VMM Virtual MIDI Machine - Manual

Mark Meeus, Sven Hermans
http://vmm.audionetwork.be/

Table of Contents

VMM - Virtual Midi Machine

VMM - Manual

Last update : 03-10-2006
contact

Table Of Contents

Introduction

Introduction

VMM Virtual MIDI Machine, is a MIDI (Musical Instruments Digital Interface) programming language written by Mark Meeus, in Visual C++. The main purpose, is to provide the possibility to program algorithms, that control MIDI hardware and or software.

VMM is freeware and used at own risk. The creator of VMM, and me, the author of this manual, are in no way responsible to where you (ab)use VMM for. If you destroy your computer, it is your own fault (or another, but not VMM's) :-) . By using VMM, you agree with this. The truth is: If you are going to do some MIDI routing or programming without care, MIDI feedback could freeze your computer or applications. Make sure you save everything a lot.

About this manual

Welcome to the VMM manual. This manual has the purpose to give you, the reader, a comprehensive background knowledge, to start programming and coding in VMM: Virtual Midi Machine. It doesn't include information about programming in general. There are plenty of excellent tutorials and books about that.

Because VMM was, is, and remains in progress: this manual is always incomplete. If you have comments, found a spelling error or anything you want to mention, contact us at http://vmm.audionetwork.be/contact/.
If VMM crashes we are always interested in the code what caused that, so don't hesitate to e-mail it.

The official and most recent version of the VMM manual and software can be found at:
http://vmm.audionetwork.be/

VMM Features

- VMM Compiler
- VMM Runtime
- MIDI Input and Output
- Total control over raw MIDI data
- Configurable MIDI port settings
- Debug Information
- Include Path Folder Settings
- Function libraries with source code
- C style syntax
- Progress bar
- Error reporting with highlighting

Requirements

This is a list of hardware and software minimum requirements:

- Microsoft Windows95/98 or Windows2000/XP
- MIDI Interface or a Virtual MIDI Router.
- Soundcard
- VMM Compiler
- needed dll libraries
- VMM Runtime

VMM is compatible with any Microsoft Windows product except 3.1. It has been succesfully tested on Windows98SE, 2000 and XP.
However, the recommended Operating System to use is Windows XP. Mainly, because WindowsXP has some needed dll files already included. This possibly saves you some installation trouble. If you think you have a compatibility problem on your Operating System, let us know.

A soundcard. Any SBLive soundcard is recommended for most users.

You will need a MIDI input and output. If you don't use external equipement and therefore haven't got a MIDI Interface, you can simulate this by using a MIDI router.

- win95/98 : Hubi's Loopback Device or MIDI Yoke.
- winXP/2000 : MIDI Yoke.

Note that in conjunction of MIDI Yoke, you will need MIDIOX. This is highly recommended because it gives you multi-client options and much more.

- Yoke/OX : midi-ox
- Hubi's : Hubi's Loopback Device

The VMM Compiler is a programming editor, where you type your source code, save it, and eventually, compile it. This will create a VMM excecutable file.

The needed DLL libraries are mfc42do.dll and mfc42d.dll and can be found here:
http://plaza.harmonix.ne.jp/~tosiwata/dll/download.html

The VMM Runtime is needed and used to run/play your compiled program.

Download VMM

Download Virtual MIDI Machine
Use winrar or winace to extract the file.

Installation

First of all, you will have to download VMM*.rar (or .zip) from the download section. The most recent version is always named VMM.rar. There might be older versions too, that can be recognized by version in the form vmm-n.n.n.zip where n is a number.

After you have downloaded VMM. Extract it to the directory where you want to install VMM. Any directory will do. If you don't know how to extract a .rar file, download winrar at http://www.rarlab.com/.

Now, make sure you have correctly installed the needed libraries. To avoid multiple programs using the same dll at the same time, it's best that you put them in the directory where you have VMM installed. eg. C:\VMM\ . If the files aren't allready there, put the dll's in your %systemroot%\system32\ directory (2000/XP). On a windows98 computer this would be c:\windows\system\ .

The VMM compiler can be started by clicking vmm.exe. If you don't get any errors, you can start coding. You can listen to your result, by playing the generated file in the VMM runtime, vmmrt.exe.

IMPORTANT: If you want to use VMM's MIDI functions, you will have to set the path to your include folder. Do this by clicking Options - Settings in the compiler and set the path to c:\vmm\include\ (if

you would have installed VMM in c:\vmm\). Note the "\" (backslash) at the end of your path.

To upgrade VMM, no uninstall is needed. You can overwrite your files, or install the new VMM in another directory and run multiple VMM versions on one machine.

VMM Syntax

The VMM syntax looks a lot at C syntax. Most of the time they are not identical, however, many similarities can be noticed.

The program itself always starts with:

```
proc main()
{

}
```

A VMM excecutable can not be succesfully compiled without the proc main. Everything between the {ldelim} {rdelim} will be excecuted. If you write a function outside the brackets, but don't call it within the proc main, nothing will happen.

All expressions must end with a ";".

Comments are started with "//". Everything what follows on the same line will be ignored. Block comments can be used this way:

```
/* This is a block comment */
```

Both methods can be used at any place in your program.

Data Types

There is only one type available: numeric, and can be an integer or decimal.

Variables

Variables have to be declared before they can be used. This _must_ be done in the beginning of your program and may not be declared after any statement. If there is no value given to a variable, the variable remains unknown. The name of a variable can be both upper and lower case. A variable may not start with a number.
Underscores are allowed.

Global Variables

```
global <name>;
global var;  // defines the variable "var" as global.
global MyVar; // is also correct.
global 2test; // is wrong.
```

Local Variables

```
<name>;

var2test; // declares "var2test" as a local variable.
a;  // a variable can be a single letter.
b;c;d;e;  // and they can be more than one declared on the same line.
AndTheyCanBeLongToo;
```

To assign a value to a variable:

```
<name> = <value>;
a = 10; // assign 10 to a.
```

```
var = 5;  // assign 5 to var.
newvar = oldvar; // would copy oldvar to newvar.

<value> can be a number, or another variable.
```

Arrays

VMM provides the possibility to use arrays. An array, can have infinite dimensions. Similar to a variable, an array also has to be declared (in the beginning of your program), before it can be used.

```
<MyArray> [ <n elements> ];
```

"MyArray" can be any word upper and or lower case.
"n elements" must be a number that defines the total elements in MyArray. Note that this number can be a result from a function, another variable or array, etc.
Thus, declaring array[10]; means there are 10 elements available in array.
The array index starts counting from zero. This means that to fill the whole array, you have to use the range from array[0] till array[9].

```
/* syntax */
ThisArray[6]; // declares "ThisArray".
              // "ThisArray" has 6 elements. [0-5]

/* example 1 */
  arr[2];
  arr[2] = 5; // gives an Error: Out of Range

/* example 2 */
arr[2];
arr[0] = 10; // is correct.
arr[1] = 25; // arr has 2 elements with the values 10 and 25.

/* example 3 */
notes[5];
notes[0] = 64;
notes[1] = 72;
notes[2] = 56;
notes[4] = 80;
/*
Debugging all keys in this array will have the result:
ThreadId : 1 => 64.000000
ThreadId : 1 => 72.000000
ThreadId : 1 => 56.000000
ThreadId : 1 => 0.000000
ThreadId : 1 => 80.000000
*/
```

Global Arrays

```
global <GlobArr> [ <total elements> ];
```

Global arrays have the same syntax as local arrays. Except global arrays are used in the global scope of a program.

Multi Dimensional Arrays

```
<MultiArr> [ <n elements> , <x elements> ];
```

MultiArr is a 2 dimensional array. "n elements" are the number of elements in the first dimension. The dimensions are seperated by a comma ",".

"x elements" declares the second dimension.

```
MultiArr[2,3,1];
/*
MultiArr is a three dimensional array.
The first dimension has 2 elements.
The second dimension has 3 elements.
The third dimension has 1 element.
*/

MultiArr[0,0,0] = 30;
// Would assign 30 to the first possible key in the array.

MultiArr[1,2,0] = 64;
// Would assign 64 to the last possible key in the array.

MultiArr[1,2,123] = 64;
// Would result in the Error: Out of range.
```

To assign a value to an array key element:

```
TwoDim[3,3]; // first declare "TwoDim"

// then assign a value to each key

<MultiArr> [ <key1>, <key2> ] = <value>;

TwoDim[0,0] = 20;
TwoDim[0,1] = 30;
TwoDim[0,2] = 45;
TwoDim[1,0] = 12;
TwoDim[1,1] = 65;
TwoDim[1,2] = 78;
TwoDim[2,0] = 64;
TwoDim[2,1] = 19;
TwoDim[2,2] = 38;
```

In a graphical table, it would look like this:

```
   |0 |1 |2 |
 -|--|--|--|-> key1
0 |20|12|64|
 -|--|--|--|-
1 |30|65|19|
 -|--|--|--|-
2 |45|78|38|
 -|--|--|--|-
key2
```

(note: In this figure you could also switch key1 and key2's X and Y directions, on condition that you also change the resulting values.)

But, VMM goes much further than that. These examples are also correct:

```
var[variable];
a[foo()];
MyArray[(var + bar())+10];
notes[a[1,2],(b+24)*test(64)];
```

Important: if you declare a variable again, all previous information of that variable in memory is cleared.

Operators

Mathematical functions in VMM:

```
+   plus
-   minus
%   modulo
=   equals
/   divide
++  add by one
--  substract by one
```

Control - Structures

if

```
if (expression) { do something }
```

Optional, you can make use of "else" in the statement.

```
if (expression) { do something } else { do something else }
```

loop

To make a loop in VMM:

```
loop{ }
```

If you want to break a loop, you can use a condition what returns true or false, followed by:

```
exit;
```

eg.

```
i; i = 1;
loop{
  do somethings...
  if(i > 10){
    exit;
  }
  i = i + 1;
  // would loop somethings 10 times, and exit.
}
```

while

Identical in purpose but different in syntax from loop(), you can use while().

```
while ( <condition> ){
  do something
}
```

Expressions

An expression can be anything that returns a value, function, variable. Control structures can be:

```
== equals
<  smaller
> greater
```

```
<= smaller or equal
>= greater or equal
|| OR
&& AND
!= NOT



var;
var = 20;
if (var == 10) { } // returns FALSE
if (var == 20) { } // returns TRUE
```

NOTE: In previous versions there wasn't a NOT "!=" operator.
Of course you are encouraged to use the latest VMM release, however, there is a way around this:

```
proc main()
{
a;b;true;false;
a = 10;
b = 20;
true = 1;
false = 0;
 if(0 == (a <b)) // if a is not smaller than b..
 {
  debug(true);
 }else{
  debug(false);
 }
}

// returns false because a _is_ smaller than b
// ThreadId : 1 => 0.000000

// note:
// if(0 == (a  <b))
// is exactly the same as
// if (a > b)
```

Functions - Procedures

```
proc MyFunction ( <parameter> ) { do something }
```

The function or procedure is started by "proc", followed by a "name". There can be no, one, or more parameters. More than one parameter must be seperated by a comma.

To break a function:

```
return(var);
```

In the example, MyFunction has to be called"MyFunction(somevalue)", before it will produce any output.

A simple function could look like this:

```
proc a(var){
  b;
  b = var + 3;
  return(b);
}

  a(30); // returns 33
```

Of course, VMM has also functions all ready build in.

Currently, the core functions you can use are:

```
out();
sleep();
rand();
srand();
critical();
in();
debug();
int();
GetSystemTime();
```

These functions are expandable by your VMM Function Library. You can create your own, or use the default VMM MIDI function Library. More information is in the MIDI Functions section.

out

To send data to a midi output, use the "out()" function. Between the brackets come the bytes you want to send to your MIDI hard or software. The bytes are interpreted as hexadecimal values.

```
out ( <portnumber> <amount> <n_nr> <n_kind> );
```

That means, if you would like to send a note event, it can be explained like this:

```
out ( <portnumber> <velocity> <pitch> <note> );
```

or a controller event:

```
out ( <portnumber> <amount> <ctrl_nr> <control> );
```

or a shorter alternative:

```
out ( <message> );

// eg.:
proc main()
{
  out(0x007F3C90);
}
```

Would send a note event on MIDI channel 1 with a pitch of 60 and a velocity of 127.

- 0x defines the use of HEX values.
- 7F is volume in HEX (127 in decimal)
- 3C is pitch. (60 in decimal)
- 90 indicates a note event on channel 1 (144 in decimal)

sleep

```
sleep ( <time> );
```

<time> can be anything that returns a value. (in milliseconds).

sleep(10); // delays the (further) execution of your program with 10 milliseconds.

Of course, time can be variable:

```
var;
var = 10;
sleep(var);

var = var + 5;
sleep(var); // "var" will be 15 here.
```

rand

```
rand();
```

rand(); returns a value.

```
pitch; pitch = rand(); // makes "pitch" random.
pitch = (pitch % 50); // modulo from pitch divided by 50.
```

In this example, pitch is always bigger than 0 and smaller than 50.

srand

```
srand ( <parameter> );
```

srand() is used to initialise the random generator. Random generators calculate their first value from a seed. The default seed sits somewhere in the system, but you can add one by yourself using srand().

critical

```
critical();
```

Everything between a "critical" function gets executed before anything else.

in

"in" reads MIDI input. This input can be used as realtime variable, or send to another output. It is read as a global variable, therefore it can be used in the whole script.

```
in();
```

```
proc main()
{
  msg;
  loop{
    msg = in(); // "msg" is input
    out ( msg + ((rand()%3) * 0x100));
  }
}
```

The pitch on the output becomes a random value 3 semitones higher or lower than the pitch on the input.

debug

In VMM Runtime, you can debug the value of a variable. Its current value will be displayed in the VMM Debug Window.

```
debug();
```

```
proc main()
{
  a;b;c;
  a = 13;
  b = 12;
  c = a + b;
  debug(c);
}

ThreadId : 1 => 25.000000
```

Another example:

```
proc main()
{
  a;b;
  a = 7;
  b = 7;
    if(a>=b){
      debug(a);
    }
}
```

will show:
ThreadId : 1 => 8.000000
in the debug window.
If "a" would be 5 (and smaller then 7), it will show nothing.

int

```
int();
```

The int function rounds a value to an integer number. It is only supported when you make use of the default VMM library stdMidi.vmm.

```
#include "stdMidi.vmm"
proc main()
{
  n;
  n = 7/4;
  n = int(n);
  debug(n);
}

ThreadId : 1 => 1.000000
```

Look at the difference when int isn't included:

```
#include "stdMidi.vmm"
proc main()
{
  n;
  n = 7/4;
  debug(n);
}

ThreadId : 1 => 1.750000
```

NOTE: This is important when doing mathematical functions on MIDI Data!

GetSystemTime

```
GetSystemTime();
```

GetSystemTime gets the computer system time.
It starts counting at midnight from 0, and the value it returns is in seconds.

```
// when it is 01:00am, and you do
debug(GetSystemTime());
// the value it returns is:
ThreadId : 1 => 3600.000000
```

MIDI Functions

To make VMM a bit more user friendly, MIDI Channel Messages are included for you, and all have their own Functions. They are free to use, deleted or changed. But keep in mind that when you make your own MIDI commands, other users cannot help you out of trouble. Above that, when you don't include your own library, they are not possible to run your programs. To avoid confusing syntax and code, please always use the default MIDI commands.

These are:

- NoteOn()
- NoteOff()
- Controller()
- PitchBend()
- ProgramChange()

AfterTouch() is not implemented yet.
System Common, System Real Time, and System Exclusive functions aren't supported in the default library. But if you really need them you can create your own.

ALL MIDI data is send to the MIDI output port you activate in Options - Settings in VMM Runtime.

#include

To make VMM's default MIDI functions available, you must have the correct include file in your script, before one of the MIDI functions gets called. It's recommended to start every program you write, with the include file that you want to use.

The default MIDI functions are in the file stdMidi.vmm in your %vmm%\include\ directory. Be sure to set the path in Options - Settings to the right path as described in the Installation part of this manual.

syntax:

```
#include "stdMidi.vmm"
```

NOTE: there may NOT be a ";" at the end. You may not use comments in stdmidi.vmm.

NoteOn

NoteOn is the command that sends a NOTE ON. All flags are needed. A NOTE ON in MIDI language always comes with some bytes that are needed to play a note. These paramaters are also needed when playing traditional instruments so there is nothing weird about that. The only difference is that you must define a MIDI channel. And that's good, because you can use 16 channels, that means 16 times more fun. The other parameters to play a NOTE ON are PITCH (the frequency of the note), and VELOCITY (the volume of the note).

NoteOn();

syntax:

```
NoteOn(<Channel>,<Pitch>,<Velocity>);
```

The values must be in these ranges:

```
<Channel>   1 - 16
```

```
<Pitch>     0 - 127
<Velocity>  0 - 127
```

So, if you want to play a note with pitch C3 (Middle C) with full velocity on channel 1:

```
NoteOn(1,60,127);
```

OK that's fun, but now the NOTE ON doesn't stop. To make it stop use NoteOff();

To make the note play for one second, use sleep() between the NOTE ON and OFF;

```
NoteOn(1,64,127);
sleep(1000);
NoteOff(1,64);
```

NoteOff

To stop a note from playing use a NOTE OFF. This is similar to the NoteOn() function. Actually a NOTE OFF in MIDI, is a NOTE ON but with a Velocity of 0. Notice that the NOTE OFF pitch value must be the same as the NOTE ON pitch value, else you will stop a note that even wasn't playing.

```
NoteOff();
```

```
NoteOff(<Channel>,<Pitch>);
NoteOn(1,64,127);
sleep(1000);
NoteOff(1,64);
```

That would be the same as:

```
NoteOn(1,64,127);
sleep(1000);
NoteOn(1,64,0);
```

Of course, the first example, with NoteOff, is the most obvious to use. There are other ways to send a NOTE OFF (with a release velocity) but that is not implemented at the moment. If you want a NOTE OFF function with release velocity you create one yourself.

Controller

With VMM you can send Control Change Message data to every parameter you want.

Controller();

syntax:

```
Controller(<Channel>,<ControllerNumber>,<Value>);
```

Controller(1,10,127);

Is a PAN (ctrl.10) controller event, on channel 1, with a value of 127. That means the sounds would be fully panned to the right.
For a list of all controller numbers and their purpose look here.

PitchBend

With a pitchbend you can change pitch with fine or coarse resolution.

PitchBend();

syntax:

```
PitchBend(<Channel>,<Range>,<Amount>);
```

eg.: PitchBend(1,64,50);

ProgramChange

A Program Change changes the current patch in a bank to the new value.
Most synthesizers and samplers have 127 programs in one bank.

ProgramChange();

```
ProgramChange(<Channel>,<Program>);
```

ProgramChange(1,1);

On a General MIDI synthesizer this would set the sound on MIDI Channel 1 to patch/program 1 (piano).

Thread - MultiTask

VMM can run different tasks at the same time. These tasks in VMM are called "threads". A thread can have it's own functions. You can run as much threads you want!

```
thread <function> ( <parameters> ) { }
```

VMM Runtime

The VMM Runtime is the program where you load your compiled executables. VMM executables have the extension ".vmx".
After setting the MIDI OUT port, you can "play" your code and actually listen to what you have coded.
(if you like to test an example script, you can use, play, and modify test.vmx from the scripts directory).
If you hear sound, congratulations :-)
If you here silence, make sure your MIDI port settings are correct.
If everything fails, make sure you can play MIDI at all.

VMM Networking

To call a function externally use "extern".

```
extern MyFuntion(<value>, [valueN]){
  ...
}
```

Read more in the VMM Networking section. You can also download vmm_networking.pdf 314KB.

Examples

This is an example of a raw MIDI message, note on and off. A short delay of 500ms between the 2 events is inserted.

```
proc main()
{
  out(0x007F3b90); // note on chan1
  sleep(500); // sleep 500
  out(0x007F3b80); // note off chan1
}

Is the same as:

proc main()
{
  note_on;
  note_off;

  note_on = 0x007F3b90;
  note_off = 0x00003b80;

  out(note_on);
  sleep(500);
  out(note_off);
}


Then a dump in MIDI-OX looks like:
```

```
PORT STATUS DATA1 DATA2 CHAN NOTE EVENT
  1     90    3B    7F    1  B 3  Note On
  1     80    3B    00    1  B 3  Note Off
```

I hope this has helped to get you starting with the Virtual MIDI Machine.
If you have further questions, comments, tips, corrections, ..., contact us
VMM Team

vmm.audionetwork.be 2006 .print.