This is the "*how it is supposed to work*" manual. Reality may differ. Greatly.

USB-MIDI

v0.01 (alpha 3)

2013/08/07

by

Rick Murray

heyrick1973 -*at*- yahoo -*dot*- co -*dot*- uk

Work in progress - stuff may change.

Introduction

This document describes the USB-MIDI module, version 0.01. 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 *Beagleboard* and the *RaspberryPi*.

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 dummy SWIs and do nothing.
  - The MIDI module does not support timing.
  - The MIDI module does not interpret and act upon system messages; everything is placed in the receive buffer.

- 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.

- 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 1024 bytes, the transmit buffer is 3 bytes (yes, *three* ... when a valid packet has been formed, it will be output immediately).
  These values are fixed and *cannot* be changed.

- Scheduling/timing is not supported, neither are MIDI commands timestamped. MIDI data is available when MIDI data arrives. MIDI data is sent as it is given to the module. No exceptions. The scheduling command will return an error of no free schedule slots if you attempt to schedule data. Either run your own timing/dispatch or use the `Sound_QSheduler` SWI (as *!Maestro* does).

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.

- Only the first MIDI interface encountered is used, others are ignored.
  Resolution:
  None, yet. A future version of the MIDI module will support up to four USB devices.

- 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. If this is a grave limitation to you, please get in touch.

- Only one program at a time can utilise the MIDI interface (as a program taking data out of the buffer means it will not be there for other programs).
  Resolution:
  None as yet. Please get in touch if you'd like to discuss ideas.

- **THIS IS ALPHA QUALITY SOFTWARE AND MAY CONTAIN BUGS.**
  Just thought I'd make that really clear..

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 5.19.

If you are the creator of MIDI sequencing software and my MIDI module does *not* work with your software; consider sending me a complete copy of your software product (registered, if commercial) and I will see what I can do to support it (*though, note the lack of timing support, if you use MIDI_TxCommand with a schedule time*).

If you are the creator of MIDI sequencing software that *does* work and you're feeling generous enough to send me a copy, I will give it a mention in this user guide. **(^_^)**

FUTURE DEVELOPMENT

IT IS VERY IMPORTANT TO NOTE THAT THERE ARE NO PLANS TO ADD MUCH IN THE WAY OF FUNCTIONALITY TO THE USB MIDI MODULE.

What I intend to do:

- Look to supporting up to four simultaneous MIDI devices.
- Specific requests if viable, to better support existing software that uses the Acorn MIDI module in certain (legal) ways.
- Bug fixes.
- Creeping Featuritis. I *know* I'm going to throw in a few things, so I'll 'fess up that I will probably throw in a few things.

Now for the rationale. I do believe the Acorn MIDI module interface is *messy* and does not necessarily lend itself well to system-wide MIDI support. To give an example, if a MIDI timing message is received, the (Acorn) MIDI system itself can deal with this, or it can put it into the input queue for the application to pick up. Given that MIDI ticks are percentages of a beat and beats are percentages of a second (something like 24 per beat, and an allegro 120bpm will run at 2 beats per second, or 12 ticks per second), wouldn't it be better to have some mechanism to broadcast this through the OS so the application can *know* that a tick has happened? Or a broadcast to know that data has been received, and by which interface? I like the idea of having "easy" SWIs like *TxNoteOn* instead of having to remember MIDI command sequences, so I may preserve that, but rationalise it a little bit to reduce the gazillion different SWIs provided by this module. For instance, a *PlayNote* SWI that turns notes on (velocity non-zero) and off (velocity zero). Maybe some sort of automatic *and programmable* system to extend the programme (voice) selection to cater for more modern devices that offer rather more than the standard General MIDI defined voice set.

In short, writing an implementation of Acorn's MIDI module for USB devices is partially to scratch a personal itch, partially to learn about USB interfacing, but mostly just a stop-gap. It is my hope and intention to devise something better.

As you are bothering to read this, I assume you have an interest in MIDI on modern RISC OS machines. Therefore, I would really appreciate it if you could contact me. I would like to hear two things from you. Firstly, I would like you to provide me with your thoughts on a potential future API. If you have to talk to MIDI devices as an application programmer (that's probably what you are if you are reading this!), how would you like this to happen? How would you like to be notified of MIDI events? How would you like to send and receive MIDI commands? Please keep it realistic, though!
Secondly, throw in your wish list. This doesn't need to be realistic. Suggest stuff that you think would be "epic". Who knows, I might agree and put it in the API!

Just one note – however – there are no plans at all for any form of collusion with older Acorn-era hardware. I have no MIDI hardware on my older machines, nor USB hardware either. **The base bottom line is RISC OS 5.xx on something *modern***; the Beagles, the RasperryPis, the Panda and Pandora, etc. I'm sorry if this disappoints some of you, but maybe it is time to acknowledge that the RiscPC *is nearly **twenty** years old!*
Perhaps it is time to move on...
[...and fit a RISC OS 5 ROM set into your RiscPC? :-)]

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, USB to serial MIDI convertors exist and can be found for under £10 on eBay and similar. These convertors can be used for the case (as is *often* the case) that the computer you wish to connect your MIDI keyboard to offers no sort of serial connection, only USB.

Internally, each MIDI instrument has a transmitter and 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 yourself or third-party software. The RISC OS music scene is not particularly active these days. Unfortunately we don't have anything to rival *Sibelius* (which is ironic given that it was first developed for RISC OS). Maybe this may change in the future? Until then, you might like to know that you can get some impressive results out of the rudimentary music editor supplied with RISC OS – *!Maestro*. Just load the MIDI module, load *!Maestro*, load a music file (I suggest the one called "*Fanfare*"), then start it playing.

Connecting it up

In the old days, there were three MIDI ports (IN, THRU, OUT) which had specific purposes.

These days, just find a USB lead that fits (it is usually the type with the big plug). Plug that into the keyboard, then plug the normal flattish USB plug into a free USB slot on your computer (or a hub). Wait a short while for the system to notice the device.
That's it.

For connecting older equipment, the situation is slightly more complicated. You should have an interface with a USB lead at one end and two chunky-looking round connectors with five pins inside at the other end. The round connector marked "IN" plugs into the port on your instrument that says "OUT" (yes, back to front). The one marked "OUT" goes into the port marked "IN". The USB connector goes into your computer. 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
```

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

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 (*will currently be zero*)
R1 must be zero

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 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 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 command is recognised, but it has no effect.

MIDI_SetMode (&404C1)

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

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 1-16 of device 1 (*not currently supported*)
33-48 means channels 1-16 of device 2 (*not currently supported*)
49-64 means channels 1-16 of device 3 (*not currently supported*)

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

If the channel points to a device that is not present, an error will be raised.

MIDI_SetTxActiveSensing (&404C3)

This functionality is not currently implemented. Replies with dummy data.

On exit
R0 = 0 (= all devices: Active Sensing not enabled; not receiving Active Sensing data)

MIDI_InqSongPositionPointer (&404C4)

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

On exit
R0 = 0 (position)
R1 = 48 (bit 4 = system messages in RX buffer; bit 5 = no special treatment of them)

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. *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 failure (*not currently supported*)
"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 "B", "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.

On entry	R0	=	Device number (*should be 0*)
On exit	R0	=	Received byte, or 0 if there is no data to return.
	R1	=	0

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.

On entry	R0	=	Device number (*should be 0*)
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
	R1	=	0

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.

On entry	R0	=	byte 0 is the byte to transmit bits 28-31 specify the MIDI device (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 a complete MIDI command.

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 are the number of bytes (0-3) in this command bits 28-31 are the MIDI device that this message was received by
	R1	=	0 (*immediate*)
On exit	R0	=	0, or -1 if R1<>0 on entry

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, in which case any data will be accepted after &F0 until &F7 is passed.

If R1 was non-zero on entry, R0 will be returned as `-1` and the command will *not* have been sent. *This version of the MIDI module does not support scheduling.*
SysEx messages cannot currently be sent.

MIDI_TxNoteOff (&404CB)

Transmits a MIDI “Note Off” command.

On entry	R0	=	Note (0-127, middle C is 60 and each digit represents one semitone)
	R1	=	Velocity (0-127)
On exit	–		

MIDI_TxNoteOn (&404CC)

Transmits a MIDI “Note On” command.

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
R1 = Control value
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. In other words, tell it to stop being a piano and start being a violin (if supported, that is).

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

The *General MIDI* specification defines 128 voices, which are arranges 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*, 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 music; though 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 previously selected song.
(there are no internal effects as 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 command is recognised, but it has no effect.

On entry / exit –

MIDI_TxSynchSoundScheduler (&404E0)

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

On entry – (would be <0 to read value; 0 to stop fast clock; >0 to set the fast clock Tx rate)
On exit R1 = 0

MIDI_Init (&404E2)

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

On entry R0 =
- 0 to reset everything
- bit 1 set to clear receive buffers
- bit 2 set to clear transmit buffer (this is only useful if using *TxByte*)
- bit 3 set ignored
- bit 4 set clear current error (if possible, some like “no USB device” cannot be cleared!)
- bit 30 set ignored (System Realtime Messages are always put into RX buffer)
- bit 31 set ignored (no special actions are performed on System Realtime Msgs)

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**.

MIDI_SetBufferSize (&404E3)

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

On entry	–		
On exit	R0	=	1024 (buffer size in bytes)
	R1	=	1024 (total number of bytes claimed; is *not* ×5 because data is not timestamped)

The buffers are fixed and cannot be altered.

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 = USB logical device numbers such that the first device is in byte 0, the fourth device is in byte 3.
R2 = Pointer to the internal raw receive buffer.
R3 = Undefined.
R4 = Undefined.
R5 = `1` if the global poll ticker is currently in use, else `0`.
R6 = Last poll interval, in centiseconds.
R7 = Pointer to internal device data table (this is for debugging, *you should not use it*).

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 = USB logical device number of this device.
R2 = Pointer to the internal raw receive buffer.
R3 = Pointer to device's receive buffer.
R4 = Pointer to device's transmit buffer.
R5 = `1` if this port is receiving on a ticker callback, else 0.
R6 = Value of *lasterror* byte for this port (same as for *MIDI_InqError*).
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.

At this time, only the first port/device is valid, anything else will reply with `-1`.

The device descriptor array is as follows, but this is for completeness and may change:

```
typedef struct device_descriptor
{
   int  isvalid;    // Non-zero if device is valid.
   char id[8];      // Device ID, like "USB10".
   int  lasterror;  // Last error - 0, 'A', 'B', 'D', 'X', or '/'.
   int  iep;        // IN endpoint
   int  oep;        // OUT endpoint
   int  ifp;        // IN file handle
   int  ofp;        // OUT file handle
   int  xxx;        // unused
   int  yyy;        // unused
   char *rxbuffer;  // Pointer to our virtual buffer
   int  bufferhead; // Buffer head pointer
   int  buffertail; // Buffer tail pointer
   int  rxontick;   // Are we RXing on a ticker?
   char *txbuffer;  // Pointer to our TX buffer
   int  txptr;      // Pointer (count) to stuff in TX buffer
}
```

MIDI_Options (&404EB)

This command has not yet been implemented.

On entry	R0	=	Options bitmap, or `-1` to read
On exit	R0	=	Value of options bitmap.

R1 to R3 are currently undefined on entry and exit; assume they are corrupted.

Not finalised, liable to change!

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** <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
```

***MidiUSBInfo**

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

```
MIDI USB information:
  Current MIDI device is USB7,
    using IN endpoint 1 (file handle 243)
    and OUT endpoint 2 (file handle 242)
    Current TX channel is 0 (channel 1 on port 0).
```

This will be expanded when multiple devices are supported.

***MidiUSBDebug**

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

```
MIDI USB debug information:
  Device #0:
    Is valid?       Yes
    USB device      "USB7"
    Lasterror flag  66
    IN endpoint     1
    OUT endpoint    2
    IN file handle  243
    OUT file handle 242
    RX virt buffer  &20296534 (1024 bytes)
    RX buffer head  1021
    RX buffer tail  0
    Periodic RX?    No
    RX interval     10 cs
    TX virt buffer  &2004FF34 (8 bytes)
    TX buffer ptr   0
    Raw RX buffer   &201ED1B4 (256 bytes)
    Ticker in use   No
```

This is intended for debugging, not playing with. ;-)

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)