MIDI AND THE AVR

INTRO TO USING THE AVR IN MIDI APPLICATIONS

PAUL MADDOX

MARCH 2002

TABLE OF CONTENTS

Introduction

This guide is intended to help you on the way to useing MIDI with the AVR range of processors, explaining the concepts and protocols used to some of the gotchas and common problems people face. Where possible I've tried to kept the guide as processor independant as possible.

Assumptions

You need a certain amount of knowledge about serial communication, not much, but a few basic concepts. Also a little understanding of the hardware involved, AVR, opto-isolators, etc. Also a basic knowledge of what a MIDI system is, e.g. Sequencer, synthesiser modules, keyboards.

Who is this bean?

My name is Paul Maddox and I'm a keen electronic/synth/micro enthusiast. That is to say I do this for fun, not for a living, you can see my web site here ☺ www.wavesynth.com as you can see from the site its a fairly major part of my life. I wrote this as a guide to get people into useing AVRs for MIDI, Something that very few people seem to use them for, perhaps they dont know how to? If not, this will hopefully explain it.

MIDI, what is it?

So, what's all about?

A few years back, quite a few actually, people could see the need for a common standard for communication between synthesizers. Many manufacturers had their own standards (V/hz, V/Oct, DCB, PPG BUS, etc) none of which were easily convertable to communicate with each other. With the increase of sequencers and affordable computers it was decided to got for a digital system, offering better accuracy and precision than the tradional CV/Gate type of interface. The standard they decided on was called MIDI and it stands for 'Musical Instrument Digital interface'. A webpage published by the organisation that was formed to control the standard is available at Http://www.MIDI.org/ and is a very useful resource for anything relating to MIDI.

There are *two* parts of the MIDI system to consider, the hardware interface and the protocol used. The main bulk of the issues and problems with useing MIDI are to related to the protocol, so this document focuses mostly on this, though the hardware is mentioned.

Protocol

The MIDI system is based on the concept of the user being a keyboard player, rather than say a wind instrument or guitar payer. In its simplest form MIDI is the means by which note information is sent, e.g. when a 'key' on the keyboard is pressed. The type of information that is sent is *event* information and *not* audio. That is to say MIDI is used to describe when a note is pressed, which note it is, how hard and for how long, but not the sound that is created by this action. MIDI is also used for a whole host of other events, but the action of pressing a note is the simplist to describe and emphasise the point at hand. The other thing to note is that MIDI is a *serial* data stream and runs at 31250 baud.

Hardware

This is the physical means of connection. Data is sent via a current loop, and

therefore there is no electrical connection between one unit and another via the midi cable. This prevents Ground loops and removes any chance of *hum* on the output of the device. MIDI is a *point* to *point* method of connection, ie you can only connect one device to one other. This can be increased with the use of a MIDI thru box, which is explained later.

MIDI specs, a quick guide

This section will deal with the protocol standard and explain some of the problems commonly encountered whilst developing MIDI interfaces.

A 'standard' MIDI word consists of three bytes, though depending on use it may have more or less, generally though. the first is a *Status* byte, the second and third are *databytes*.

So, with a stream of data comming in, how do you know which byte is which? and which byte is for you? lets take the action of pressing a NOTE on the keyboard, this is called a *Note Event*. Three bytes are sent; NOTE_ON , NOTE, VELOCITY
if we asssume we're working on Midi Channel one the data would look something like this

0x90, 0x3C, 0x7F

lets go through the bytes one at a time.. first the statusbyte, **0x90** if we translate this into binary we get ;- 10010000 The upper nybble (1001) shows we have a NOTE_ON event, the lower nybble (0000) is the MIDI channel, MIDI device can have a channel number between 1 and 16, because People work better with numbers starting at 1 rather than 0, so simply subtract one from the channel number to get the expected value here, e.g. if we wanted MIDI channel 8 we would expect to see 0111 (7). The next byte is the NOTE value, in this case **0x3C** which is note 60, which is in fact middle C. The final byte is **0x7F**, which is the velocity ie how hard the key was pressed, in this case maximum.

Now, many of you will have noticed that I said the maximum velocity was 0x7F, which in binary is 01111111. But surely, you ask, isnt the maximum 0xFF ie 11111111? The answer is not for MIDI, a *Status* byte is defined by the fact that it has the MSB set, *all* other bytes that are data must NOT have the MSB set, so the valid range is 0-127 for any data. This may seem very limiting at first but given that this gives nearly 12 octaves of note data and also given that very few people can press a note with an accuracy approaching seven bits, its not as bad as it may seem. Remember we're just sending note information, not audio, so resoloution isnt so much of an issue. This gives us the added advantage that status byte are easily seperated from data bytes, how? Easy *all* status bytes have the MSB set to 1, all data bytes have it set to 0. You can see this in the table below.

So here is a list of the basic commands, first column is the *Status* byte, second coloumn is the second byte and what it relates to,third coloumn the same, thirdbyte and what its use is. The fourth column is a description.

STATUS	2nd Byte	3rd Byte	Description
1000cccc	**0nnnnnnn**	**0vvvvvvv**	Note off,'cccc' is channel, 'nnnnnnn' note value''vvvvvvv' is velocity value
1001cccc	**0nnnnnnn**	**0vvvvvvv**	Note on, 'cccc' is channel, 'nnnnnnn' note value''vvvvvvv' is velocity value
1011cccc	**0nnnnnnn**	**0vvvvvvv**	Control Change, 'cccc' is channel,'nnnnnnn' is Controller Number,'vvvvvvv' is value

1110cccc	**0LLLLLLL**	**0mmmmmmm**	Pitch Wheel Change,'cccc' is channel, 'LLLLLLL' is LSB of value 'mmmmmmm' is MSB of valuecenter (no pitch change) is 0x2000
11111000	**none**	**none**	MIDI Clock pulse, 24 pulses per quarternote, 96 to the *bar* (assuming 4/4 time)
11111010	**none**	**none**	Start,sent at start of sequence, followed by *MIDI clock pulses*
11111011	**none**	**none**	Continue,sequence carries on from where it was stopped
11111100	**none**	**none**	Stop,sent when sequence is stopped
11111110	**none**	**none**	Active sensing

This is a brief (very brief) list of the events, these are the events you will see most of when useing MIDI data, there are more and there are some odd quirks which I will explain. I've also shown some of the *MIDI status* bytes that are only one byte long. MIDI clock for example, this ensures that if you have, say, a drum machine and sequencer that they stay in time with each other. active sensing is mentioned in the 'gotcha's' section of this guide, but its one you will see a lot of, so its in this list. a full list of Status bytes and message can be found on the midi.org page ;-) http://www.midi.org/about-midi/table1.htm

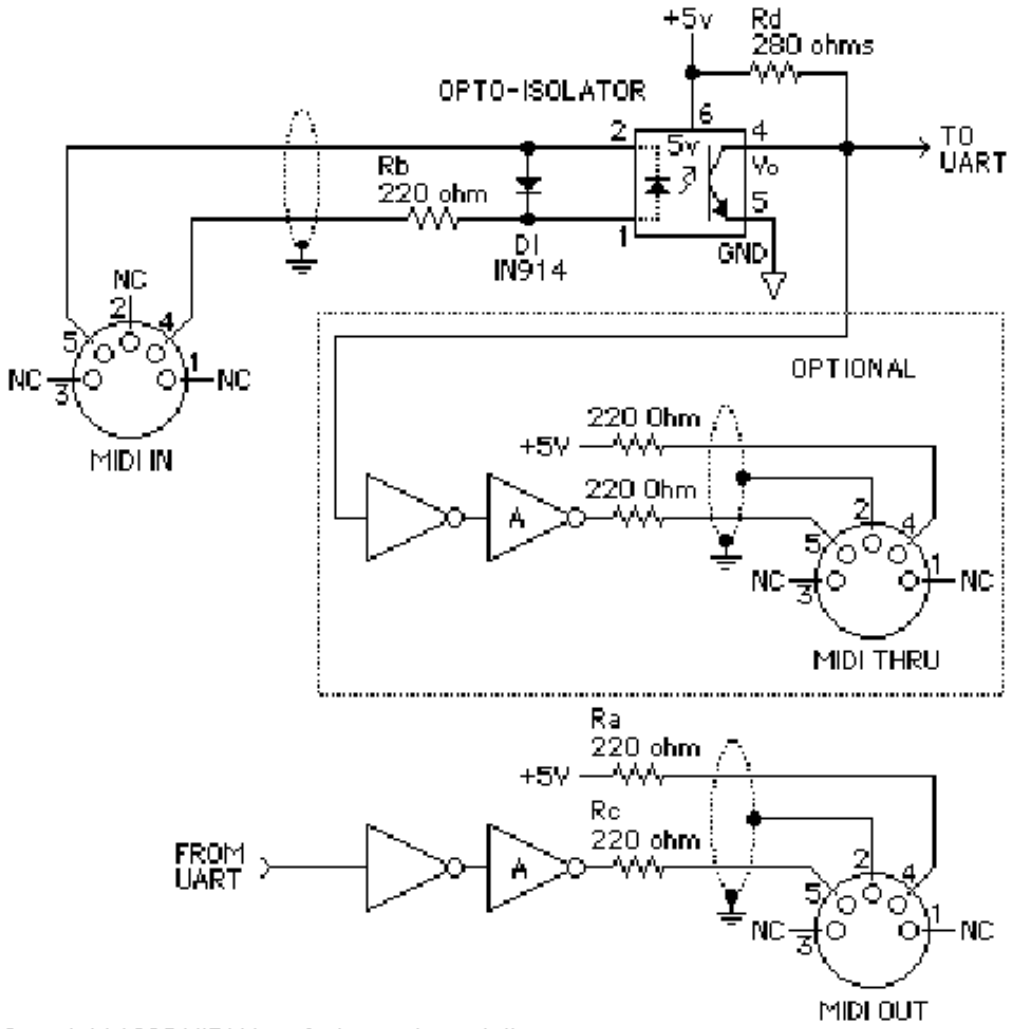
MIDI; the hardware

So, why use the AVR for MIDI?

Why not use any of the many other micros available?

There are some good solid reasons for my choice (and many others) of the AVR when useing MIDI. Firstly, it has a built HARDWARE UART, this means no messing with 'bit bashing' and constantly reading ports. Secondly, it's interupt driven! This means the AVR can be doing other things, and not worry about MIDI data until it arrives. Thirdly, it's quick! At 8MHz (on an 8515 for example) you can get nearly 8 MIPs. Processing of MIDI data can get quite complex, and with a large system you can have an almost constant stream of data to work with, so it needs to be quick and decide which bytes it wants to use and which it does not care about (data on other channels for example).

Below is a schematic showing how to connect the AVR to various MIDI ports.

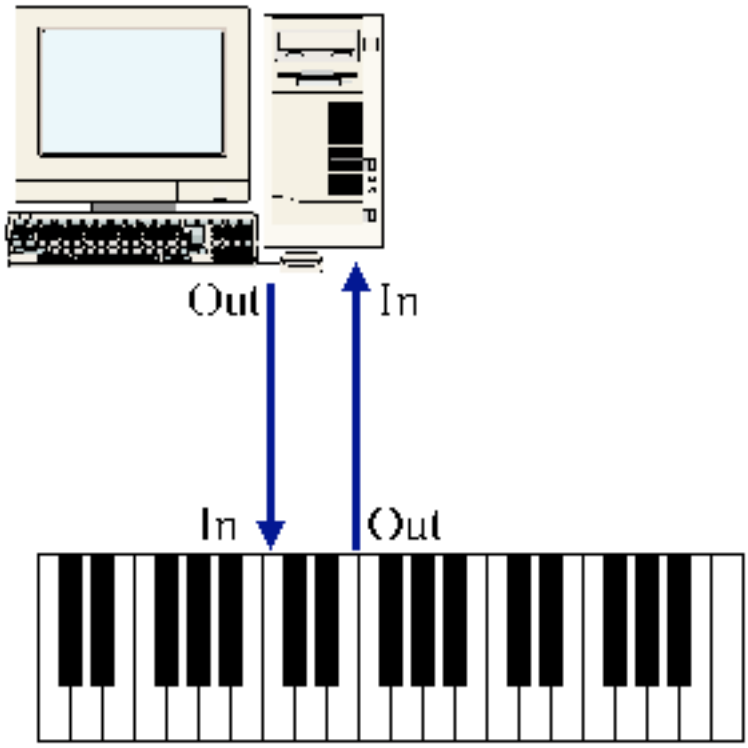


Until now I've not discussed much about the ports used. MIDI uses 3 ports and is a point to point system, i.e. you can only connect one device to one other. You can see above there are three ports:- IN, OUT and THRU..

You may be wondering 'why the opto-isolater aswell?' Well MIDI is in fact a current loop, and NOT a voltage signal. This means, in theory at least, long cables shoudn't present a problem (upto 50 feet). This also ensures that the midi interface is kept isolated from the rest of the interface, this avoids ground loops and the associated 'hum' that goes with them. Look at the schamtic, and notice pin 2, the spec for a midi cable says that the sheild should be connected to pin 2 of the 5 Pin DIN connector. The OUT and THRU ports have this pin grounded, the IN however doesn't.. another measure taken to prevent ground loops, this is important and you should stick to the standard pin configuration for the ports, especially pin 2 *not* being connected on the MIDI IN port!

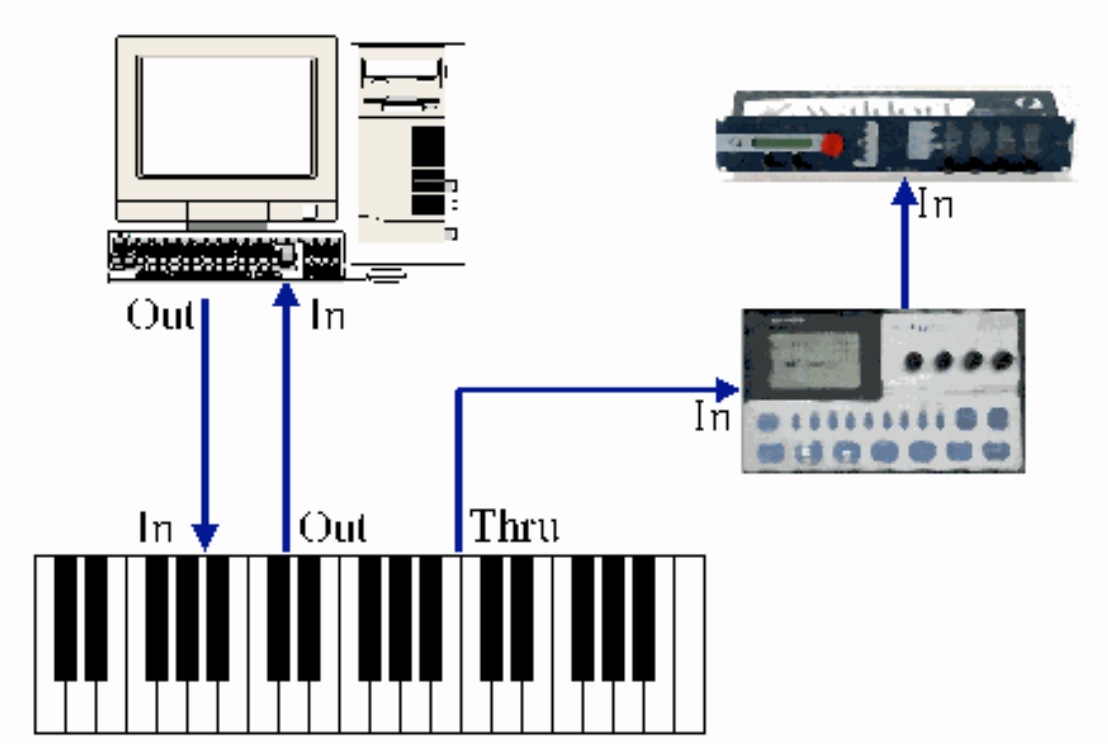
IN is used to receive data *from* another MIDI device OUT is used to send data *to* another MIDI device and THRU *replicates* what comes in on the IN port and passes it through.

So in its simplest form you might have a computer and keyboard, connected as shown below,



This setup allows you to record MIDI data on your PC, and also for your PC to play your keyboard.

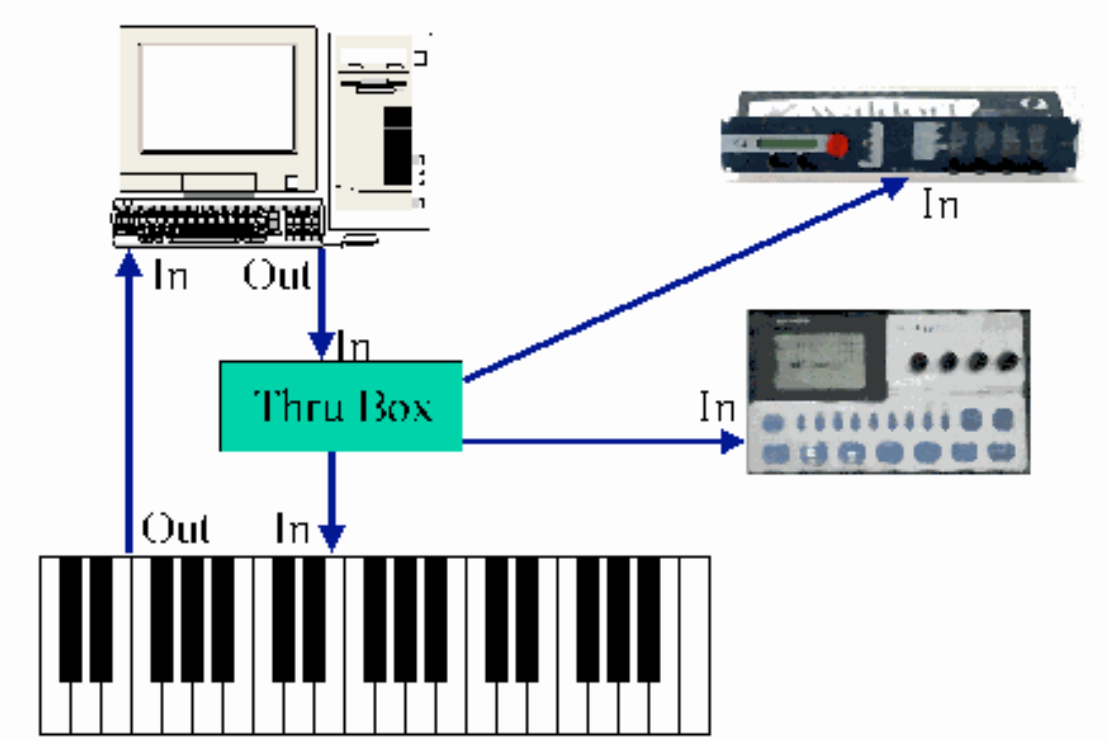
so what if you have say a computer, a keyboard, a drum machine and synthesiser module, how would you connect them? The simplist way is shown below



You can see that the computers OUT connects to the keyboard's IN, the data is then fed to the DRUM Machine through the 'THRU' port of the Keyboard, the synth module is then connected to the THRU port of the DRUM machine, so DATA

from the PC can go to the keyboard, Drum machine and Synth module. If we set up the keyboard, drum machine and synth module to use different channels, then messages for the synth module will be ignored by the keyoard and drum machine. Now your keyboard can send data to your PC, and your PC can play your keyboard, drum machine and synth Module. But, notice you cannot play your drum module or synth module from your keyboard, without first telling your PC to allow data from it to be 'passed through' its out port, this enables the PC to 'mix' data from your keyboard with the data it is generating to play the system.

But, the fact that the THRU port uses logic gates can generate delays, only small ones but a delay none the less.. Imagine you have 1 keyboard and 15 Synth modules, the delay when connected as above would become noticeable on the final synth module in the system. You can use a 'THRU PORT' module which consists of one IN port and several THRU ports.. In our case a 16 way thru port would be ideal, each synth module (and keyboard) would only be just *one* delay away from the computer sending the signals, as shown below



You can see that the ways of connecting the system you have are very flexible and expandable. This is something you should be aware of, as people will expect to be able to use your AVR MIDI module in any number of ways and expect it to work in any configuration and with any other gear.

MIDI gotcha's

So, no you've seen the basic protocol information and also the hardware.. so far it seems easy doesn't it? so why do people struggle?
There are several reasons for the problems, none of which are well documented anywhere on the web.

Running status

My program doesn't seem to receive all the note events its being sent?

This is probably the place *most* people fall down when doing anything with MIDI data streams. So, whats this all about then? Well imagine the scenario where you have a lot of data going down the line, things can get slow, if you have say 16 channels playing, each playing a three note chord at the same time, this is 16*3*3 (144) bytes to be sent instantly, you can see that at 31.25Kbaud this is a considerable delay and would be audible. so the trick used is running status, this reduces the data quite considerably to 112 bytes, hows this done?

The trick here is to realise the *common status bytes*, Takeing the example above (16 channels, each playing three notes) the data for, say channel 1, might look like this,

0x90	0x3C	0X7F	0x90	0x40	0x70	0x90	0x43	0x7A	0x91	etc
NOTE_ON	C	Velo	NOTE_ON	E	Velo	NOTE_ON	G	Velo	NOTE_ON	etc
(chan1)			(chan1)			(chan1)			(chan2)	etc

what running status does is realise that there is a common status byte for the three bits of note data, so the above stream would turn into;-

0x90	0x3C	0X7F	0x40	0x70	0x43	0x7A	0x91	etc
NOTE_ON	C	Velo	E	Velo	G	Velo	NOTE_ON	etc
(chan 1)							(chan2)	

we've saved two bytes! not a lot , but if you have 16 channels and 3 notes a channel that's 32 bytes saved, this makes a difference to the timing of the events. Another example of this would be pitch bend information, moving the wheel generates a stream of data as you move it slowly from its centre position up or down, this would be sent with just ONE status byte and then the stream of LSB and MSB data bytes...same is true for Mod-wheel, going from minimum to maximum is 128 events (0-127), sending ONE Status byte saves 127 bytes of data!

The best way to handle this is remember the status byte when it arrives, and assume that the data following is data relating to the status UNLESS you get another status byte (byte with MSB set). The only problem then is you need to keep track of which data bytes you have and haven't received a simple counter can be used for this, in the example above of the chord being played, for instance, you counter would be something like this ;-

Byte	0x90	0x3C	0x7F	0x40	0x70	0x43	0x7A	0x91	etc
counter	0	1	2	1	2	1	2	0	0

0 being the status byte, 1 being the first byte of the data, in this case note value and 2 being the second byte of data, again in this case the velocity value, when the status byte comes in for the next channel the counter is reset to '0' and as its not for us (we're interested in channel 1) we just ignore the data.

Two types of note off!

Two types? you ask, surely you only let go of a note once? Yes, But one of the things that is done by various manufacturers is to use one of two types of note off. Remember the velocity byte of the NOTE_ON command? If it was set 0 you would not expect the note not to sound, wouldn't you? I mean try pressing a note without any velocity, it can't be done.. So, using our middle C note a stream could look like this ;-)

0x90	0x3C	0x7F
NOTE_ON	C	Velo

.....some time later.....

0x80	0x3C	0x7F
NOTE_OFF	C	Velo

Velocity? on a note off? Yes, it's the *release* velocity, how quickly you let go of the key.

However some manufacturers send this ;-

0x90	0x3C	0x7F
NOTE_ON	C	Velo

.....some time later.....

0x90	0x3C	0x00
NOTE_ON	C	Velo

This last event with a note on and a velocity of '0' would mean 'turn off the note please' So your software needs to be able to recognise *both* types of note off command.

MIDI Clock

The output of my program is wrong sometimes, why? This is probably because of MIDI clock data... a little documented fact is that MIDI clock data has precedence over everything.. MIDI clock is important and the timing of MIDI should be rigid and *very* tight, so MIDI clock data can be sent at any time... what do I mean?

Consider our chord event data, you have a sequencer running and its sending MIDI clock data, playing your drum machine and all the backing tracks. You're feeling a bit confident and want to show off, so you jump on a six note chord and you expect the data stream would look like this ;-

0x90	0x3C	0x7F	0x40	0x70	0x43	0x7A	0x46	0x71	0x18	0x7F	0x24	0x73
NOTE_ON	C4	Velo	E4	Velo	G4	Velo	A#4	Velo	C1	Velo	C2	Velo

But what you get is this ;-

0x90	0x3C	0x7F	0x40	0x70	0xF8	0x43	0x7A	0x46	0x71	0x18	0x7F	0x24	0x73
NOTE_ON	C4	Velo	E4	Velo	**CLK**	G4	Velo	A#4	Velo	C1	Velo	C2	Velo

Oops, look at that big horrible clock signal in there! Your software should be able to *ignore* clocks if it doesn't use them, more importantly it should be able to handle a clock event happening in the middle of a stream of data that you're busy decoding.

Active Sensing

Sometimes I can see data when I'm not doing anything on the MIDI chain, why? This is sent when you're not doing anything on your main keyboard, i.e. when there is *no* MIDI data being generated, the status byte sent is **0xFE**, and has no following data.

Here's what the midi page says about it ;- "Use of this message is optional. When initially sent, the receiver will expect to receive another Active Sensing message each 300ms (max), or it will be assume that the connection has been terminated. At termination, the receiver will turn off all voices and return to normal (non-active sensing) operation."

This was originally intended to prevent note hangs in the middle of say a live gig when someone steps on the midi cable and unplugs it.. Instead of the unit continuously playing the note as it never received a note off command (lead got unplugged before you let go of the keys) it will realise that no data has come in for more than 300mS (no note data or Active sensing message) and say to itself 'Oops, I've lost communication with the outside world, I'd better stop playing any notes I've got!'. Quite handy when it does happen!

MIDI to CV conversion; a quick project

Ok, we've done all the theory and the explaining so lets see some code. Here is the .ASM file. This example uses an 8515 running at 4Mhz, with a couple of simple changes it could be made to run on any AVR that has a hardware UART.

Note Data is put out on **PORTB**, only 7 bits are used, I would suggest using an 8bit or, ideally, a 12Bit DAC and hooking the lower seven bits of PORTB to the UPPER 7 bits of the DAC, and putting GND on the lower bits of the DAC, this ensures any non-linearities in the DAC are minimized.
Gate data (on or off) is put on the **MSB** of **PORTB** and is active high.

Velocity Data is on **PORTA**

The code is commented on virtually every line, ensuring that it should be easy to read and understand.

The basic flow is like this ;-

after initial setting up the code sits waiting for an interrupt from the UART (WFF loop). Once received the code then first checks to see if its a status byte (bit 7 of data is set) If it was then we check to see if the value is greater than or equal to 0xF0, If its is greater than or equal to 0xF0 then we simply ignore it this insures that we skip running status bytes, midi clock and so on. Notice that if it is we don't rest or touch *ANYTHING*, we just simply ignore it and go back to the WFF loop Assuming its less than 0xF0 we then check for NOTEONBYTE or NOTEOFFBYTE,We then remember this in the 'recstat' register for future use. If its NOT a status byte then it must be data, so we now use the data first we check what the running status byte was (RS:) If it was NOTEONBYTE then we jump forward and start processing the data If it was NOT a NOTEOFFBYTE then we don't need to reset running status back to 0 and the 'noteon' Next we check to see if we've previously had note data If not, then we save it as note data and increase 'noteon' If we have then the data is velocity data. Now we check to see if the velocity data is zero, if so, then we don't change the velocity data that's already on the PORT. If not then we set the GATE bit on, remember the note data and output the velocity If it is zero we check to make sure the note data matches the note we're playing, if it does we turn off the gate.

This gives a basic idea of the flow of the code without going into to much detail, like I say the code is pretty well commented and if you're unsure then just follow the flow of the code and it should become clear.

Useful links

http://www.midi.org/ - The ofiicial page for the Midi Manufacturers Association, you can buy the Official MIDI spec book from here.

http://www.borg.com/~jglatt/tech/midispec.htm - Great pages, nicely laid out and very extensive.

http://www.popeye-x.com/tech/midi_spec.htm - Good page, and has plenty of information about RPN/NRPN use.

http://www.indiana.edu/%7Eemusic/hertz.htm - Very usefull if your planning on building a synthesiser with MIDI built in

These are just some of the many that are around. Just use your favourite search engine and type 'midi Spec' or 'midi information' and you'll be gauranteed to find lots of pages with usefull information on.