Solar Panel Voltage Converter for IoT Devices

Yes we CAN exploit indoor lighting

By **Sunil Malekar** (Elektor India labs)

The goal of this microcontroller-free project is to design a compact supply to power small indoor IoT devices from a solar panel, indoors (!). A key aspect of the project is its ability to operate from an input source as weak as 7.5 µW so that a low cost mini solar panel can be used. Not forgetting battery backup and extremely compact size, all thanks to the LTC3129 IC.

The Internet of Things (IoT) is said to define a network of physical objects embedded with electronics, software, sensors and connectivity to provide services by exchanging data with the connected devices. It's also a globally interconnected *continuum* of devices and objects and things that emerged with the rise of cheap, license free RFID technology. Alternatively some consider the IoT as a scenario in which objects, animals or people are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. Eloquent definitions all, but Elektor's field being electronics, we're interested in components, components... what components?

Starring...

Among the key features of the Types LTC3129 and LTC3129-1 buck/boost converters from good old Linear Technology are a fixed 1.2-MHz operating frequency, current mode control, internal loop compensation, automatic Burst Mode operation or low noise PWM mode, an accurate RUN pin threshold to allow the UV_{LO} threshold to be programmed, a power-good output and an MPPC (maximum power point control) function for optimizing power transfer when operating from photovoltaic cells. The full story is at [1] and [2]. Let's see what these devices can do if it comes to powering IoT devices in an eco-friendly way.

Main Features

- Stores energy from indoor lighting
- Solar panel V_{out} 5 V typ., 42 µA
- Input power down to 7.5 µW usable
- 3.2 V typical out
- Choice of LTC3129 or LTC3129-1 converter IC, board configurable for either
- Easily configurable for many output voltages
- Optional backup battery
- Optional super-charge capacitor
- Ready-assembled unit with LTC3129-1 fitted, 3V3, MPPC mode.

in collaboration with DESIGNSPARK

Power to the IoT (too)

The circuit shown in **Figure 1** exploits the unique ability of the LTC3129 and LTC3129-1 to start up and operate from an input power source as "weak" as 7.5 µW (microwatts) — making them capable of operating from small, low-cost solar cells with indoor light levels less than 200 lux.

To make the low current start-up possible, both the LTC3129 and LTC3129-1 draw a lousy two microamps of current (even less in shutdown) until three conditions are satisfied:

- the voltage on the RUN pin must exceed 1.22 V (typical);
- the voltage on the V_{IN} pin must exceed 1.9V (typical);
- V_{CC} (which is internally generated from V_{IN} but can also be supplied externally) must exceed 2.25 V (typical).

Until all three of these conditions are satisfied, the part remains in a 'soft-shut-down' or standby state, drawing just 2 µA. This allows a weak input source to charge the input storage capacitor until the voltage is high enough to satisfy all three conditions, at which point the LTC3129/LTC3129-1begins switching, and V_{OUT} rises to regulation, provided the input capacitor has sufficient stored energy.

The circuit features battery backup circuitry around BAT1 (sorry about the missing T). A CR2032 backup battery provides power when solar power is insufficient. The LTC3129 is used in this case, allowing V_{OUT} to be programmed for 3.2 V to better match the voltage of the coin cell. With 5 V input from the solar panel on connector K1 you can expect an output

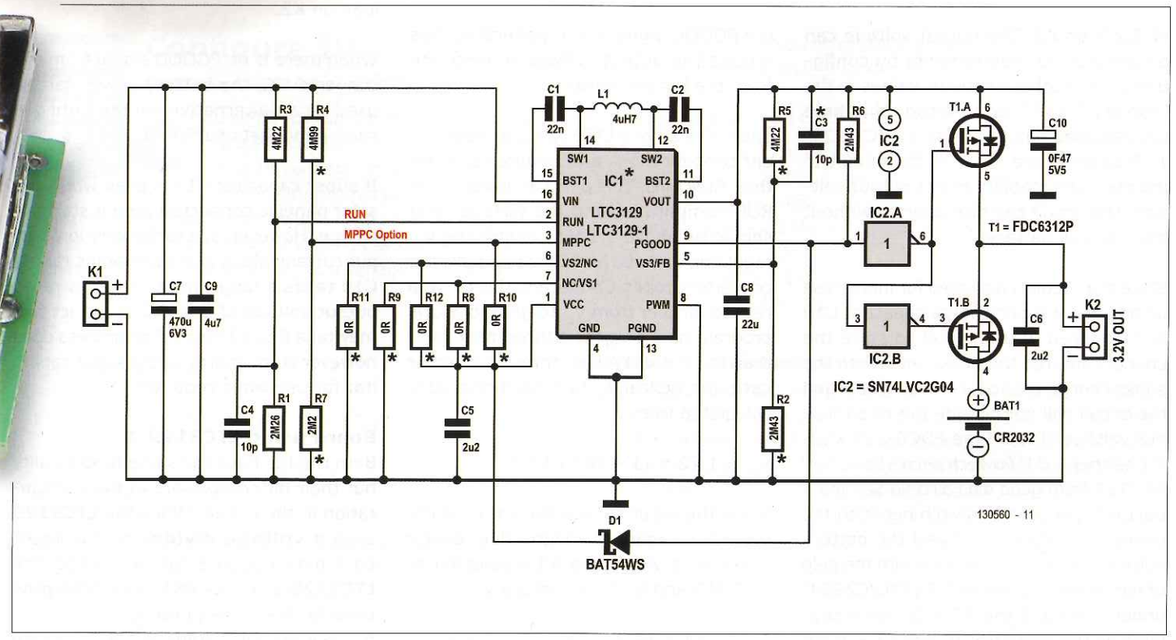


Figure 1. At the heart of the converter is either an LTC3129 or an LTC3129-1 IC, depending on your choice. A number of components need to be configured as a function of the IC you decide to use. Hardly a microwatt of solar power is wasted in this circuit.

Component List

Resistors
R1 = 2.26MΩ 1%, 0.063W, 0603 *
R2,R6 = 2.43MΩ 1%, 0603 (R2*)
R3,R5 = 4.22MΩ 1%, 0603 (R5*)
R4 = 4.99MΩ 1%, 100mW, *
R7 = 2.2MΩ 1%, 0603 *
R8-R12 = 0Ω, 0603 *

Capacitors
C1,C2 = 22nF, 25V, 0603
C3,C4 = 10pF, 25V, 0603
C5,C6 = 2.2µF, 25V, 0603
C7 = 470µF, 6.3V, case D
C8 = 22µF, 10V, 0603
C9 = 4.7µF, 25V, 0603
C10 = 0.47F, 5.5V, Super Capacitor, radial

Semiconductors
IC1 = LTC3129EMSE#PBF or LTC3129-1 *
IC2 = SN74LVC2G04DBVR
T1= FDC6312P (Newark/Farnell # 1700713)
D1 = BAT54WS-E3-08

Inductor
L1 = 4.7µH 20%, SMD

Miscellaneous
BAT1 = Lithium Battery, CR2032, coin cell, 3V, 20mm, with PCB mount holder
K1,K2 = 2-pin pinheader
Solar panel, AM-1815CA, 58.1x 48.6mm (Panasonic)
PCB # 130560-1 v. 1.0 from Elektor Store
Ready assembled unit, # 130560-91 from Elektor Store. Version: LTC3129-1, 3.3V out, MPPC, battery excluded.

* Component, value and/or use subject to user configuration; see text.

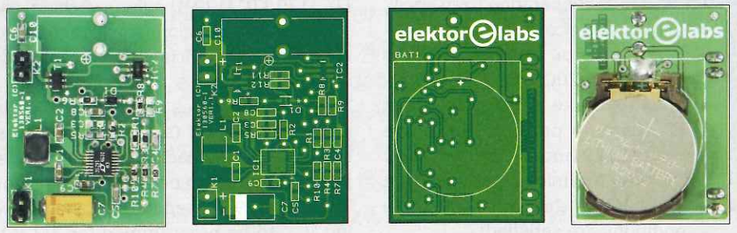


Figure 2. The very compact printed circuit board designed for the Solar Panel converter employs SMDs for the most part. The board is available not only unpopulated (# 130560-1) but ready assembled also (# 130560-91, see text and component list).

of 3.2 V on K2. The output voltage can be adapted to requirements by configuring the feedback resistor values in the case of LTC3129, or by setting the three programmable pins in case of LTC 3129-1. Provisions are made in the circuit to maintain the stability of the output voltage. The circuit can also operate without the coin cell battery.

Since the circuit is designed for indoor use an additional super-charge capacitor C10 is introduced in the circuit to store the energy captured from daylight. When the super charge capacitor is being charged the circuit will not provide the rated output voltage. T1, a Type FDC6312P *Dual P-Channel 1.8 V Powertrench® Specified MOSFET* from good old Fairchild Semiconductor [3] is used to switch between the convertor output (V_{OUT}) and the battery output. The selection is done with the help of two inverter gates in IC2 (74LVC2G04) under control of the PGOOD signal supplied by the convertor IC when the input and output voltages are at their rated values. If the input voltage is too low then the PGOOD signal is not generated thus forcing the output voltage to emanate from the backup battery.

The LTC3129 and LTC3129-1 have different configurations and requirements for their RUN pins. This pin is an input to the RUN comparator, and the voltage on it should be above 1.1 V to enable the V_{CC} regulator, and above 1.22 V to enable the converter proper. Connecting this pin to a resistor divider from V_{IN} to ground allows programming a V_{IN} start threshold higher than the 1.8 V (typical) threshold. In our case, the typical V_{IN} turn-on threshold is calculated from

$$V_{IN} = 1.22 \times [1 + (R3 / R1)]$$

Since the input source current is of the order of microamps, a high value resistor is required. Assuming R3 is selected as 4.22 MΩ and V_{IN} taken as 3.5 V:

$$1.22 \times [1 + (4.22 \times 10^6 / R1)] = 3.5$$

Hence R1 = 2.26 MΩ here.

More configuration options, calculations and component parameters for the LTC3129 and LTC3129-1 are shown in the **Configure It!** inset.

Board using LTC3129

When a solar panel is connected to the input connector K1 whose voltage and current is 5 V, 42 µA, capacitor C7 starts to charge. Since the charging current is too small, it takes 20-30 seconds to reach a voltage of 5 V. When the input supply voltage reaches 3.5 V, the voltage on the RUN pin due to voltage divider R3/R1 equals to 1.22 V which enables the converter output. At the same time the PGOOD signal is generated and IC2.A causes dual transistor T1 to pass the output voltage from converter IC1 to connector K2. The feedback from V_{OUT} is sent to the feedback pin of IC1 via voltage divider built around R5 and R2 which determines the output voltage available at connector K2 (set to 3 to 3.2 V). The feed-forward capacitor C3 on the feedback divider is used to reduce burst mode ripple on the output voltage.

Assuming BAT1 is connected (a 3-V coin cell), in case of no solar panel or low light conditions then IC2 switches on transistor T1.B to pass output current to the load on K2.

When there is no PGOOD signal from the converter IC, the battery power can be used as an alternative source until the solar panel gets sufficient light.

If super capacitor C10 is used when the solar panel is connected then it starts to charge. However, due to the very low output current which is in microamps range, C10 takes a long time to reach a rated output voltage of 3 to 3.2 V, in fact this may take 8 to 12 hours. The process does however store energy in the super capacitor for use when required.

Board using LTC3129-1

Both the ICs have the same functionality but their differences are in the configuration in the circuit. While the **LTC3129** uses a **voltage divider** on the feedback pin to set the output voltage, the **LTC3129-1** has its **VS1, VS2, VS3 pins** used for the same purpose.
A 3.3-V output voltage is obtained with VS1 (pin 7) of the LTC3129-1 connected to the V_{CC} pin through 0-ohm resistor R8; VS2 (pin 6) connected directly to ground; and VS3 (pin 5) also connected to ground but through resistor R2 using another 0-ohm resistor.

The MPPC pin is taken to V_{CC} through 0-ohm resistor R10 as our functionality requires an input source stronger than 10 mA. In case of other configuration and input source this function can be used and set by using voltage divider resistors R4 and R7 as discussed. Other functions like battery backup and super-charge capacitor C10 are similar to the board with the LTC3129.

Building and testing

Provided you have professional (SMD) production equipment, or access to it, the printed circuit board with its overlay pictured in **Figure 2** can be built using either the LTC3129 or the LTC3129-1 with their required components. That's why a Component List is published here. Fully respecting all 100% DIY readers out there we kindly advise however that the board is also available ready-assembled through the Elektor Store as item # **130560-91**, it's the LTC3129-1 version, MPPC mode, 10 mA min. in, 3.3 V, backup battery not included.

Web Links
[1] LTC3129: www.linear.com/docs/42736
[2] LTC3129-1: www.linear.com/docs/42735
[3] FDC6312P: www.fairchildsemi.com/pf/FD/FDC6312P.html

The components marked * in the schematic are optional and/or need to be selected depending on the IC used. The Feedback, RUN, and MPPC voltage divider resistor values can vary depending on your choice as well as on the input and output supply.
In the case of the LTC3129-1, the output can be configured up to 5 V just by changing the resistor configuration on VS1 and VS2 pins of the IC. Be sure to fit either R11 or R12, not both.
The use of a backup battery and the super-charge capacitor depends on requirements — the board can work without them.

Waste Not Want Not

To power the board, a solar panel with an output voltage rating up 5 V is connected to K1 and the output load, to K2. By connecting a Sanyo AM-1815 solar panel, the board can operate from indoor lighting, and if super capacitor C10 is used a useful charge gets built.
Let's consider charging at 35 µA "stolen" from a light source. If maintained for 24 hours then it will give a cumulative charge of almost 700 µAh, equal to 7 mA for 6 minutes or is almost 42 mA for 1 minute.

Thus the power supply board is most suitable in applications where the power requirement of the devices is within the above range, which should include many IoT devices that need to wake up, respond briefly and go to cybersleep again. Tell us what you've in mind to power with this circuit — IP address optional. ⏮

(130560)

Configure It!

MPPC on LPC3129 and LPC3129-1

An MPPC (maximum power point control) programming pin is common to both LTC3129 variants. To enable the MPPC functionality this pin is connected to a resistor divider from V_{IN} to ground. If the load on V_{OUT} exceeds the capacity of the power source, MPPC action will reduce the inductor current to regulate V_{IN} to a voltage calculated as:

$$V_{IN} = 1.175 \times [1 + (R4 / R7)]$$

Assuming R4 = 4.99 MΩ:

$$1.175 \times [1 + (4.99 \times 10^6 / R7)] = 3.2 \text{ V}$$

$$[1 + (4.99 \times 10^6 / R7)] = 2.9787 \text{ V}$$
$$R7 = 2.13 \text{ MΩ} \approx 2.2 \text{ MΩ}$$

By setting the V_{IN} regulation voltage appropriately, maximum power transfer from the limited source is assured. Note this pin is noise sensitive; therefore minimize PCB trace length and stray capacitance.
In our example the supply is less than 10 mA and hence MPPC is not used. The pin is connected to V_{CC} through R10 which is a 0-ohm device. If you want MPPC though, remove R10 and configure potential divider R4/R7.

Feedback signal on LTC3129

The feedback pin is a feedback Input to the Error Amplifier. This pin is connected to a resistor divider from V_{OUT} to ground. To get an output voltage of 3.2 V from our converter, assuming R5 = 4.22 MΩ we calculate:

$$V_{OUT} = 1.175 \times [1 + (R5 / R2)]$$
$$R2 = 2.44 \text{ MΩ} \approx 2.43 \text{ MΩ}$$

Output voltage configurations on LTC3129-1

VS1, VS2 and VS3 are the output voltage select pins that need to be connected either to the ground pin or to V_{CC} for programming the output voltage. These pins should not float or go below ground. The configuration of these pins for the desired voltage is given in the table below.

VS3	VS2	VS1	V_{OUT}
0	0	0	2.5 V
0	0	VCC	3.3 V
0	VCC	0	4.1 V
0	VCC	VCC	5 V

albeit rather abundant the QTouch documentation is confusing at the same time

clear about the clock speed of the MCU. The examples seem to be intended for 4 MHz, but the code is not consistent. Furthermore it is not clear if the clock speed is of any importance. To avoid problems we decided to run the MCU on 4 MHz by modifying the clock prescale register `CLKPR`.
The library must be initialized globally as well as for every button/channel:

- Fill in the structure `qt_config_data` (default values worked fine for us);
- For every channel call `qt_enable_key` (example values worked fine for us);
- Call `qt_init_sensing`.

Now you're ready to start sensing. Make sure to regularly call `qt_measure_sensors`. A timer firing every 25 ms is fine, we used Timer1. Inspect the returned value. If the flag `QTLIB_BURST_AGAIN` is set, you must call `qt_measure_sensors` again.
When the flags `QTLIB_NO_ACTIVITY` and `QTLIB_BURST_AGAIN` are cleared you can check the library for active buttons. The best way to do this is by inspecting the array `qt_measure_data.qt_touch_status.sensor_states`.
It is possible to detect multiple key presses at once, but our firmware defaults to one key at a time. The active key is flagged on the serial port with an ASCII string "Sxx" where "xx" is from "00" to "12". Key Up events are not being sent.
Because port D is used for QTouch channels the MCU's hardware serial port is not available. For this reason the software uses a software serial port running at 9600 baud (no parity, 8 data bits, 1 stop bit).

Experiments, hints & kinks

When you play with different materials for the panel overlying the keypad (glass, wood, acrylic plastic, etc.) remember to restart the software every time you have changed something, otherwise the system will not work properly.

Instead of using `qt_measure_data.qt_touch_status.sensor_states` to discover button states you can also call `qt_get_sensor_delta`. This function gives more detailed information, but it requires better knowledge of your hardware. Changing something in the hardware will change the delta values. These values can be very high (no overhead panel) or very low (thick overhead panel) so make sure you know the range of these values for your specific configuration.

A callback `qt_filter_callback` can be registered to filter channel measurements before they are processed. We have added a simple 4-sample averaging filter here.

Changing the default values of structure `qt_config_data` did not seem to have a lot of effect. Only the detect integration limit has a noticeable influence as it slows the system down when the limit is increased. The following commands (terminated with <Enter>) can be sent to the keypad to play with these values:

- **[i|I]** detect integration (DI) limit (default = 4)
- **[n|N]** negative drift rate (default = 20 [x 200 ms])
- **[p|P]** positive drift rate (default = 5 [x 200 ms])
- **[h|H]** drift hold time (default = 20 [x 200 ms])
- **[m|M]** maximum on duration (default = 0 [x 200 ms])
- **[r|R]** recalibration threshold (default = RECAL_50 = 1)
- **[d|D]** Positive recalibration delay DEF_QT_POS_RECAL_DELAY (default = 3)

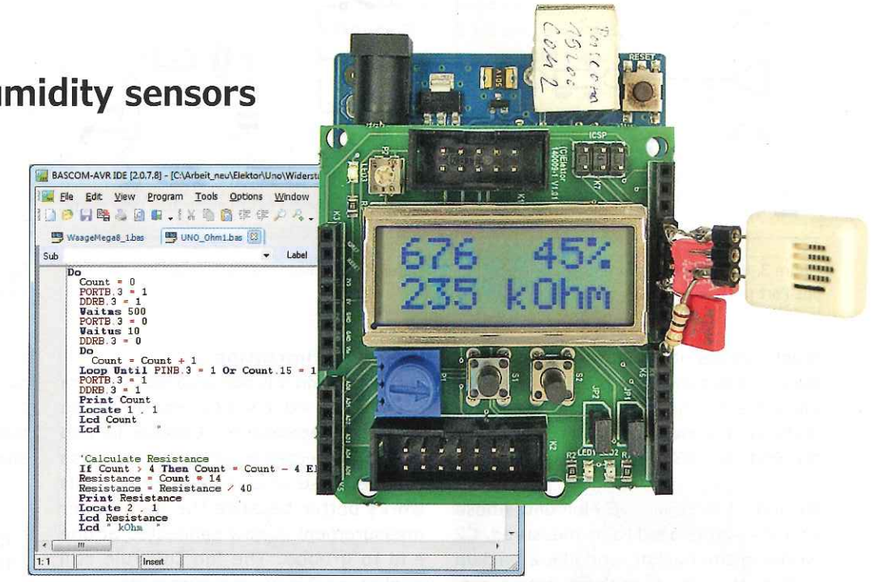
Refer to the QTouch User Manual [3] for more details about these parameters. (130105-I)

Web Links

[1] Atmel QTouch Library: www.atmel.com/tools/qtouchlibrary.aspx
[2] Project Software: www.elektor-magazine.com/130105
[3] Atmel QTouch User manual: www.atmel.com/images/doc8207.pdf

Resistance Measurement with the Arduino

Great for testing humidity sensors



By **Burkhard Kainka** (Germany)

Next to the oscilloscope, the ohmmeter is the most widely used T&M device in the electronics lab. Using this you can measure component characteristics, trace wires, find errors in circuits and evaluate many types of sensor. But a normal ohmmeter or multimeter cannot always fulfill the demands placed upon it, for example when we need to measure alternating current (AC). This is when a microcontroller can help. Sure, another opportunity to use the Arduino Uno, our Extension Shield and Bascom!

The starting point for this project was some newish resistive air humidity sensors, whose resistance varies in the range 1 kΩ to 10 MΩ. The datasheet states expressly that they must be measured using AC. And that's precisely what a regular ohm meter cannot do.

A representative example of these resistive sensors (**Figure 1**) is the HCZ-H8A(N), which you can obtain from Farnell, Conrad, Mantech and other suppliers. The datasheet can be found here [1]. Normally these sensors are driven using an AC voltage of 1 V_{eff}. If you connect them to a signal generator using a voltage divider, you can view the result very clearly on the oscilloscope (**Figure 2**). Put your hand close to the sensor and the humidity increases; you can watch the resistance fall and the signal voltage rise at the input of the oscilloscope.

Why can't you use direct current (DC) for this? The answer is all to do with polarization. Water molecules, as you probably know, have polar characteristics; in other words they are more positive on one side and more negative on the other. Gradually they will align according to the DC applied and change resistance in the process.

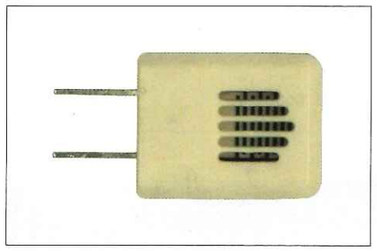


Figure 1. A resistive humidity sensor.

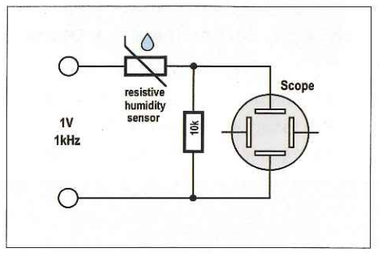


Figure 2. The first test.

The same effect arises when you measure the conductivity of water; here too we must use only AC. A long time ago I even observed this effect while measuring the conductivity of wood as a function of humidity. The resistance starts off low and then rises slowly. A pair of stainless steel nails pushed into damp wood even behave something like a battery that you can charge up. In those days I was amazed. Since then, however, I found out that this is the same principle you have in dual-layer capacitors, also known as supercapacitors or GoldCaps.

Resistance measurement

How do we get round the fact that microcontrollers prefer DC? Or that handling the huge measurement range involved is no easy task for a 10-bit A-to-D converter? Consequently thoughts turned towards *R-C* elements and time measure-

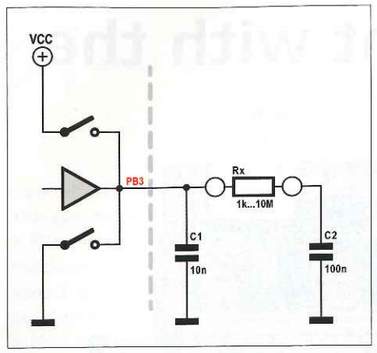


Figure 3. Resistance measurement using only one Port pin.

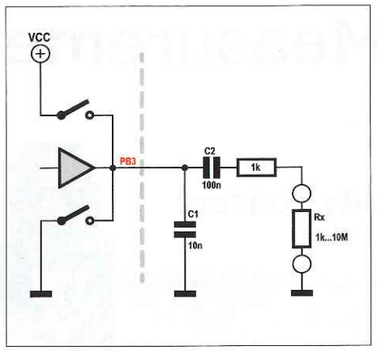


Figure 4. Optimized measurement circuit.

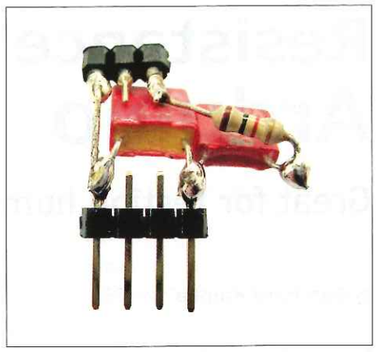


Figure 5. Mini Shield for resistance measurement.

ment. Oh yes, it would be very handy if we could get away with using only one pin of the microcontroller too. The result is the simple measurement circuit using the Port pin PB3 (**Figure 3**).

Rx and C1 form an *R-C* element, whose time constants need to be measured. C2 works in the background like a backup battery that provides the charging current required. All the same, because C2 is also charged via the resistance under measurement, the DC flowing through Rx in the middle is zero. The actual measurement process (**Listing 1**) proceeds in three phases:

1. **Charging.** The Port is connected to V_{CC} via a low resistance, so that C1 charges immediately and C2 more sluggishly.
2. **Discharging.** The Port is connected to GND very briefly, precisely long enough to discharge C1 but short enough to leave C2 retaining almost full voltage.
3. **Measurement.** The Port is configured as a high-impedance input. We then measure the time taken for the input to revert to 1 (High).

The method produces counts that are within broad limits proportional to the resistance. With the input unconnected the test result is limited to 32,768 maximum.
There is still one small problem in that measurement malfunctions with resistances of significantly less than one k-ohm. The reason is evidently that with very small resistances the larger capacitor is discharged immediately during the short discharge pulse, meaning the charging source is now lacking.

Circuit optimization

For this reason it is better to add an extra resistor of 1 kΩ in series, which you can easily subtract later on. Because Rx and C2 are connected in series, you can also change these around (**Figure 4**). This works better because the item under measurement is now connected at one end to ground. The 'no DC' rule still applies, so it's AC measurements only. The same method should be usable for resistive humidity sensors. The small number of components involved can be soldered to a piece of header connector strip (**Figure 5**), for connection to the corresponding Arduino socket strip. This makes our test adapter a kind of Mini Shield. For indicating the value measured we use once more the display on the Elektor Extension Shield [2], on which the Arduino connector strips are duplicated. When you plug the Arduino Uno, the Extension Shield and the test adapter all together, the whole combination looks like our heading photo.

Using linear conversion (**Listing 2**) we can output the resistance in kΩ. The result is not to be sniffed at: between 1 kΩ and 1 MΩ we achieve really good linearity of around 5 %. In the range up to 10 MΩ the variance is a bit larger. In any case, absolute accuracy is also dependant on the tolerances of the capacitors

Listing 1. Measuring the time to charge.

```
Count = 0
Portb.3 = 1
Ddrb.3 = 1          ' Charge
Waitms 500
Portb.3 = 0         ' Discharge
Waitus 10
Ddrb.3 = 0
Do                  'Counter
  Count = Count + 1
Loop Until Pinb.3 = 1 Or
                    Count.15 = 1
Portb.3 = 1
Ddrb.3 = 1
Print Count
Locate 1 , 1
Lcd Count
Lcd "       "
```

Listing 2. Conversion into k-Ohms.

```
'Calculate Resistance
If Count > 4 Then Count = Count - 4 Else Count = 0      ' -1 kOhm
Resistance = Count * 14
Resistance = Resistance / 40
Print Resistance            '...kOhm
Locate 2 , 1
Lcd Resistance
Lcd " kOhm  "
```


and the exact switching threshold of the input. The method is not noted primarily for great accuracy, rather for its broad measurement range and simple circuitry.

Logarithmic measurement

Resistive air humidity sensors possess more or less exponential characteristic curves (**Figure 6**). You need to measure the resistance and then express it logarithmically. For the Arduino this is an easy exercise. The calculation is carried out in several steps (**Listing 3**), in which we produce the natural logarithm in Bascom using the Log function. The following formula delivers the air humidity in percent from the value Count:

$$\text{Air humidity [\%]} = (103 - 8.9 \times \ln(\text{Count}))$$

Errors arise from a certain curvature of the characteristic line in the logarithmic scale. The values in the formula are selected so that the smallest errors occur at 40 % and 80 %. The largest deviation arises from variations among different examples of sensor, however. Calibration is costs money and plenty of simple humidity measurement devices for sale are equally imprecise.

Moreover, temperature dependency is not considered; a room temperature of 20 °C is assumed instead. Nevertheless you can see variations in air humidity very clearly. Unavoidable measurement error is due least of all to any inaccuracy in resistance measurement, since in the logarithmization process these errors merge into virtually nothing.

As these lines are written, it is cold outdoors and warm inside. The air indoors is fairly dry and the sensor (**Figure 7**) is indicating 40 %. That could well be right, according to one of my hygrometers. My flowers urgently need to be watered. Once I do this, the air humidity rises immediately to 41 % and after a while to 42 %. Larger variations arise when you put your hand close to the sensor. With a finger placed either side of the sensor, it shoots up to over 80 % quite rapidly.
By the way, the circuit can of course be used as an ohm meter. All values between 1 kΩ and a few MΩ can be displayed reliably (**Figure 8**). ⏮

(150160)

Listing 3. Conversion into relative air humidity.

```
'Calculate Humidity
F = Count
F = Log(f)
F = F * 8.9
F = 103 - F
Humidity = Round(f)
Locate 1 , 6
Lcd Humidity
Lcd "%      "
```

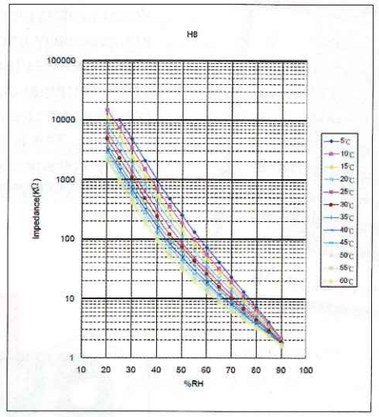


Figure 6. Sensor resistance in relation to humidity and temperature (Source: Datasheet [1]).

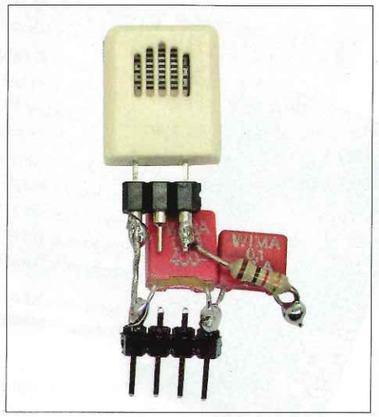


Figure 7. Mini Shield with sensor attached.

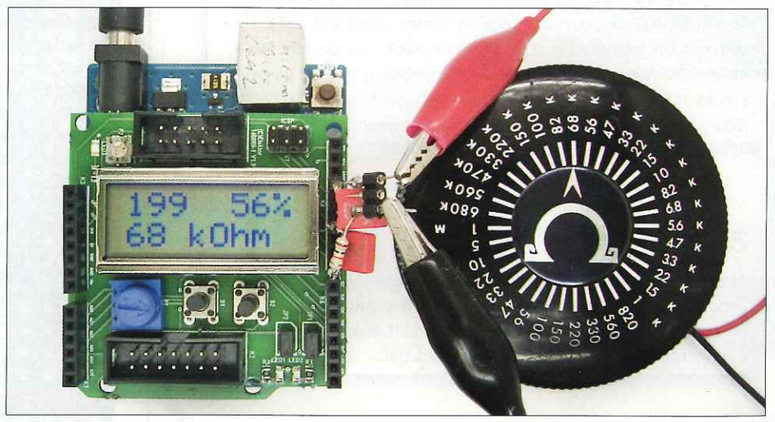


Figure 8. Ohmmeter test.

Web Links

[1] www.farnell.com/datasheets/1355478.pdf
[2] www.elektor-magazine.com/140009

MIDI Analyzer

MIDI In/Out module for Arduino and friends

By **Jens Nickel**

In this article we extend the familiar pairing of the Arduino and the Elektor Extension Shield into a module offering a MIDI (Musical Instrument Digital Interface) input and output. Its Embedded Communication Connector (ECC) allows it to be connected to other microcontroller boards as well. The demonstration firmware decodes MIDI messages and shows them on a display, but the software modules we use lend themselves to a wide range of other applications.

Once again our kitchen website at www.elektor-labs.com proves a valuable source of inspiration: original poster (OP) 'midi-rakete' followed up a project he had had published twenty years ago in Elektor with an updated version, the MIDI Channel Analyzer MkII [1]. In that project a set of sixteen LEDs, one for each channel, was used to indicate when communication was occurring on MIDI channels 1 to 16, for example between a controller keyboard and a synthesizer. (Note that the channels are in fact numbered from 0 to 15 within the MIDI data.) However, the unit does not show what types of MIDI messages are being sent over the wire.

I have personally become interested recently in electronic music production and find it a fascinating hobby, with some similarities to programming. Although I don't get up on stage to perform and so don't often have to worry about connecting various pieces of equipment together, it struck me that a small tool that decodes and displays MIDI messages could nevertheless be very handy. As luck would have it, my colleague Clemens Valens had just designed a small MIDI module for his J²B synthesizer [2]. It consists of a microcontroller with a built-in serial interface, configured to receive and transmit MIDI messages.

The MIDI physical layer

No great 'intelligence' is required in such a device: the MIDI signals are nothing more than data bytes transmitted serially at a fixed data rate of 31250 baud [3]. Unlike, for example, RS-232, the interface does not use defined voltage thresholds for the low and high levels that correspond to individual data bits and start and stop bits. Instead, the interface is based on a 5 mA current loop between the MIDI Out of one device and the MIDI In of another. A current flow represents a logic zero, while the absence of a current represents a logic one. **Figure 1** shows the implementation in more detail. MIDI In and MIDI Out interfaces each require two pins, and two devices are connected together using a two-wire cable. One of the output pins is permanently pulled to +5 V via a resistance of 220 Ω. In order to send a logic zero, the second output pin is taken to ground, also via 220 Ω. The result is a small current flowing through the pins of the MIDI In connector on the other device. To send a logic one, the second output pin is taken to +5 V, and then no current flows. The arrangement is therefore, conveniently enough, compatible with UARTs operating at TTL logic levels.

On the MIDI In side there is an optocoupler which includes a phototransistor. When a current flows in the MIDI cable this transistor pulls the output to ground; in the quiescent state the output swings to +5 V. We can therefore connect this signal directly to the RX input of a microcontroller.

The two connections used for input and output are almost invariably taken to pins 4 and 5 of a five-pin DIN socket on the MIDI device. Externally, therefore, MIDI In and Out connectors look identical, but of course separate sockets must be provided on any device that needs both input and output functions.

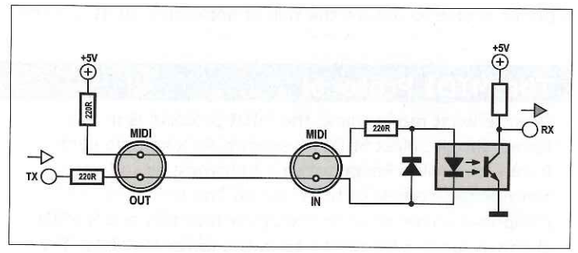


Figure 1. MIDI signals are transmitted using a current loop, where logic zero is represented by a current flowing and a logic one by the absence of a current. RX and TX can be connected directly to a microcontroller with 5 V logic levels.

The hardware

The characteristics described above mean that it is easy to design a circuit to add a MIDI input and output to an existing microcontroller board. It will come as no surprise that I felt that the best choice for connecting the MIDI module to the microcontroller board was to use an Embedded Communication Connector (ECC). So, what we need to do is add an ECC to Clemens' circuit board and then we can, for example, simply connect the MIDI interface board to the proven combination of an Arduino Uno and an Elektor extension shield [4].

The resulting circuit is shown in **Figure 2**. On the left is the MIDI input with a type 6N137 optocoupler. The output of the optocoupler is connected to pin 6 of the ECC (K2), which will in turn be connected to the RX pin on the microcontroller. Pin 5 of the ECC carries the microcontroller's TX signal to the MIDI module, which we use to drive the MIDI output socket (K3)

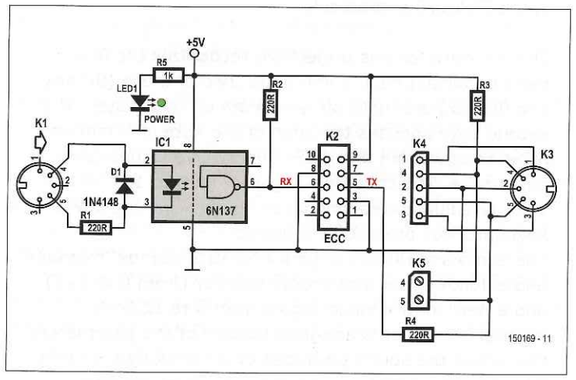


Figure 2. Circuit diagram of the MIDI In/Out module.

Component List

Resistors
R1,R2,R3,R4 = 220Ω
R5 = 1kΩ

Semiconductors
D1 = 1N4148
LED1 = LED, green, 3mm
IC1 = 6N137, DIP8 (incl. socket)

Miscellaneous
K1,K3 = DIN socket, PCB mount, 180°
K2 = 10-way (2x5) boxheader, 0.1" pitch
K4 = 5-way pinheader receptacle, 0.1" pitch
K5 = 2-way screw terminal block, 0.2" pitch
PCB # 150169-1 v1.0

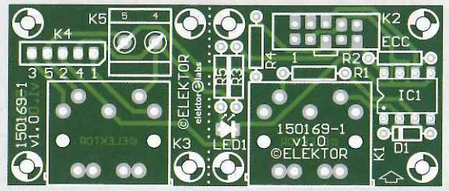


Figure 3. The printed circuit board can easily be cut into two pieces.

directly. The microcontroller board supplies power at +5 V to the circuit via pin 9 of the ECC, and an LED is provided to indicate when power is present.

Ton Giesberts of Elektor Labs has designed a printed circuit board for the module (see **Figure 3**), which is perforated to allow it to be separated easily into two parts [5]. The right-hand part can be used on its own as a simple MIDI input, while the left-hand part can be used as a MIDI output (with MIDI signals supplied at K5) or as a flexible DIN-socket breakout board, with K4 as its input. Our Elektor Labs prototype (**Figure 4**) uses a pinheader for K4, although a header socket would be a better choice to reduce the risk of accidental short-circuits.

The printed circuit board is available from the Elektor Store [6], and populating it should present no difficulties.

Software

The populated MIDI module is connected to the ECC header on the extension shield using a ten-way ribbon cable, and the extension shield is in turn mounted on the Arduino Uno. The ATmega328P on the Arduino Uno can now receive MIDI bytes using its UART, but of course we need some software [6] to make our MIDI analyzer a reality. Fortunately the MIDI protocol is not too complicated (see text box). The most important MIDI messages almost all comprise three bytes, and it is easy to detect the first byte of a three-byte message. The software

The MIDI Protocol

Despite what many think, the MIDI protocol is in fact rather simple. Most MIDI messages, for example sent from a controller keyboard to a hardware or software synthesizer, consist of three bytes. The first byte comprises a command in the upper four bits and a MIDI channel number from 0 to 15 in the lower four bits. The most significant bit of this byte is always set and the following two bytes always have values in the range 0 to 127 and hence have their most significant bit clear. This makes it easy to detect the beginning of a message in the data stream; a similar very simple mechanism was used in the ElektorBus protocol.

The software for this project [6] recognizes the four most important MIDI commands. Two of these (90 hex and 80 hex) are 'note off' and 'note on' messages. The second byte encodes the pitch of the note in semitones as an integer from 0 to 127: thus twelve values cover one octave. The third byte, again from 0 to 127, is a velocity value corresponding to how quickly the key on the keyboard was pressed or released.
The command B0 hex begins a 'control change' message and is followed by a controller number (from 0 to 127) and a new control value (again from 0 to 127). A controller number is assigned to each of the parameters that affect the sound produced by a synthesizer so that they can be controlled over MIDI, even in real time

90_{HEX} + CH	NT	VE	NOTE ON		
80_{HEX} + CH	NT	VE	NOTE OFF	CH CHANNEL 0...15 NT NOTE 0...127 VE VELOCITY 0...127	
$B0_{HEX}$ + CH	CC	CV	CONTROLLER VALUE	CC CONTROLLER 0...127 CV VALUE 0...127	
$E0_{HEX}$ + CH	PL	PH	PITCH BEND	PH PL PITCH	HIGH 0...127 LOW 0...127

during a performance. For example, controller number 1 is reserved for the modulation wheel found on almost all controller keyboards. More details on this can be found at [8].
The command E0 hex begins a 'pitch bend change' message. Fine resolution is required for this function, and so a range from 0 to 16383 (14 bits) is provided for. The second byte of the message carries the least significant seven bits of the value, and the third byte the most significant seven bits.

carries out the following tasks.

1. Initialize the UART and set the data rate to 31250 baud.
2. Store received bytes in a circular buffer under interrupt control.
3. Periodically check the circular buffer; detect a new MIDI message on the basis of its first byte; decode this byte and the two following bytes; store the decoded elements in a dedicated structure; and then call a specified function to indicate that a new MIDI message has been decoded.
4. Show the elements of the newly-received message on the display. A message will remain on the display until a new message is received (for example when the value for one of the MIDI controllers changes), at which point the display will be updated. This makes it easy, for example, to watch the effect of adjusting a controller.

The ideal situation is that we have a separate software module responsible for each of these tasks, in the interests of improving reusability, portability and ease of maintenance. Dependencies (including time-dependencies) between the modules should be kept to a minimum, and we shall return to this issue later. The Embedded Firmware Library (EFL) [7] is a natural choice for our demonstration software, as it already includes an implementation of the circular buffer and a display library, which in turn are based on board and microcontroller files for the extension shield, the Arduino Uno and the ATmega328P. Essentially all that remains is to write a small MIDI library to handle task 3 above, although as we shall see there is rather a lot hidden in that word 'essentially'.

MIDI decoding

The MIDI messages are decoded in a small library called MidiEFL.h/.c. Like other EFL protocol libraries that we have described in Elektor (such as the ElektorBus library), it is designed so that it can work with a range of different physical communication channels, and so in principle it could even be used to decode MIDI messages received over a TCP/IP connection. All that matters is that the bytes to be decoded arrive in a circular buffer, whose location is communicated to the library when it is initialized as follows:

```
MIDI_LibrarySetup(UARTInterface_Send, 0,
   UARTInterface_GetRingbuffer(0), MIDIIn_Process);
```

The first parameter specifies the function that should be called when a MIDI message is to be sent. The second parameter specifies that UART interface 0 is to be used to transmit and receive bytes. (In fact there is only one UART interface on the Arduino Uno.) The third gives the circular receive data buffer, and the last parameter is a pointer to a callback function that must be implemented in the main part of the code. This function will be called by the MIDI library when a new message has been received and decoded.
The main loop of the program must regularly call the library function `MidiProtocol_Engine()`, which handles the work involved in the third of the tasks listed above.

You've got MIDI

As soon as a message has been decoded it is stored in a structure `ReceivedMidiData` of type `MidiData`. The main part of the code can obtain a pointer to this structure using the function `Midi_GetReceivedMidiData()`, and hence can access the decoded elements. In the callback function mentioned above we now have convenient access to all the elements of the most recently received MIDI message, and we can show them on the display. For example, typical code might be as follows:

```
MidiData* ReceivedMidiData =
   MIDI_GetReceivedMidiData();
Display_WriteNumber(0, 0, ReceivedMidiData->Channel);
```

This code will show the channel number of the received message in the first line of the display.

Unfortunately the EFL display library only includes commands for showing text and numbers on a specified line of the display, always starting from the extreme left of that line. The display on the extension shield has only two lines, and we wish to show four elements on it. To resolve this problem the library was extended so that it is possible instead to specify that text and numbers are shown in any one of up to eight possible fields. The fields are numbered from 0 to 7 (see **Figure 5**), and this number is called the field's 'position'. Positions can also be used to select a particular LED within an LED block or a button within a button block.

The output commands for controlling an LED (ON = 1 or

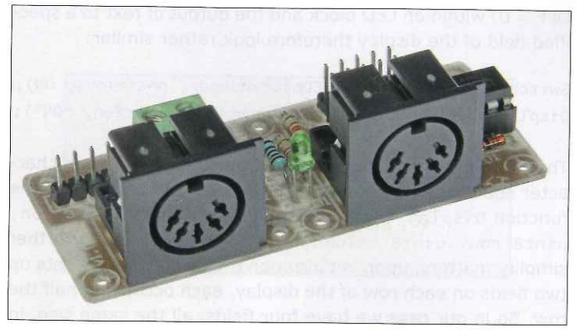


Figure 4. The Elektor Labs prototype fitted with two DIN sockets.

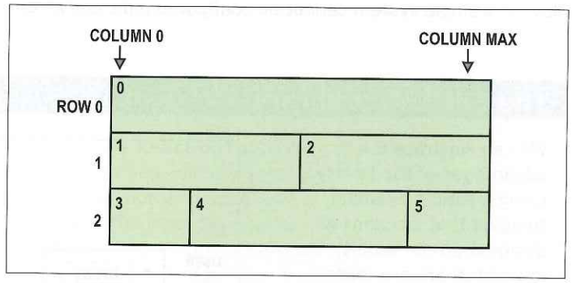


Figure 5. The extended display library allows for the configuration of fields in the display area where text and numbers can be shown. Here a three-line display is shown divided into six fields numbered from 0 to 5.

OFF = 0) within an LED block and the output of text to a specified field of the display therefore look rather similar:

```
SwitchLED              (LEDblocknumber, position,  ON);
Display_WritePosition (displaynumber,  position, "ON");
```

The coordinates and the extents of the fields in terms of character spaces can be specified from the main code using the function `Display_SetPosition(uint8 DisplayPosition, uint8 row, uint8 column, uint8 columnmax)`. To further simplify matters, upon initialization the display library sets up two fields on each row of the display, each occupying half the row. So in our case we have four fields, all the same size, in which we can show the four most important elements of the received message (see **Figure 6**).
To keep storage requirements to a minimum, different displays in a single system cannot be configured differently from one another, and a field is not allowed to occupy more than one row. Of course, the functions can be enhanced to remove these restrictions if required.

Flexible output

If writing to the display is implemented directly inside the function that is called by the MIDI library when it has decoded a new message, we have voluntarily created a close coupling between the two modules, in particular from a timing point of view, and this is not desirable. A better idea is not to update the display immediately. It is possible that further extensions to the software will include tasks that require higher priority: for example we might wish to log incoming MIDI bytes with timestamps, or perhaps at some point we might create a mini-synthesizer to turn incoming note messages into sounds. The solution to this problem involves making an entry in a special table (called 'StateTable') for each of the different MIDI elements to be displayed. Alongside the current value of each element we maintain a flag that indicates whether the value has changed since we last refreshed the display. Inside the function that is called after each MIDI message is decoded we write the new values of the MIDI elements into the StateTable. If the value does not agree with the value previously stored there, the 'STATE_UPDATED' flag is set.

We must now periodically check whether the display needs refreshing with updated values. Note that how often this is done is now independent of the rate of arrival of MIDI messages. Refreshing the display in response to changes in values is done with a call to the function `Reaction_Process()` in the main program code, implemented in a new EFL module called InOutEFL.c. The function inspects the table entries to see which have their flag set, and then for each calls the required output function. The desired output functions are given in a table called

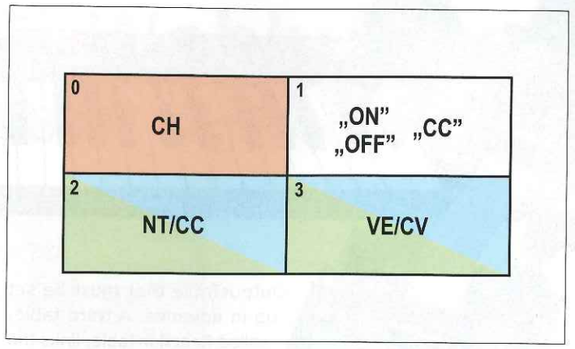
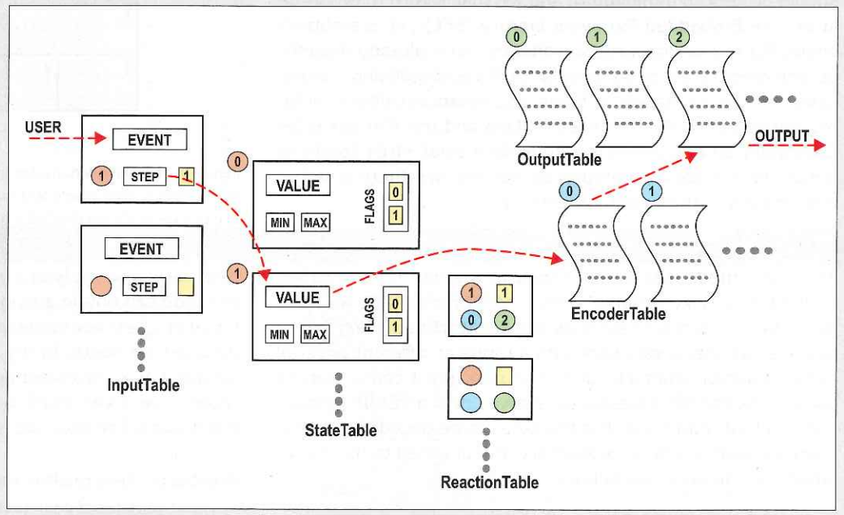


Figure 6. The various elements of a received MIDI message are shown in four fields on the display: CH=channel; NT=note (as a text string); VE=key velocity; CC=controller number; CV=controller value.

The EFL InOut library

We can illustrate the advantages of the library using a simple example. Imagine that we want to design a power supply, in which there are four application variables, or state values, called U_setpoint, U_actual, I_actual and I_max. There are input controls (such as buttons, rotary encoders or potentiometers) that adjust the values of U_setpoint and I_max, and the values of U_setpoint, U_actual and so on are to be output, for example to a display. We would like first of all to be able to run the software on different boards with a minimum of adaptation. For example one board might offer a potentiometer for setting U_setpoint while another might offer 'up' and 'down' buttons. Of course some changes will be needed to the code when we port it, but the idea is to make it as simple as possible by isolating the hardware-dependent parts of the program in the `ApplicationSetup()` function.
A second objective is to decouple from one another, both from a software perspective and a temporal perspective, the processes involved in input, changing of state values, and output. If readings of a value are taken at the rate of a thousand per second it does not make sense to update the display with each one as it arrives.
The third objective is that we would like to minimize the programming effort involved. User inputs that involve setting values always need to be validated against upper and lower limits, and the new framework should spare us the drudgery of coding line after line of 'if' statements.



The new EFL common library InOutEFL.h/.c maintains several tables which must be initialized at the beginning of the program. The functions `State_Add()`, `Output_Add()`, and so on are used to generate a new entry in the corresponding table; they return the index of the newly-added entry, which can then be used as a parameter when creating a new entry in another table. This approach can be compared to the wiring of a circuit board: adding table entries is like adding wires to create connections between inputs, state changes, and outputs.

At the heart of the architecture is the **StateTable**. Each entry in this table gives the current (16-bit) value of an application variable, its minimum and maximum permitted values, and up to eight flags. The **InputTable** links input events (such as the pressing of a particular button specified by the button block number and its position within the block) with an index in the StateTable. Alongside this is a step size by which the value of the variable is to be adjusted when the event occurs, and the flag that is to be set when the event occurs.
In the InputTable it is therefore possible to specify, for example, that when a particular button is pressed the variable U_setpoint (which might represent a target voltage in millivolts) is to be increased by 100. When the value is increased it is checked to verify that it lies within the preset limit values. If the upper limit is exceeded the value will either be clamped to the upper limit value or reset to the lower limit value, depending on the configuration. The latter alternative allows values to be adjusted in a 'wrap-around' fashion. If the value changes as a result of this process, a specified flag is set: in this case we would choose the STATE_UPDATED flag.
The **OutputTable** is the counterpart to the InputTable. It stores the output functions (which might be implemented for example in the display library) along with the block number and position of the output element. Any function can be used here as long as it has the signature:

```
functionname(uint8 blocknumber, uint8 position, int16
   numericalvalue)
```

or:

```
functionname(uint8 blocknumber, uint8 position, char*
   textstring)
```

The **EncoderTable** stores encoder functions that convert numerical values into text strings. They can be implemented in any of the various EFL modules, but must have the signature:

```
char* functionname(int16 numericalvalue)
```

An entry in the **ReactionTable** links an index to the StateTable (or equivalently an application variable), a flag, an output function and optionally an encoder, all stored as indices to the respective tables. To ensure that output functions are triggered it is necessary to call the function `Reaction_Process()` regularly. This function inspects the entries in the ReactionTable and for each checks whether the flag specified to trigger the reaction that accompanies the specified application variable in the StateTable is set. If it is, the output function is called. Normally this function will either output the current value of the application variable or the result of passing it through the encoder function. Finally, the flag in the StateTable is cleared.

Continuing with our example, we could define an entry in the OutputTable to output text at position 0 of display 0. Also, we could implement an encoder function to convert a value in millivolts to a text string with the format 'x.yyy V'. This function will be added as a new entry in the EncoderTable.
A new entry in the ReactionTable can now be used to link U_setpoint, the STATE_UPDATED flag, and the indices of the entries we have just generated in the OutputTable and the EncoderTable.
Now, if we press the button and increase U_setpoint by 100 mV, upon the next call to `Reaction_Process()` the encoder function will be called and then the text will be written to the display. Altering the code to use the Continental European-style comma instead of the decimal point is easy: we simply write a slightly modified version of the encoder function, add it to the EncoderTable, and then change the corresponding index in the ReactionTable. This can be done even while the program is running, for example if we wish to allow the user to change the language of the user interface.

There will be more on the EFL InOut library in a future edition of Elektor.

OutputTable that must be set up in advance. A third table, called ReactionTable, links the two others, ensuring that there is an output function associated with each value that might change. It is also possible to provide an index into a fourth table, called EncoderTable. An example will help explain the interrelationships between these tables, and we will now look at how a received MIDI note value is processed.

At the beginning of the program we set up some table entries as follows.

```
S_MidiIn_Note  = State_Add(0, 0, 127,
   STATE_MINMAXMODE_OVERFLOW);
O_Write_Pos2   = Output_Add(Display_WritePosition, 0,
   2);
E_Midi_Note    = Encoder_Add(Midi_NoteEncode);

Reaction_AddOutput(S_MidiIn_Note, STATE_UPDATED, O_
   Write_Pos2, E_Midi_Note);
```

When a new note value (which must be in the range from 0 to 127) is received it must be written to the StateTable and the STATE_UPDATED flag must be set. This is implemented in the callback function as follows.

```
MidiData* ReceivedMidiData =
   Midi_GetReceivedMidiData();
State_Update(S_MidiIn_Note, ReceivedMidiData->Note);
```

The function `Reaction_Process()` is periodically called in the main loop, and the function checks whether the flag is set. If it is, then the encoder function that has been configured to handle the value is called: in this case the encoder function is `Midi_NoteEncode()`, which is implemented in the EFL MIDI library. This function takes the note value from 0 to 127 as a parameter and converts it into a string such as 'C#4'. This string is in turn passed as a parameter to the configured output function `Display_WritePosition()`, which writes the text to position 2 on display 0.

The behaviour of the software can be modified simply by changing table entries, even dynamically while it is running. For example, the user interface language could be changed simply by changing the table entry for the encoder function.

Hardware independence

We have brought the degree of hardware independence and modularity of EFL applications to a new level. At the start of the program in the `ApplicationSetup()` function all we need to do is suitably initialize our tables, and then everything will be taken care of and the various modules are kept decoupled from one another. We can easily port the firmware to another board, where for example we might want to use display 1 rather than display 0 to show the MIDI data, or we might want to change the positions where the various elements are displayed; another possibility might be to output the decoded MIDI elements over a (different) UART instead of showing them on a display; and yet another might be to visualize keyboard velocity information using the brightness of an LED. All these can be implemented with simple modifications to the setup code, leaving the rest of the program unchanged.

In a future article we will expand this project to include user input. The **text box** 'The InOut Library' gives an outline of the idea. Watch this space!

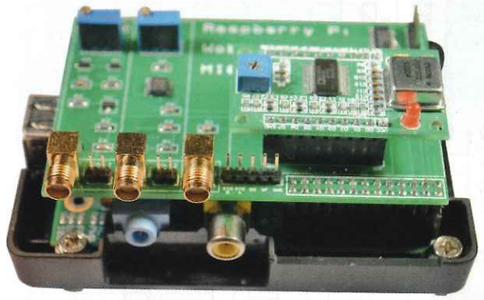
(150169)

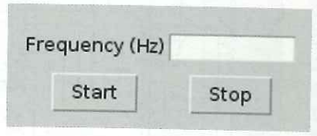
Internet Links

[1] www.elektor-labs.com/project/midi-channel-analyzer-mkii-140065-i.13380.html
[2] www.elektor-magazine.com/140182
[3] http://en.wikipedia.org/wiki/MIDI
[4] www.elektor-magazine.com/140009
[5] www.elektor-labs.com/project/150169-midi-analyzer-light.14481.html
[6] www.elektor-magazine.com/150169
[7] www.elektor-magazine.com/120668
[8] www.midi.org/techspecs/midimessages.php

A Raspberry Pi Wobbulator

With AD9850 DDS and AD8307 Log. Detector





By **Tom Herbison,** MIOIOU (United Kingdom) Twitter: @TomHerbison

The Raspberry Pi computerette harnesses so much connectivity it's just crying out for extension circuits to make it do unusual things. This Reader's Project covers the evolution of an RPi and a purpose designed extension circuit into a DIY RF sweep generator, or 'wobbulator'. The project started out at elektor-labs.com and went through the full Learn, Design, Share cycle.

As you can see in **Figure 1**, a wobbulator (or 'sweep generator') is a piece of test equipment which is used in conjunction with an oscilloscope to measure the frequency response characteristics of an (RF) circuit. It uses a "ramp" or "sawtooth" function generator connected to a voltage controlled oscillator (VCO) to produce an output sweep over a defined frequency range. The response characteristics of the circuit under test — usually a filter or an amplifier — can then be displayed on an oscilloscope. A wobbulator is a useful tool for aligning the intermediate frequency (IF) stages of superheterodyne receivers, but can also be used to measure the frequency response characteristics of RF filters and other circuits.

One of the great things about a wobbulator is its direct presentation of the filter or amplifier response curve, allowing you to see specifications like the 3-dB roll-off points, slope steepness, and in-band ripple, "live" on the screen as you tweak the circuit under test. Multi-stage RF passband and notch filters especially are a joy to 'wobble' into shape with instant visual feedback on their response. It's like drawing the curve in small steps and can keep you busy for hours to get the textbook shape.

The Plan

While in the dim past a wobbulator was a complex, expensive all-analog instrument (even with vacuum tubes), today we have little computers like the RPi to do the job with simpler hardware when it comes to control, and with the luxury of software, which can be changed and optimized for best results.

The Raspberry Pi Wobbulator implements the functionality of a conventional wobbulator by using a Raspberry Pi computer, a Direct Digital Synthesizer (DDS) module and an Analog to Digital Converter (ADC) module. The Raspberry Pi's General Purpose Input Output (GPIO) interface is programmed to control the DDS module to generate the frequency sweep and to communicate with the ADC module to measure the response of the circuit under test. The Graphical User Interface (GUI) allows the user to choose the parameters for the frequency sweep and also displays the results.

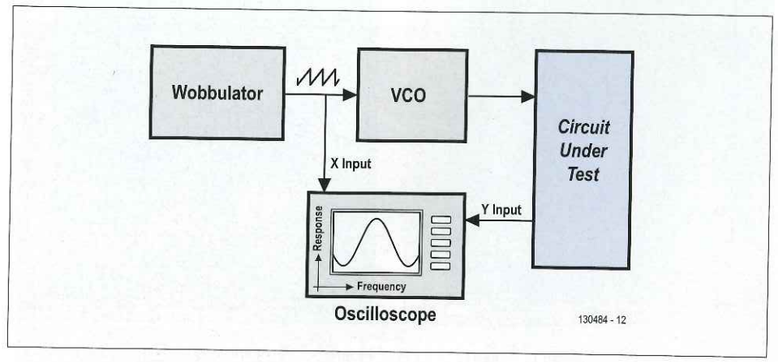


Figure 1. Basic operation of a wobbulator, also known as a sweep frequency generator.

The Circuit

Figure 2 shows the latest version of the RPi hardware extensions that together form the wobbulator.

Initially the wobbulator had a single input amplifier/buffer followed by a rectifier to display the linear response of the circuit under test. As the project evolved this circuit was extended with a logarithmic amplifier for a dBm readout on the Y scale. The LIN amplifier was retained though, here it is seen around FET U1, with diodes D1 and D2 doing the RF rectification.