USB-MIDI

v0.08

2023/03/18

USB MIDI for RISC OS 5

by

Rick Murray
&
Dave Higton

rick -*at*- heyrick -*dot*- eu

Introduction

This document describes the USB-MIDI module, version 0.08. The module itself is called "**MIDI**" and offers the same SWIs as the old Acorn MIDI module; it is intended to be a (mostly) drop-in replacement. The difference, however, is that this module works with MIDI devices attached via USB; plus it is intended to work on the newer range of RISC OS machines such as the *Beagle, RaspberryPi, ARMX6, Titanium...*

Differences

If you are used to the Acorn MIDI module (from 1989), then please note the following differences between that module and this implementation:

- There is **no** MIDI interpreter/player, nor is one planned. As a consequence of this:
  - The SWIs and commands related to the interpreter are dummies and do nothing.
  - The MIDI module does not support timing in the same way as the original module, specifically timing using BARS and BEATS.

- USB-MIDI data is sent as distinct packets. Therefore it is not possible to transmit data with a running status. For instance, on original MIDI you could do the following:
  `<note on><note><velocity><note><velocity><note><velocity>`
  (note there is only one "note on"; the rest are *assumed*)
  When sending this sort of sequence via the MIDI module, the first note will be sent and all of the subsequent notes will be discarded as invalid data.
  Additionally, data may not be transmitted immediately if using TxByte. It will be buffered until a complete MIDI command is available, and then it will be sent.

- USB devices are hot-pluggable. This means they may be arbitrarily connected and disconnected. For this reason, please note:
  - The presence of the MIDI module does *not* imply MIDI hardware. You must explicitly ask how many MIDI interfaces are present.
  - It is valid for there to be ***zero*** interfaces, and *for this value* ***to change at any time***.
  - Likewise, a MIDI interface *in use* may disappear at any time. In this situation, look at the error byte (SWI `MIDI_InqError`), it will be '**X**' if the interface has vanished.

- The receive buffer is 2048 bytes, the transmit buffer is 3 bytes (yes, *three* ... when a valid packet has been formed, it will be output immediately). The schedule buffer is 8192 bytes, but note that in addition to MIDI data, time values are also stored.
  These buffers are per-port, and the values are fixed and cannot be changed.

- MIDI data is timestamped. If a high resolution timer is available, then it can be stamped with millisecond accuracy in FastClock mode. If the timer is not available, then this behaviour will be faked by multiplying the system 100Hz ticker by ten.

- The use of the schedular requires both a high resolution timer *and* the module to be running in Fast Clock mode. Refer to `MIDI_FastClock` on page 19.

Limitations

There are some limitations in this version of the MIDI module. Please read this *carefully!*

- There is no built-in MIDI interpreter.
  Resolution:
  None. There are no plans to implement a MIDI interpreter.

- The SWI to provide direct access to the module's innards does not work, it returns an error!
  Resolution:
  None. This module is completely different to the Acorn module (it is written in C for a start!) and as such there is no mechanism to jump into the innards of the module, ***nor will there ever be such a thing****.*

- Commands that are not part of the MIDI 1.0 standard cannot be handled.
  Resolution:
  None. Please get in touch if you'd like to discuss ideas.

- The module receives, but does not transmit, Active Sensing messages.
  Resolution:
  None. Please get in touch if you'd like to discuss ideas.

- Using TxByte to send notes with running status (in other words, one note on command followed by multiple notes) is not supported.
  Resolution:
  Use TxCommand to send notes individually. Please get in touch if you'd like to discuss ideas.

- SysEx commands are unlikely to be correctly supported (it's untested).
  Resolution:
  None. Please get in touch if you'd like to discuss ideas.

- The module does not immediately transmit bytes that have been sent to it.
  Resolution:
  None. MIDI over USB sends commands as a frame of up to four bytes, so the module will buffer data (if sent byte by byte) until a valid command has been received, and that will then be sent.

General compatibility

My aim is, as much as possible, to provide a module that is a drop-in replacement for the original Acorn MIDI module that grants access to modern USB-based MIDI devices; with a simple and clear API that is sufficiently documented that you do not have to spend an eternity trying to work out what is supposed to happen and when.

In terms of hardware, this module has been developed on a *RaspberryPi (256MiB Model B rev. 2 type 1233)* with a *Yamaha PSR e-333* keyboard and a generic *USB to MIDI interface* connected to a *Roland E-16*.

In terms of software, this module has been tested with my own programs (mostly written in BASIC, some are supplied with the module) and the off-the-shelf version of *!Maestro* that comes with RISC OS 5, as well as the recommended software *Rhapsody4* that you can download from:

`https://jeanmichelb.riscos.fr/Rhapsody4.html`

Introducing MIDI

MIDI means *Musical Instrument Digital Interface*. Originally specified as a serial protocol (akin to connecting a modem), MIDI is a means of connecting electronic instruments (synthesisers and keyboards, etc) not only to each other, but also to computers and sequencers so music can be "programmed" and played at will. In the reverse sense, it is also possible for a keyboard to be used for music input, and there are programs that are able to convert music played on a keyboard into notation.

While serial MIDI devices exist, they are increasingly rare as USB has become a ubiquitous protocol. Many modern keyboards offer USB connectivity. For older keyboards, it may be necessary to purchase a USB to serial MIDI convertor. Note, however, that the ones you can purchase for a tenner on eBay, Amazon, etc tend to be *terrible*, more details below.

Internally, each MIDI instrument has a transmitter ad a receiver (this is true for serial *and* USB alike). These communicate with other MIDI devices using a standardised code (known as "*General MIDI*") which sends informations such as which notes to play, how hard the keys are being pressed, whether or not the sustain pedal is in use, and all sorts of other messages, such as which "program" (voice) to select – should it sound like a grand piano or a tubular bell?

With a MIDI interface connected to your RISC OS computer, you can use the computer to control your musical instruments, either with software you write yoursnelf or third-party software. The RISC OS music scene is not particularly active these days, however *Rhapsody4* is available for free and it is really rather good.

A note of warning regarding cheap USB MIDI interfaces

You can purchase inexpensive USB to MIDI interfaces on eBay, Amazon, etc. While these *work*, it should be pointed out that they work *to a degree*. The one that is typically available is only able to cope with five notes being received at the same time. I do not know if there is a quirk in RISC OS's handling of USB data or if the device doesn't bother to report back "hang on, buffer full", but what I do know is that too much data at once will cause data to be lost.

This issue is discussed in more detail at:

`http://www.heyrick.co.uk/blog/index.php?diary=20141122`

By default, no delay is used, however you can specifically enable a delay on a per-port basis using the MIDI_Options SWI described on page 22.

However, if you are serious about music, you'll simply want to avoid cheap interfaces. By handling only around five notes at a time, that's a chord and two notes of melody on *one* instrument. They're cheap for a reason...

Connecting it up

In the old days, there were three MIDI ports (IN, THRU, OUT) which had specific purposes. Now it's just a standard USB cable.

If you have a serial keyboard with an adaptor, push the round plug marked IN into the keyboard's IN socket, ditto for the OUT plug and socket, and then insert the USB plug into your computer.

If you have a USB keyboard, then just hook the cable up as appropriate. The computer uses the usual flat type A, the keyboard will probably use the chunky square type B.

That's all.

To test your hardware, with the keyboard/instrument connected to your computer and the MIDI module loaded, go to the command line (press F12) and type:

```
*MIDIUSBSend 144 60 80
```

You should hear a Middle C being played. If your instrument does not stop playing the note, type:

```
*MIDIUSBSend 144 60 0
```

If you don't hear anything, enter:

```
*USBDevices
```

and verify that the device is present and is being recognised. For my cheap interface, it appears like this:

```
15   1   7   0/ 0 QinHeng Electronics USB2.0-MIDI
```

Then enter:

```
*MidiUSBInfo
```

and the last few lines of the output should say something like:

```
Information for port 0:
Current MIDI device is USB15, VID 1A86, PID 752D,
using IN file handle 254,
and OUT file handle 244.
```

This will tell you if and what is recognising the device.

MIDI timestamp

The default MIDI timestamp is the MIDI beat counter, which increments around 20 times per second (for ~98BPM).
Using the `MIDI_FastClock` SWI, the module will switch to a millisecond timestamp if your machine has an available high resolution counter. If not, it will use a fake millisecond timestamp by using the current system ticker (the 100Hz tick).

Setting bit 0 of R0 to `MIDI_Options`, you can choose to use the faked timestamps if you wish.

Programming MIDI

The MIDI module provides a number of SWI calls intended to make programming MIDI as flexible as possible.

You may know that SWI calls correspond to the BASIC command `SYS`. So let's quickly recap what `SYS` looks like:

```
SYS "<swi name>", r0%, r1%, r2%, r3%, r4% TO r0%, r1%, r2%, r3%, r4% ;  flags%
```
command to call | registers values to send | register values to receive | CPU flags

Pretty much all of this, except the SWI to call, is optional. Assume you have a SWI to add two numbers, passed in R0 and R1 and the result is returned in R2. You would call this as follows:

```
SYS "AddStuff", firstnum%, secondnum% TO ,,result%
```
(made up name!) — note the two commas here

To put this into more useful practice, the SWI "`MIDI_TxCommand`" takes the following parameters:

R0, byte 0, is the MIDI command
R0, byte 1, is the first data byte (if required)
R0, byte 2, is the second data byte (if required)
R0, bits 24-25, are optional (*and not currently supported*)
R0, bits 28-31, are the port (MIDI device) number
R1 is the time in which to send the note, or zero to send it immediately.

So if you recall from the previous page, the MIDI command to broadcast a middle C is 144,60,80. Therefore we would construct our data word, for port 0, as follows:

```
data% = (80 << 16) OR (60 << 8) OR 144
```

Then we will pass this as a SWI call:

```
SYS "MIDI_TxCommand", data%, 0
```

You should have heard a middle C sound on your instrument.

There's another way to do this. The "`MIDI_TxNoteOn`" SWI sends the command to play a note. It takes the following parameters:

R0 is the note to play (60 is middle C; each number up/down is one semitone)
R1 is the velocity of the key (0 is silent (this usually *stops* a note playing), 127 is extremely hard)

`Which means that:`

```
SYS "MIDI_TxNoteOn", 60, 80
```

would have also played a middle C.

To read data, we will look at the "`MIDI_Init`" SWI. This scans for MIDI devices, registers them, and then returns the number of devices found in R0, like this:

```
SYS "MIDI_Init", 0, 0 TO devices%
PRINT "You have "+STR$(devices%)+" MIDI devices connected."
```

Please refer to the BBC BASIC Manual, or Google, for further details on the `SYS` command, if necessary.

SWI commands

SWI commands are listed numerically. There are many. *Don't panic!*
(well, maybe panic a little, but do it quietly or you'll scare the sheep)

MIDI_SoundEnable (&404C0)

This SWI is recognised, but it has no effect (no MIDI interpreter).

On entry –

On entry – (preserved)

MIDI_SetMode (&404C1)

This SWI is recognised, but it has no effect. Replies with dummy data.

On entry –

On exit R0 = 1
R1 = byte 0 is 1, byte 1 is 1.

MIDI_SetTxChannel (&404C2)

Specifies the port (device) and MIDI channel for all subsequent *MIDI_Tx.....* commands.
This is all of the transmission commands *except* for MIDI_TxByte and MIDI_TxCommand.

On entry R0 = Channel number (1-64) or 0 to read.
1-16 means channels 1-16 of device 0
17-32 means channels -16 of device 1
33-48 means channels 1-16 of device 2
49-64 means channels 1-16 of device 3

On exit R0 = New (or current, if reading) channel number

If the chosen channel points to a device that is not present, then *no change will be made*.

MIDI_SetTxActiveSensing (&404C3)

Transmission of Active Sensing is not currently supported, however you can use this SWI in order to determine if Active Sensing messages are being received on a given port.

On entry R0 = Bit 0 = Disable sending Active Sensing messages
Bit 1 = Enable sending Active Sensing messages (*not currently implemented*)

On exit R0 = Bits 0-3 correspond to MIDI ports 0-3 if Active Sensing is being sent.
Bits 4-7 correspond to MIDI ports 0-3 if Active Sensing is being received.

MIDI_InqSongPositionPointer (&404C4)

This command returns the current MIDI clock divided by six, along with information on the MIDI timing.

On entry –

On exit R0 = 0 (position)
R1 = Bit 0 = Set if Internal Timing is in use (Fast Clock mode)
Bit 1 = Set if External Timing is in use (not Fast Clock, and after *MIDI_Start*)
Bit 2 = Set if in Fast Clock mode
Bit 3 = Also set if in Fast Clock mode (originally "Version 3 facilities")
Bit 4 = Set if System Real Time Messages are put in the receive buffer
Bit 5 = Set if System Real Time Messages are *not* acted upon.

If Fast Clock is not in use and no MIDI_Start has happened, it is possible for bits 0 and 1 (timing source) to both be zero.

MIDI_InqBufferSize (&404C5)

Despite what the name may imply, this call returns the number of *unused* bytes in the receive or transmit buffer.

The complete buffer sizes are:
Receive buffer – 1024 bytes
Transmit buffer – 3 bytes

On entry R0 = 0 to read the receive buffer size
1 to read the transmit buffer size
and bits 1-2 specify the MIDI device (0-3)

On exit R0 = Number of bytes *free* in the selected buffer.

MIDI_InqError (&404C6)

Returns the value of the MIDI error flag, byte 0 for device 0, byte 1 for device 1, etc. Only the most recent error state is given, previous errors will have been overwritten.
Reading the error byte clears the error condition.

On entry –

On exit R0 = Four bytes, one per device, giving the current error code for each device.

Possible values of the error byte are:

`0`	(0)	There is no current error to report.
"`A`"	(65)	Active Sensing is no longer being received.
"`B`"	(66)	Receive buffer is full, and data has been lost.
"`D`"	(68)	Transmit data has been discarded due to unrecognised command/data.
"`X`"	(88)	USB device has been disconnected.
"`/`"	(???)	USB device not present.

The errors "`F`", "`L`", "`O`", "`T`", and "`V`", are not supported (UART/interpreter errors that are not applicable in this implementation), while "`D`", "`X`" and "`/`" are our own (new) error codes.

MIDI_RxByte (&404C7)

Returns the next received MIDI byte. It is recommended that *MIDI_RxCommand* be used in preference to byte reads.
Normally Real Time messages are handled by the module and are not present in the receive buffer, however this can be changed by flags given to *MIDI_Init*.

On entry R0 = Port number, or -1 to look at all ports and return whichever has data.

On exit R0 = Received byte, or 0 if there is no data to return.

If a byte was received, then:
Bit 24 = 1
Bits 28-31 = Port number where this byte was received

R1 = Received time, or 0 if no byte or clock disabled.

Does *not* return an error if the buffer has overflowed. The background error event will be raised, and the error byte (check with *MIDI_InqError*) will be 'B', if overflow occurred.

MIDI_RxCommand (&404C8)

Returns the next complete MIDI command as a set of bytes (normally excluding Real Time messages).

On entry R0 = Port number, or -1 to look at all ports and return whichever has data.

On exit R0 = byte 0 is command
byte 1 is data byte 1 (or 0)
byte 2 is data byte 2 (or 0)
bits 24-25 are the number of bytes (0-3) in this command
bits 28-31 are the MIDI device that this message was received by

Will be 0 if the receive buffer is empty.

R1 = Time when last byte was received, or 0 if buffer empty or clock disabled.

Note that System Exclusive messages will be received as the command &F0 and a sequence of data bytes (across however many calls to *MIDI_RxCommand* are necessary) until done.

Does *not* return an error if the buffer has overflowed. The background error event will be raised, and the error byte (check with *MIDI_InqError*) will be 'B', if overflow occurred.

MIDI_TxByte (&404C9)

Inserts a byte into the transmit buffer to assemble a MIDI command for transmission.
Note that unlike the original MIDI module that would send the byte immediately, USB MIDI works with a series of command packets, so a packet is assembled and transmitted in its entirety.
Note that sending Running Status is not supported.

On entry R0 = byte 0 is the byte to transmit
bits 28-31 specify the MIDI port (0-3) to transmit from

On exit –

Does *not* return an error if the byte is discarded due to incorrect type (the commands are vetted). Use *MIDI_InqError* to check the status is *not* "D".

MIDI_TxCommand (&404CA)

Transmit or schedule a complete MIDI command on the channel defined by *MIDI_SetTxChannel*.

On entry R0 = byte 0 is command
byte 1 is data byte 1 (or 0)
byte 2 is data byte 2 (or 0)
bits 24-25 *optional* number of bytes in this command *if not a known command*
bits 28-31 are the MIDI port to transmit from

R1 = 0 to send immediately, or clock value upon when to send (in Fast Clock mode)

On exit R0 = Number of scheduler slots *free* in queue, or -1 if failed because the queue is full.

Running status is not supported, only complete commands will be sent.

Any partial data in the transmit buffer (via *Midi_TxByte*) will be overwritten by the data provided in this command.

Commands are sanitised, and will be discarded (without error) if the command byte does *not* have bit 7 set, or any of the data bytes *do* have bit 7 set.
The exception to this rule is when sending a System Exclusive message because *SysEx messages cannot currently be sent.*

Notes that are *earlier* than the current schedule time are sent *immediately*, they don't error.

The size of the scheduler queue is 1024 commands.

MIDI_TxNoteOff (&404CB)

Transmits a MIDI "Note Off" command on the currently selected channel.

On entry R0 = Note (0-127, middle C is 60 and each digit represents one semitone)

R1 = Key off velocity (0-127)

On exit –

MIDI_TxNoteOn (&404CC)

Transmits a MIDI "Note On" command on the currently selected channel.

On entry R0 = Note (0-127, middle C is 60 and each digit represents one semitone)

R1 = Velocity (0-127; using velocity 0 is an alternative to Note Off; 64 is "average")

On exit R1 = `-1` if some sort of error ocurred.

Does *not* return an error if the MIDI device is not present. It sets R0 to `-1` instead. This is so older programs unaware of USB disconnection don't crash when the device is removed.

MIDI_TxPolyKeyPressure (&404CD)

Transmits a MIDI "Poly Key Pressure" command for after-touch effects. The exact effect of this command varies depending on the receiving device and the selected voice.

On entry R0 = Note (0-127, middle C is 60 and each digit represents one semitone)

R1 = Pressure value (0-127)

On exit –

MIDI_TxControlChange (&404CE)

Transmit a MIDI "Control Change" command.

On entry R0 = Control number (0-127)

R1 = Control value (0-127)

On exit –

Control numbers 122-127 are reserved for Channel Mode Messages, which may be either transmitted using this SWI or by the following 6 SWIs (*MIDI_TxLocalControl* to *MIDI_TxPolyModeOn*).

MIDI_TxLocalControl (&404CF)

Transmits a MIDI “Local Control” command (control number 122).

On entry R0 = 0 for Local Control Off, or

127 for Local Control On

On exit –

MIDI_TxAllNotesOff (&404D0)

Transmits a MIDI “All Notes Off” command (control number 123).

On entry/exit –

MIDI_TxOmniModeOff (&404D1)

Transmits a MIDI “Omni Mode Off” command (control number 124).

On entry/exit –

MIDI_TxOmniModeOn (&404D2)

Transmits a MIDI “Omni Mode On” command (control number 125).

On entry/exit –

MIDI_TxMonoModeOn (&404D3)

Transmits a MIDI “Mono Mode On” command (control number 126).

On entry R0 = The number of channels to respond to (1-16), or 0 to respond on all supported channels.

On exit –

Refer to your MIDI device handbook, as multi-timbral devices frequently have *interesting* interpretations of what this means.

MIDI_TxPolyModeOn (&404D4)

Transmits a MIDI “Poly Mode On” command (control number 127), and thus ends Mono Mode.

MIDI_TxProgramChange (&404D5)

Transmits a MIDI "Program Change" command, that is to say to instruct the instrument to change its program/voice/tone/patch/instrument and start sounding like something different.

On entry R0 = Program (voice) number (0-127)

On exit –

The *General MIDI* specification defines 128 voices, which are arranged as 16 "families" (piano, reed, brass, etc) each containing 8 instruments.

PIANO

1 Acoustic Grand
2 Bright Acoustic
3 Electric Grand
4 Honky-Tonk
5 Electric Piano 1
6 Electric Piano 2
7 Harpsichord
8 Clavinet (not *Clarinet*)

CHROMATIC PERCUSSION

9 Celesta
10 Glockenspiel
11 Music Box
12 Vibraphone
13 Marimba
14 Xylophone
15 Tubular Bells
16 Dulcimer

ORGAN

17 Drawbar Organ
18 Percussive Organ
19 Rock Organ
20 Church Organ
21 Reed Organ
22 Accoridan
23 Harmonica
24 Tango Accordian

GUITAR

25 Nylon String Guitar (acoustic)
26 Steel String Guitar (acoustic)
27 Electric Jazz Guitar (electric)
28 Electric Clean Guitar (electric)
29 Electric Muted Guitar (electric)
30 Overdriven Guitar
31 Distortion Guitar
32 Guitar Harmonics

BASS

33 Acoustic Bass
34 Electric Bass (finger)
35 Electric Bass (pick)
36 Fretless Bass
37 Slap Bass 1
38 Slap Bass 2
39 Synth Bass 1
40 Synth Bass 2

SOLO STRINGS

41 Violin
42 Viola
43 Cello
44 Contrabass (double bass)
45 Tremolo Strings
46 Pizzicato Strings
47 Orchestral Strings
48 Timpani (a drum?)

ENSEMBLE

49 String Ensemble 1
50 String Ensemble 2
51 SynthStrings 1
52 SynthStrings 2
53 Choir Aahs
54 Voice Oohs
55 Synth Voice
56 Orchestra Hit

BRASS

57 Trumpet
58 Trombone
59 Tuba
60 Muted Trumpet
61 French Horn
62 Brass Section
63 SynthBrass 1
64 SynthBrass 2

REED

65 Soprano Sax
66 Alto Sax
67 Tenor Sax
68 Baritone Sax
69 Oboe
70 English Horn
71 Bassoon
72 Clarinet

PIPE

73 Piccolo
74 Flute
75 Recorder
76 Pan Flute
77 Blown Bottle
78 Skakuhachi (a bamboo flute)
79 Whistle
80 Ocarina (like a ceramic seashell with holes in it)

SYNTH LEAD (synth lead melody)

81 Lead 1 (square)
82 Lead 2 (sawtooth)
83 Lead 3 (calliope (steam organ))
84 Lead 4 (chiff)
85 Lead 5 (charang (like a lute))
86 Lead 6 (voice)
87 Lead 7 (fifths)
88 Lead 8 (bass + lead)

SYNTH PAD (continual tone)

89 Pad 1 (new age)
90 Pad 2 (warm)
91 Pad 3 (polysynth)
92 Pad 4 (choir)
93 Pad 5 (bowed)
94 Pad 6 (metallic)
95 Pad 7 (halo)
96 Pad 8 (sweep)

SYNTH EFFECTS

97 FX 1 (rain)
98 FX 2 (soundtrack)
99 FX 3 (crystal)
100 FX 4 (atmosphere)
101 FX 5 (brightness)
102 FX 6 (goblins)
103 FX 7 (echoes)
104 FX 8 (sci-fi)

ETHNIC

105 Sitar (Indian/Pakistani guitar)
106 Banjo (you watched *Deliverance*, right?)
107 Shamisen (Japanese 3-string guitar-like)
108 Koto (Japanese 13 strings-on-a-board)
109 Kalimba (African plucked metal tines)
110 Bagpipe
111 Fiddle
112 Shanai (Iran/India/Pakistan double-reed oboe)

PERCUSSIVE

113 Tinkle Bell
114 Agogo
115 Steel Drums
116 Woodblock
117 Taiko Drum
118 Melodic Tom
119 Synth Drum
120 Reverse Cymbal

SOUND EFFECTS

121 Guitar Fret Noise
122 Breath Noise
123 Seashore
124 Bird Tweet
125 Telephone Ring
126 Helicopter
127 Applause
128 Gunshot

Note that the voices are numbered 1 to 128 while the MIDI commands count from zero, thus you should subtract one from the above numbers when selecting which voice to play.

Modern synthesisers and keyboards offer a far greater variety of voices, such as the *Erhu* (Chinese 2-string fiddle), *Tamboura* (Balkan lute), and *Church Bells*, among many others. However the methods used to select these additional voices are more complicated.

Usually you would issue a "Bank Select" command (controller 0 (coarse) and 32 (fine)) to switch the bank, then a Program Select to choose the specific voice.
Unfortunately, the allocation of voices and the methods of accessing them varies between manufacturers and even devices; so you will need to consult your handbook for specifics.

MIDI_TxChannelPressure (&404D6)

Transmits a MIDI "Channel Pressure" command. This is akin to modifying the velocity as the note is playing and may change the modulation, pitch, or volume depending on the voice and instrument capabilities.

On entry R0 = Pressure value (0-127)

On exit –

The difference between KeyPressure (aftertouch) and ChannelPressure is that KeyPressure is applied to individual notes and would usually be expected to control the LFO (giving a vibrato effect) while ChannelPressure is applied to all notes playing on the channel and is usually expected to control the VCA (volume).

If you want to know more about LFO, VCA, and music synthesis in general, there is a good introductory description at `http://beausievers.com/synth/synthbasics/`

MIDI_TxPitchWheel (&404D7)

Transmits a MIDI "Pitch Wheel" command. This alters the pitch of the voice by a few cents to transpose the instrument slightly. This permits you to perform effects such as a user controlled vibrato or to 'slide' (portamento) into the next note to be played, and so on. How this is implemented depends upon the instrument's capabilities.

On entry R0 = Pitch change (0-16383 (0-&3FFF) with 8192 (&2000) being 'centre' position)

On exit –

The recommended range of the pitch wheel is +/- 2 semitones; however this is not standardised. Some instruments permit the pitch wheel range to be configured (RPN #0).

Why may the pitch alteration be necessary? MIDI, and Western music in general, works on a system of twelve tone equal temperament. As such, some forms of music (Arabic music (which ostensibly uses a 24 tone system with quarter tones; though the exact iterpretation of the tonal system differs by region), or music using *exact* frequencies (such as specifying 500Hz)) cannot easily be represented by MIDI note values alone. By using a combination of a MIDI note and a pitch modification, *any* frequency is available for play.
The range is a 14 bit number with some eight thousand offsets from the centre point permitting melismatic adjustment of the notes; though individual implementation limitations may throw away a lot of the data (for instance, working only in +/- 127 steps).

MIDI_TxSongPositionPointer (&404D8)

Transmits a MIDI "Song Position Pointer". *This does **not** automatically update the internal copy as song positions are not supported by the MIDI module.*

On entry R0 = Song Position Pointer (0-16383 (0-&3FFF))

On exit –

Songs are assumed to start at MIDI beat 0. The value specified is the MIDI ***beat*** upon which to start the song. Each MIDI *beat* spans six MIDI *clocks*. There are 24 MIDI *clocks* in a crotchet (quarter note), so each *beat* is the same duration as a semiquaver (16th note).

MIDI_TxSongSelect (&404D9)

Transmits a MIDI "Song Select". This is for choosing a specific song stored in the instrument for playback.

On entry R0 = Song number (0-127)

On exit –

Song #0 should play the first song, even though most instruments that store songs will display them to the user counting from 1.

MIDI_TxTuneRequest (&404DA)

Transmits a MIDI "Tune Request" command. This is for older synthesisers with oscillator circuits, and most likely doesn't do anything on digital synthesisers.

On entry / exit –

MIDI_TxStart (&404DB)

Transmits a MIDI "Start" command. This resets the beat counter and the Song Position to zero and causes the instrument to begin playback of the selected song.
(there are no internal effects as beat timing is not implemented in the MIDI module)

On entry / exit –

MIDI_TxContinue (&404DC)

Transmits a MIDI "Continue" command. This causes the instrument to continue playback of the previously selected song. As for *MIDI_TxStart*, there are no internal effects.

On entry / exit –

The difference between this and *MIDI_TxStart* is that the beat counter is *not* reset.

What this means is – to *play* a song:
Select the song with *MIDI_TxSongSelect*
Start it playing with *MIDI_TxStart*

-

Or otherwise – to play a song *from a specific position*:
Select the song with *MIDI_TxSongSelect*
Set the playback start position with *MIDI_TxSongPositionPointer*
Start it playing from that position with *MIDI_TxContinue*

MIDI_TxStop (&404DD)

Transmits a MIDI "Stop" command. The instrument will cease playing the song, though it will remember the current playback position (so a call to *MIDI_TxContinue* can resume playing from where it was stopped).

On entry / exit –

MIDI_TxSystemReset (&404DE)

Transmits a MIDI "System Reset" command. This resets the instrument to a state that is usually the same as its power-on state.

On entry / exit –

The effect of this command depends upon the instrument, however generally speaking:

- All playing notes will be silenced, any song playback will be stopped.
- The local keyboard will be enabled.
- Running status and timers will be reset.
- The device may reset to the default voice.
- The device may reset to Omni, Poly mode if there is no default saying otherwise.
- Anything else specific to the device.

This command *should not be used* as a matter of course; it should only be sent in response to a specific request from the user.

MIDI_IgnoreTiming (&404DF)

This operates as a switch and tells the system to either ignore any further timing Clock messages as well as Start/Continue/Stop messages, or to revert to normal reception of them.

On entry R0 = 0 to receive messages (default)
= 1 to ignore timing messages

On exit -

Note that enabling this will cause these messages to be completely discarded, they won't go in the receive buffer.

MIDI_TxSynchSoundScheduler (&404E0)

This SWI is recognised, but it has no effect. Replies with dummy data.

On entry –
On exit R0 = 0 (specifies sound scheduler is synchronised to the Sound Interrupt (default))

MIDI_FastClock (&404E1)

This SWI controls the Fast Clock. It is strongly recommended that you use this timing mode.

On entry R0 = <0 read current value of Fast Clock

= 0 stop Fast Clock, revert to beat timing

= >0 start Fast Clock
The associated behaviour of sending Timing Clock messages every *n* milliseconds is not currently supported.

R1 = If R0 > 0 then this is the value to reset the Fast Clock to.

On exit R0 = Preserved
R1 = *Previous* value of Fast Clock

The Fast Clock increments every *millisecond*.
This SWI must be called at least once to enable the Fast Clock in order to use the scheduling facilities. The value in R1 initialises the clock, and schedule times are relative to this.

Fast Clock requires access to Timer 1. Obviously this cannot be used with other software that uses Timer1 or hardware that lacks an available timer.

MIDI_Init (&404E2)

Reset the internal MIDI system status, or perform certain partial resets.

On entry R0 = 0 to reset everything
bit 0 set ignored - module doesn't support Running Status
bit 1 set to clear receive buffers
bit 2 set to clear transmit buffer (this is only useful if using *TxByte*)
bit 3 set to clear scheduler buffer
bit 4 set clear current error (if possible, some like "no USB device" cannot be cleared!)
bit 30 set Place received Real Time messages are to go into the receive buffer
bit 31 set Do not perform special actions on Real Time messages (ignore them)

On exit R0 = Number of MIDI ports installed (subtract one from the maximum port number)

There is a slight difference between this MIDI module and the Acorn original in reporting the number of MIDI ports. USB MIDI devices are allocated a port number in the order that they are connected (or if already connected, in the order they are presented to the system). The value returned in R0 is the number of MIDI devices *currently available*; **which can be zero**. Additionally, there is no guarantee that they are sequential. Two devices might be 0 and 1, or 0 and 3...

Certain applications may require setting bits 30 and 31 in order that the module pass through Real Time messages (without the module handling them) in order to fully capture the MIDI data stream.

If only Clock messages are to be inhibited, it is recommended to use *MIDI_IgnoreTiming* as setting bit 31 will cause *all* Real Time messages to be ignored.

MIDI_SetBufferSize (&404E3)

This SWI is recognised, but it has no effect. Replies with dummy data.

On entry –

On exit R0 = 2048 (buffer size in bytes)
R1 = 2048 (just repeating the previous value)

The buffers are fixed and cannot be altered. Additionally, the buffer sizes (given in the original user guide as "5 × size × MIDI ports" isn't valid as this module works differently.

Setting a larger buffer size may have been useful on an 8MHz machine in order to receive complex data without loss. This shouldn't be an issue on modern machines.

MIDI_Interface (&404E4)

Any attempts to call this SWI will result in an error being raised.

The original MIDI module ran on 8MHz hardware and message timing is in the order of microseconds. Keep in mind that MIDI runs at 31250 bits per second and a lot of computer systems of this age/clockspeed struggle with more than 19200bps. To put this into context, MIDI can pass just over 3000 bytes per second, which on a system running flat out *could* equate to 1000-2000+ commands *every second.* To further compound the issue, the Acorn MIDI hardware did not offer much in the way of buffering, so a fast software response time is important. The original MIDI module was written in pure assembler to go as quickly as possible (this was in the late '80s when using C was not very commonplace); and to further provide speed improvements over something that needed a very fast response time, Acorn provided a SWI that returns some pointers into the MIDI module, such that R0 is a pointer to the MIDI module workspace, and R1 is the address of the SWI handler code. By providing *this*, the user can place the computer into SVC mode, place the SWI offset (counting from the first SWI, so it is basically the SWI number with &40400 subtracted) into R11, place the workspace pointer into R12, and then jump directly into the module, bypassing the entire operating system SWI decode and call mechanism.

In this day and age, we (should!) recoil in absolute horror at the idea of branching *directly* into module space from a user mode application. I get it, speed was of critical importance, but I figure people might say the same thing as they're going 100kph down the wrong side of the road because <insert lame excuse here>. Really, there are some things that are just plain icky and *this* is one of them. This is like olive flavoured ice cream. Just... Don't...

MIDI_USBInfo (&404EA)

Returns information on the MIDI subsystem.

On entry	R0	=	0	=	General information
			1-4	=	Specific information on a USB MIDI device

If R0 = 0 then:

On exit	R0	=	Number of connected MIDI devices (0-4).
	R1	=	Bitmap of connected MIDI devices (bit 0 = port 0 ... bit 3 = port 3).
	R2	=	Undefined.
	R3	=	Undefined.
	R4	=	Undefined.
	R5	=	Undefined.
	R6	=	Undefined.
	R7	=	Undefined.

These "undefined" values may carry information indended for debugging and development, which may change from one release to another. You should not use this data or make any assumptions about meaning or presence of said data.

If R0 = 1-4 then:

On exit	R0	=	Number of bytes *used* (remaining to be read) in the device's receive buffer; or `-1` if the device is not valid (and if `-1`, no other data will have been set).
	R1	=	Pointer to USB device name string (like "USB8")
	R2	=	Pointer to device's product name string.
	R3	=	Pointer to device's receive buffer.
	R4	=	Receive buffer head pointer (offset from buffer start where incoming data goes).
	R5	=	Receive buffer tail pointer (offset from buffer start where Rx SWIs read from).
	R6	=	Non-zero if this device is flagged as slow hardware that needs delays.
	R7	=	Pointer to internal device decriptor (this is for debugging, *you should not use it*).

Everywhere *else*, MIDI devices are numbered 0-3 however this SWI takes device numbers 1-4. This is so passing zero can provide a generic reply.

Some cheap hardware doesn't set the USB Product data. If this is the case, the module will instead provide the USB Manufacturer data, so you should get some sort of response.

For example, to obtain the USB device ID and the product name:

```
>SYS "MIDI_USBInfo", 1 TO , id$, product$
>PRINT id$, product$
USB15     USB2.0-MIDI
>
```

MIDI_Options (&404EB)

This SWI allows you to set various options.

On entry
R0 = Options bitmap, or `-1` to read
R1 = Bitmap of devices that need hardware delays, or `-1` to read

On exit
R0 = Value of options bitmap.
R1 = Value of delay bitmap.
R2 = Undefined.
R3 = Undefined.

The available option is:

bit 0 Fake Fast Clock timestamps using system ticker instead of high resolution timer. This may have side effects.

The delay bitmap is bit 0 for device 0, bit 1 for device 1, etc.

If a device needs delay, the module will insert a pause after data has been handed over to the USB system for transmission.
The delay is 320 microseconds × (number of bytes send + 1). It uses the HAL to perform the delay at *micro*second speeds, so the entire machine may be paused for up to 1,280µS.
To put this into context, each 320µS delay is about a third of a millisecond, so the higher resolution timer used for timestamping data isn't accurate enough.
It's mostly, mind you, just a hack to get lame and cheap serial MIDI interfaces working by artificially delaying for the duration it would take to actually send the data at 31250bps.

Commands

The following commands are provided *only* for compatibility, ***they have no effect.***

***MidiSound** in|out|off [<port>]
***MidiTouch** on|off
***MidiChannel** <channel (1-16)>
***MidiMode** <mode (1-4)>
***MidiStart** <time>
***MidiStop**
***MidiContinue**

The USB MIDI module also provides the following three commands:

***MidiUSBSend** [P<port>] <command> [<parameter1> [<parameter2>]]

This command sends a MIDI command to MIDI device 0. The command is any valid General MIDI 1.0 command, so the following should cause a Middle C to be played:

```
*MidiUSBSend 144 60 64
```

If the optional port (prefixed with a P) is specified, then the command will be directed to the given port.

*MidiUSBInfo

This command reports information on the USB MIDI setup, looking something like this:

```
MIDI USB information:
  Timestamp type is Fast Clock
  MIDI Clock is 0 (Song Position = 0)
  Current TX channel is 0 (real channel 1 on real port 0).

Information for port 0:
  Current MIDI device is USB15, VID 1A86, PID 752D,
  using IN file handle 254 and OUT file handle 246.
```

The exact report depends upon what, if any, devices are connected.

*MidiUSBDebug

This command reports much more techical information on the USB MIDI setup, which may be useful if things are not working as expected.

Service calls

The MIDI module makes four service calls.

Service_MIDI (&58)

The *Service_MIDI* service call provides some information on the module status, as defined by the Acorn MIDI module:

R0	=	0	=	module has initialised
		1	=	module is dying

Additionally, the USB MIDI module provides these service calls:

R0	=	10	=	a USB MIDI device has been connected
		11	=	a USB MIDI device has been disconnected

Events

The MIDI module provides several *events*.

Event_MIDI (&11)

The original MIDI module provided the following events:

R0 = &11 (Event_MIDI)

R1 = 0 = MIDI_DataReceivedEvent
A receive buffer was empty and has now received data.

1 = MIDI_ErrorEvent
An error has occurred in the background (use the *MIDI_EnqError* SWI to determine what the error was).

2 = Unimplemented (to do with scheduling)

The USB MIDI module adds the following events:

10 = MIDI_DeviceConnectedEvent
A USB MIDI device has been connected.

11 = MIDI_DeviceDisconnectedEvent
A USB MIDI device has been disconnected.

(this page has been left blank for your notes)

(this page has been left blank for your notes)