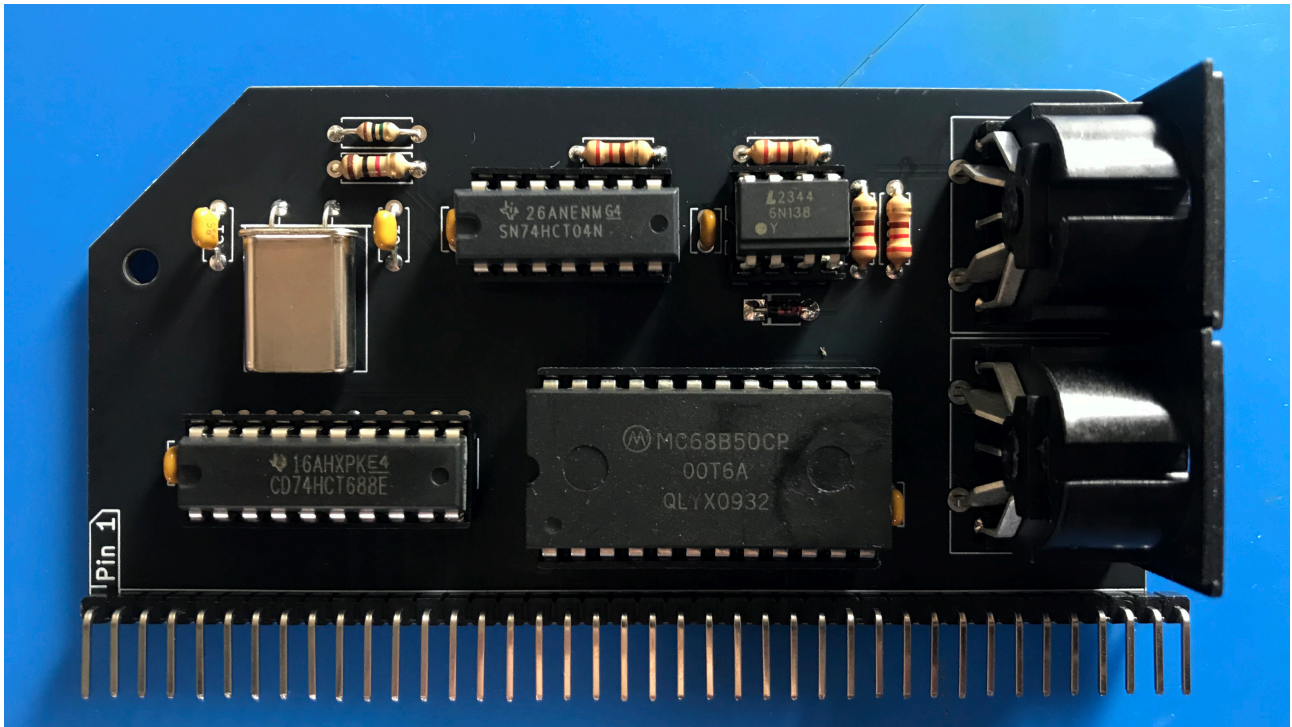


MIDI

Module for RC2014 R2



<https://peacockmedia.software/RC2014/MIDI>



Description

This module is an improvement on the mk1 MIDI module in that it is self-contained. It not only has its own clock but its own serial chip and so it'll work on any RC2014 or RCBus system and is designed to be 'plug and play' - all the user has to do is plug it in, send and read data to and from its hardware ports.

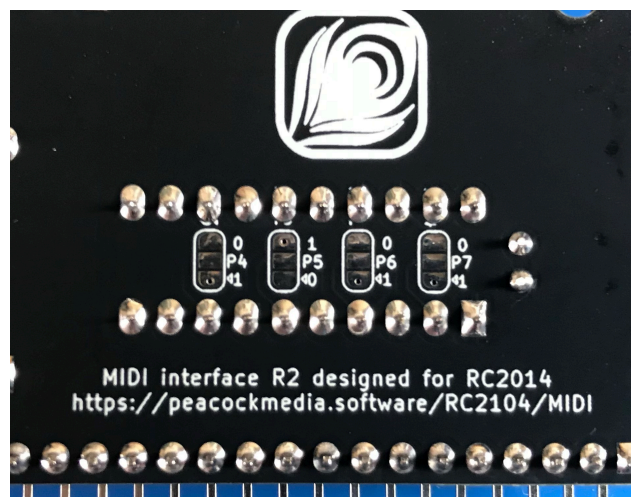
Ports

By default, this module uses ports \$DE and \$DF.

I'd like to see a standard emerge for MIDI on RC2014 and I'll revise my board if necessary in future. I'll be even happier if DE and DF becomes that standard now that I've used those ports.

If DE/DF isn't suitable for you, then the board has these 'cut and shut' solder pads. These set the high nibble for the ports. The low nibble is always E and F.

These pads are labelled P4 - P7. These correspond with the bits in the high nibble. The pads are marked 0 and 1 (NB 0 isn't consistently at the top. Also you should read them R-L, ie P7, the MSB is at the right). A little arrow points to the default. So you can see that the default is 1101 or D in hex. To make the ports AE and AF, you need to reverse P4, P5 and P6. Take a sharp knife, cut between the centre pad and the lower one, then make a solder bridge between the centre and top one.



Software

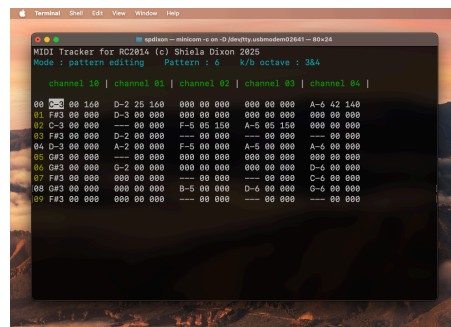
As you may have seen, I've been working on tracker software that makes use of this MIDI module for note input as well as for playing your musical creations.

A video in which I explain how this works is here:

<https://youtu.be/kKTy2W6Xdyg>

A playthrough of the demo music is here

<https://youtu.be/QJSEACHs0J0>



I developed this on a 32k machine (Classic or Mini RC2014 with ACIA i/o). I then built a version that runs on CP/M machines - classic and mini with CP/M upgrade, 'Pro' machines and ROMWBW machines. The different versions of the software, manual and demo tune are all in this zip file:

<https://peacockmedia.software/RC2014/MIDI/MIDITracker.zip>

I've made a 3-track version that sends its output to your AY/YM module (port configuration is within the software). You can still optionally use a MIDI controller (eg a MIDI keyboard) to help you input notes.

<https://peacockmedia.software/RC2014/MIDI/AYTracker.zip>

In both cases, these applications have versions for classic (32k) machines, CPM and ROMWBW. The 32k version is nice and snappy but requires saving / loading of your work via the terminal and doesn't have as much ram for your work, the ROMWBW version is clunky because of system overhead. The plain CP/M version is my favourite because of the convenience of loading/saving to your storage device and still runs smoothly. In all cases there's a demo tune that you can load, plain-text documentation and in-app reference via the [?] key.

Development framework

You'll also find a framework in assembly and C with some example applications which you can use to develop your own software.

Obtain all of this from:

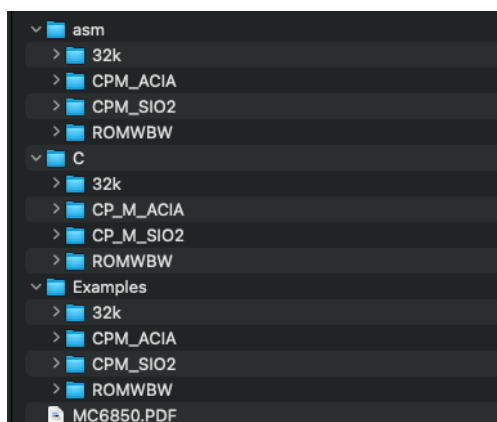
<https://peacockmedia.software/RC2014/MIDI/MIDI-Mk2.zip>

Currently, these systems are supported:

- 32k (Classic, Mini and Micro with 32k ram starting at \$8000, ACIA serial and SCM OS)
- CP/M (Pro machines running CP/M with SIO2 serial)
- CP/M (machines running CP/M with ACIA serial, eg CP/M-upgraded Mini or Classic II)
- ROMWBW (booted into CP/M or Z-system with SIO2 serial. Note that the CPM software won't work here, even if booted into CP/M)

Examples

At the top level you'll find a folder called Examples, which contains the built examples ready to run.



MIDI-MON is a simple monitor for testing. This is what you get if you simply build the framework as is. It'll send a note on / note off and then display incoming MIDI messages

AYPiano will play notes on an AY/YM chip (at the traditional D8/D0 ports, but configurable in the source) in response to incoming MIDI.

PERCSEQ is a simple 32-bit sequencer with a terminal interface that allows you to place notes with velocity for bass drum, snare and hi-hat, sending to channel 10.
<https://youtube.com/watch?v=YtQmgOX-hel>

Software framework

First of all you'll find folders for asm and C. Within those are folders for each system and it's important to choose the correct one because there are important differences.

Assembly

The important files are MIDI.asm which contains the library, and MIDI-fw.asm which is your template. All you need to do is to duplicate MIDI-fw.asm and give it a suitable name for your project.

There are two important labels:

Above main_loop: you can do your initialising Within main_loop there are a few important calls and you can add your own program.

midi_message_received: will be called when there's a new MIDI message to deal with. By default, the framework prints this to the console, but this is costly time-wise and you probably only want to leave that in while testing or debugging.

There are some useful routines that you can call, such as send_midi_message: and you'll find an example of this just before main_loop:

C

The important files are midi.c / midi.h which contains the library, and midi-fw.c which is your template. All you need to do is duplicate midi-fw.c and give it a suitable name for your project.

Within main(), you'll be able to do your setting up and you'll find a main loop for your program.

There are some useful routines - see midi.h - such as send_note_on() and send_note_off(). By default, midi-fw.c has an example of sending a note on/off just before the main loop.

Some technical details

This module uses a 68B50. This is capable of storing a single byte of incoming data at a time, and optionally sending an interrupt signal.

Ideally, we want to set up an interrupt service routine which will check whether there's a new byte ready, and if so, buffer it. (A further routine, which the user calls, called MIDI_task will check our buffer and compile the MIDI messages). Because of the way the 6850 sends an interrupt (just a simple signal), interrupt mode 1 is necessary on the Z80. This triggers a 'rst \$38' or jumps to that address, which is usually a jump to the actual ISR.

This means that a lot of the functionality will be different according to your machine:

On a **classic 32k** machine running SCM with ACIA, we can't change the vector at \$38 because that's ROM. Helpfully, SCM puts a jump table in high RAM, which the jump at \$38 points to. SCM even provides a routine for inserting our own routine. There's no problem with switching interrupts to mode 1 (SCM doesn't have interrupts enabled by default).

On **CP/M**, \$38 is in RAM, so we can insert the address of our routine there. But it is already using interrupts (mode 2 I believe) for the SIO serial. This means that we also have to reconfigure the SIO so that it's not using redirects, and we have to poll that ourselves for key input. Once we've injected the address of our interrupt routine in zero page, then we can force interrupt mode 1

ROMWBW, even when running CP/M, doesn't allow us to simply change the vector at \$38 (because it switches banks of memory in the lower 32k.) It does provide API calls for us to register a routine, so it may be possible to interrupt-enable the MIDI module for incoming data, but I haven't made this work yet. In the mean time, the framework for this system polls the 6850 (this must be done very frequently to avoid dropping bytes.)

This approach has some advantages. Since it doesn't mess with interrupts and the system's i/o, programs will run on all cp/m systems and won't care what serial chip you use. I used this framework for the cp/m version of the MIDITracker since the computer keyboard entry is more important in that case than the MIDI in (which still works fine).

An alternative method

Note that I developed my tracker (for CPM and ROMWBW systems) using the 'ROMWBW' framework. Since MIDI input is less important than computer keyboard input, I chose that framework because it doesn't use interrupts for the MIDI input and leaves the system i/o intact. That means that you have to poll the MIDI in by calling `MIDI_serial_task()`; yourself (which polls the MIDI module's 6850 and buffers the data if there is any) before you call `MIDI_task()`; which collects any data from the buffer.

The disadvantage of doing that rather than setting up an interrupt is that you have to do that polling frequently enough not to miss incoming MIDI data. The advantage is that you're not changing the system's i/o, meaning that the same software will work on any of the CPM-like systems regardless of serial chip.