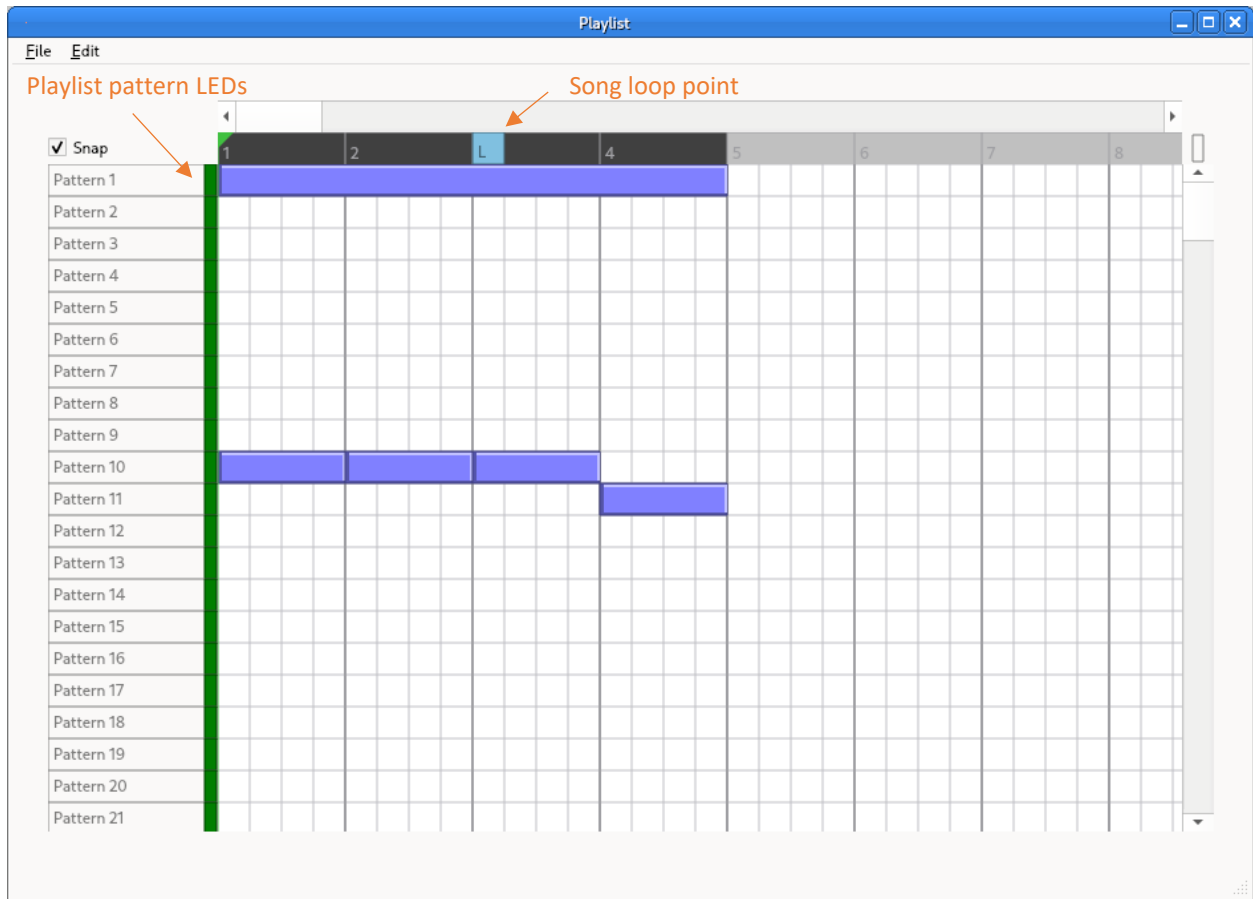

```
PlaylistPatternsWidget {
    qproperty-ledColor: green;
}
```

```
PlaylistWidget {
    qproperty-loopColor: rgb(128, 192, 224);
}
```
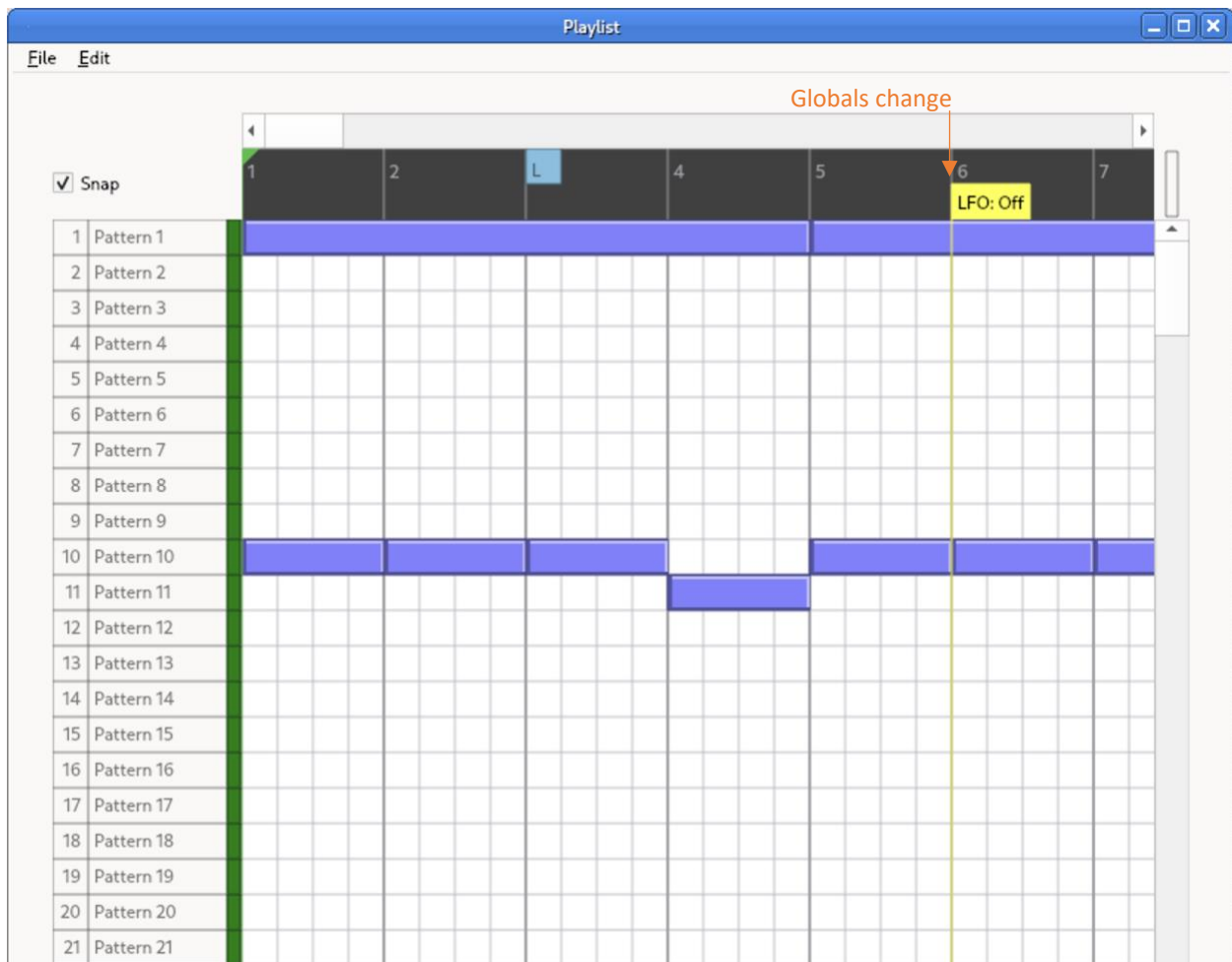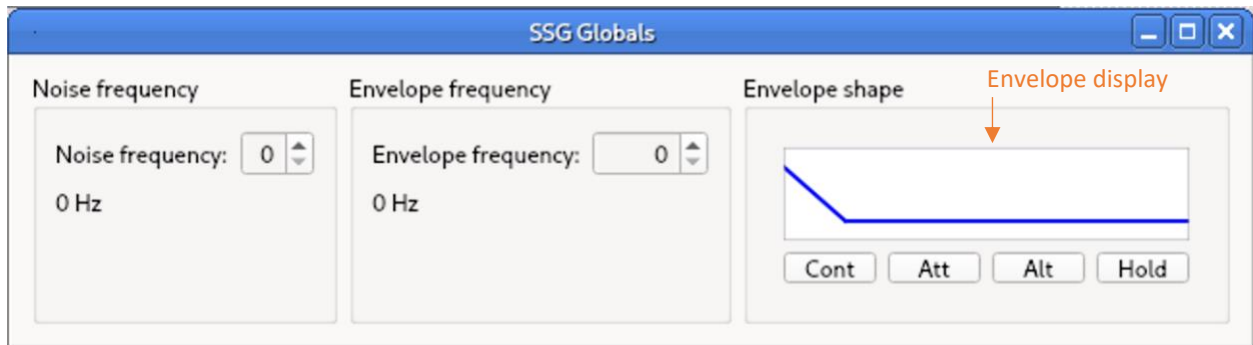
```
PlaylistWidget {
    qproperty-lfoChangeColor: rgb(255, 255, 64);
    qproperty-noiseFreqChangeColor: rgb(224, 224, 224);
    qproperty-envelopeFreqChangeColor: rgb(64, 255, 64);
    qproperty-envelopeShapeChangeColor: rgb(128, 128, 255);
    qproperty-userToneChangeColor: rgb(255, 128, 0);
}
```
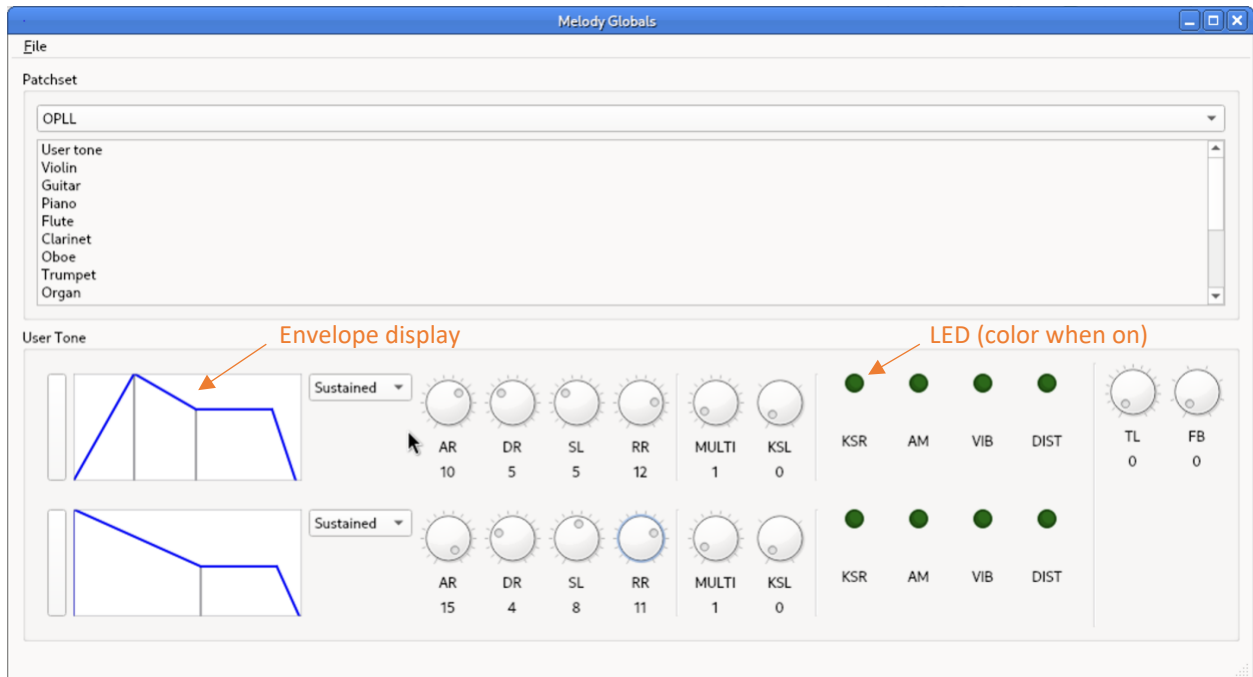
SSG Globals Editor


```
SSGEnvelopeDisplayWidget {
    qproperty-backgroundColor: white;
    qproperty-borderColor: gray;
    qproperty-envelopeColor: blue}
}
```
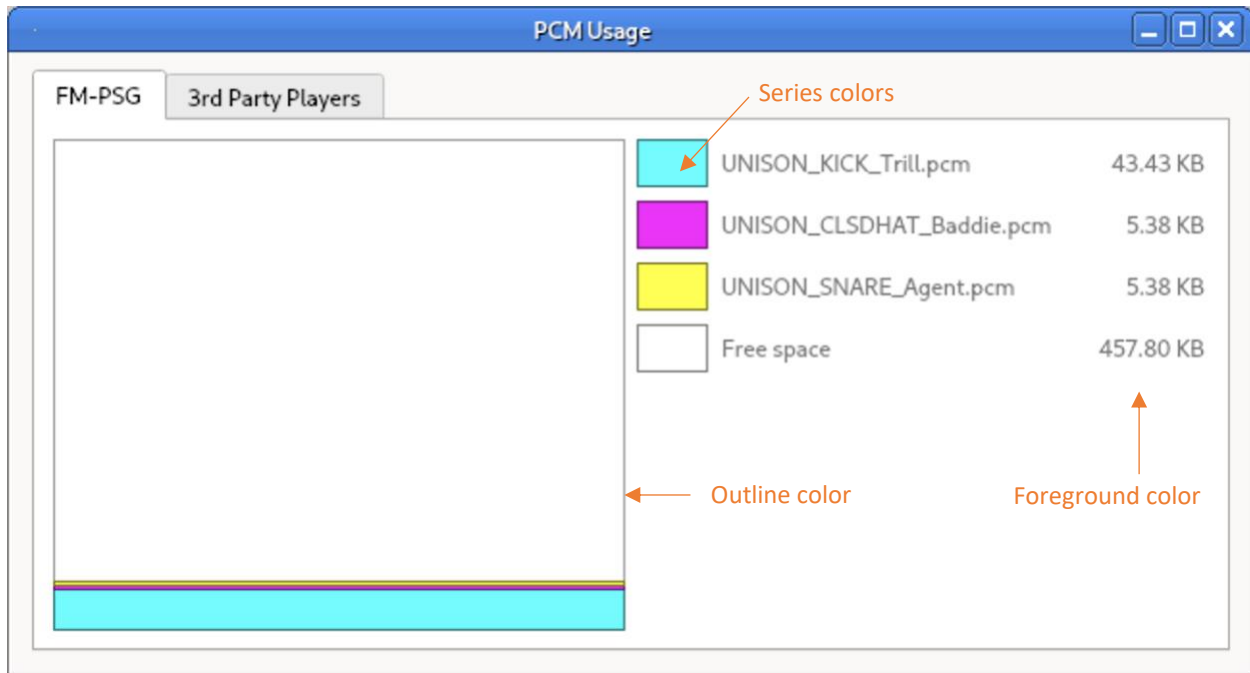
Melody Globals Editor


```
OPLEnvelopeDisplayWidget {
    qproperty-backgroundColor: white;
    qproperty-borderColor: gray;
    qproperty-envelopeColor: blue
    qproperty-levelColor: gray;
}
```

```
OPLOperatorWidget LED {
    qproperty-color: rgb(0, 212, 0);
}
```

PCM Usage Tool


```
PCMUsageDialog {
    qproperty-color1: cyan;
    qproperty-color2: magenta;
    qproperty-color3: yellow;
    qproperty-color4: lightGray;
    qproperty-color5: red;
    qproperty-color6: green;
    qproperty-color7: blue;
    qproperty-color8: darkMagenta;
    qproperty-freeSpaceColor: white;
}
```

```
PCMUsageDisplayWidget {
    qproperty-outlineColor: gray;
}
```

```
PCMUsageLegendWidget {
    qproperty-foregroundColor: gray;
}
```

Other

To apply styles globally, apply them to QWidget or any of its derivatives.

```
QWidget {
    background-color: rgb(128, 192, 224);
}
```

The document area can be switched to tabbed mode.

```
MdiArea {
    qproperty-viewMode: "tabs";
}
```

```
MdiArea {
    qproperty-viewMode: "windows";
}
```