



**Audio Tools
for
Nintendo®64 and Game Boy®**

D.C.N. NOA-06-8207-001 REV A

"Confidential"

This document contains confidential and proprietary information of FACTOR 5 LLC and Nintendo of America Inc. and is also protected under the copyright laws of the United States and foreign countries. No part of this document may be released, distributed, transmitted or reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from Nintendo.

© 1999 FACTOR 5 LLC

Printed and distributed by Nintendo of America Inc., August, 1999

MusyX, and Factor5 are registered trademarks of Factor 5, LLC.

Nintendo, Game Boy, and N64 are registered trademarks of Nintendo of America Inc.

Dolby is a registered trademark of Dolby Laboratories, Inc.

Introduction

Sequenced vs. Streaming

In an age of gigabyte sized multimedia applications, memory is no longer the limiting factor it once was. Streamed audio has become a de facto standard for most games and multimedia applications. It seems to overcome the limited possibilities associated with sequenced sound and, from the perspective of the musician, it allows studio quality productions to be used for the first time. There really seems to be no reason why one should even consider using sequenced sounds instead of digital audio streams.

The developers of **MusyX** have been working in the games industry since the 80's and therefore experienced the shift from sequenced sound to streamed audio first hand. At first glance streamed audio is tempting, but there are some major drawbacks.

- Even on CDs, memory is not totally unlimited. 650 MB seems to be a lot of memory, but even when using ADPCM compressed audio you cannot get more than 4 hours worth of audio data on a single CD. Take into consideration that program and graphics data need to be stored somewhere, too.
- Since a CD is not always the most reliable type of media, huge data buffers in memory are needed to assure that no breakups in the audio stream occur.
- Streamed audio can hardly be called truly interactive, since locating different places on a CD, when looking for specific data, results in delays.
- Crossfading between multiple pieces of music is almost impossible without major programming effort.
- Accessing a CD-Drive causes major slowdowns in game applications on most systems.
- There are still cartridge-based platforms out there, where streaming audio is not an option.

Since today's games more and more resemble little movies with a quality you would expect from big screen films, having fast, interactive sound is crucial for a smooth, real-time playing experience. The score has to react to the action when it takes place, without delay.

It therefore does not make much sense to use streamed audio, the disadvantage of which is obvious: Noticeable delays, when it is supposed to react instantly to sudden shifts in the action, which disturbs the smooth progression of the game play.

The Advantages of **MusyX** for Musicians

MusyX is designed to be used like any normal synthesizer during development. The only difference is that a Windows application is used to edit all the parameters of the synthesizer.

All the musician needs is a standard keyboard and the sequencer of his choice. A slave program running either on the same, or a separate computer, represents the MIDI synthesizer he would normally use, and emulates the sound of the target system.

MusyX actually gives the musician greater freedom because sounds are not predetermined and fixed, like in a normal General MIDI synthesizer. Therefore he can create his own sounds, use his own samples or rearrange the sounds he wants to use.

MusyX works with

Keymaps and layers	to build complex sounds and sound sets
SoundMacros	to create truly unique sounds

while a convenient grouping arrangement allows the musician to manage songs and sound effects effectively.

Sequenced sound does not need to sound sterile or artificial. With **MusyX** the musician has the power and necessary control to create vibrant, original sound with the greatest possible ease.

Screen Credits

All developers who incorporate the **MusyX** audio system into their game are required to make their best efforts to display a screen credit on the opening licensing screen, which states the following.

" **MusyX** Audio Tools Licensed by Factor 5"

Using Dolby Surround Logo

A developer may become a Dolby Surround Licensee, free of charge. This grants the right to use the Dolby Surround Logo. To become a Dolby Surround Licensee, four basic criteria need to be met.

- 1 The developer must use a Dolby-approved Phase Positioner, such as **MusyX**, or other Dolby-approved real-time encoding scheme for positioning sound effects that are triggered in real-time.
- 2 If the game includes linear audio, the developer must use a Dolby-approved Surround facility to encode linear audio.

- 3 The developer must sign and complete a Dolby Surround Trademark Agreement which grants a license to use the Dolby Surround trademark, subject to these conditions.
- 4 The developer must send a sample of the finished title at or prior to the time of release.

For more information on becoming a Surround licensee, please contact Dolby directly:

Dolby Laboratories, Inc.
100 Potrero Avenue
San Francisco, CA 94103-4813
Tel: (415) 558-0200
Fax: (415) 863-1373
Email: multimedia@dolby.com

Table of Contents

Introduction.....	3
Sequenced vs. Streaming	3
The Advantages of <i>MusyX</i> for Musicians	4
Screen Credits	4
Using Dolby Surround Logo	4
Working with <i>MusyX</i>.....	11
SoundMacros.....	11
Macrolanguage vs. Parameter Setup	12
Layers and Keymaps	12
Groups- Organizing Data	14
Dynamic Voice Allocation.....	14
Game Applications.....	15
Data Conversion	15
The Installation Process	17
Hardware Setup.....	17
Installing <i>MusyX</i> on Nintendo 64	19
System Requirements for Dual System Setup	21
Installing <i>MusyX</i> on Game Boy	24
System Requirements for Dual System Setup	26
MIDI Loopback Devices	28
Changing General Settings	29
Software Start-up Routine.....	31
Overview.....	33
How <i>MusyX</i> Works	33
SoundMacros and the SoundMacro Editor	35
General.....	35
Creating a SoundMacro	36
Creating SoundMacros by Using Templates	37
Editing SoundMacros	37
The Command Pool	38
Editing Values.....	39
Loops and Jumps in SMaL.....	40
Calling other SoundMacros in SMaL.....	40
Keymaps and Layers.....	41
General.....	41
Keymaps.....	42
Layers.....	44

The Sound Editor.....	49
What is a Sound?	49
Defining Sounds and Import/Export	49
The Sound Object Properties Window	51
The Search Window.....	53
 Organizing Data.....	 55
General.....	55
Managing Groups	59
SongGroups and Their Parameters	60
SFXgroups and Their Parameters.....	62
Testing Sound Effects.....	63
Managing the Object Pool.....	65
Adding Samples.....	65
The Different Parts of a Project	66
How the Structure Relates to the Actual Game Data.....	67
Transferring Data Between Projects	67
 The Project Manager	 69
General.....	69
Project Manager Menus.....	70
 Walk Through.....	 75
General.....	75
Start the Soundslave.....	76
Launch the <i>MusyX</i> Editor.....	76
Creating a New Project	77
The Project Window.....	79
Adding Samples.....	80
A Very Simple SoundMacro (SMaL Program)	81
Defining a SongGroup	83
Playing the Instrument for the First Time.....	85
Recording a Sequence	86
Looped Midi-files.....	87
Defining a Sound Effect	88
How to Test a Sound Effect	89
Saving Your Work.....	89
Finish!	89
 Additional Tools	 91
The Table Editor	91
Using the Table Editor	91
Using the ADSR Envelope Editor.....	92

The MIDI Setup Window	93
What is the Purpose of this Window?	93
How to Use the Window in Everyday Work	93
The Importance of the Window when Exporting Data.....	93
Using Multiple MIDI Setups within One SongGroup	94
The Virtual MIDI Keyboard.....	95
Why a Virtual Keyboard?	95
Using the Keyboard	95
Testing Sound Effects.....	96
Limitations in Comparison to a Real Keyboard.....	96
The Network Master Window.....	97
What can be Controlled Using this Window?.....	97
What Kind of Information is Displayed?.....	97
Data Conversion	99
General.....	99
What the Musician has to do to Prepare the Data	100
The Actual Data Conversion	101
The Description File	103
<i>MusyX</i> Sample Program for N64	106
Appendix 1 - N64 Musicians Reference.....	107
Appendix 2 - Game Boy Musicians Reference.....	177
Appendix 3 - N64 Programmers Reference	255
Appendix 4 - Game Boy Programmers Reference.....	345
Appendix 5 - Slave Reverb Control (N64)	385
MINI-M.O.R.T.	391

Working with ***MusyX***

SoundMacros

The element that makes ***MusyX*** so powerful is the SoundMacro.

SoundMacros are basically small and simple programs used to define both instruments and sound effects within ***MusyX***. SoundMacros are created using a customized programming language called SMaL (Sound Macro Language).

Each line of the SoundMacro program constitutes a single MacroStep command, most of which are executed sequentially and define various attributes of the sound. Other commands control the flow of the program and make it possible to create loops or conditional jumps in the program.

Something every musician wants to be able to do is to create full, complex sounds with minimal effort, which cannot be achieved by using a static set of parameters. ***MusyX*** on the other hand offers a dynamic algorithm and allows the musician to do just that. SoundMacros offer all necessary means to gradually influence sounds as they move forward in time.

A great advantage of SMaL programs is their simplicity. It is not necessary to be a professional programmer in order to design a SMaL program. This programming language is easy to learn. It is therefore easy to quickly gain maximum control over the generated sounds.

To make working with ***MusyX*** even easier, it comes with library features like predefined SoundMacro programs. The musician can simply use these or edit them. With customized instruments defined by using SMaL, it is easy to make every piece of music sound truly unique.

Sound effects are also created using SoundMacros, allowing complete control over the effect without having to sample every variation required by the sound designer.

Macrolanguage vs. Parameter Setup

All synthesizers and samplers use parameters to make modifying sounds easier, but in a game application the amount of data for the entirety of all parameters is huge. Since every single parameter needs to be processed, it would take too much CPU time to work with a parameter setup.

With the macrolanguage in **MusyX**, users choose from the beginning only those features and parameters they need to modify for each individual sound. This eliminates the need for setting and resetting unused or unneeded parameters, and gains valuable CPU time because the CPU does not have to process unnecessary data.

When working with a standard synthesizer, the musician sets specific parameters at the beginning, and is not able to change them during the course of the song or sound. With the macrolanguage of **MusyX**, the user now has the tool he needs to change the parameters of a sound while it progresses in time, starting off with one set of parameters and concluding the sound with a different set.

Layers and Keymaps

In the same context where a Macro is the next larger structure over a sample, Keymaps and Layers are above Macros.

Their design makes keymaps and layers an important tool for the musician to organize his data.

In the keymap editor, the musician allocates Macros to keys of the keyboard, one macro per MIDI key number (1-128). This way he can reference up to 128 sounds (i.e. Macros) in one keymap. The keys that are not defined by the user remain empty.

If the user wishes to do so, he can allocate layers or even other keymaps instead of Macros.

Keymaps are perfect for creating drum sets or other complex arrangements.

In comparison to a keymap, a layer is more powerful.

In the layer editor, the musician also allocates Macros to keys of the keyboard. One of the main differences is that he can reference more than one key per Macro. He can specify whole key ranges, where the same Macro is supposed to play.

For the next step, he might now want to specify a key range for a different Macro, this key range overlapping the previously mentioned one. This way, whenever one of the keys that are allocated to both Macros at the same time is played, both Macros will play – at the same time.

The other difference is that the size of a layer varies, whereas the size of a keymap is fixed by design.

Since the musician can now work with overlapping key ranges when he uses a layer and is able to play several Macros simultaneously, layers are the perfect tools for building multi-sample instruments.

Groups- Organizing Data

Data within the system is stored in either SongGroups or FXGroups, depending on whether it is used for music or for Sound effects. Both groups contain references to all objects used in that Group, like Macros, layers, keymaps, samples, and tables.

Data within the system is stored in either SongGroups or FXGroups, depending on whether it is a whole song or a sound effect. Both groups contain references to all instruments used, and to all of the macros, samples, and other data used to create or edit these instruments.

In addition, SongGroups contain references to arrangements and General MIDI information, while FXGroups store information related to sound effects.

Since the amount of memory used for sound tends to be limited, working with groups is an effective way to manage the data. Depending on the needs of the game, groups can be moved in or out of memory quickly and easily.

Dynamic Voice Allocation

In order to be played, instruments and sound effects need to be assigned a hardware voice. As the amount of voices is limited, **MusyX** uses an intelligent mechanism to allocate voices on the basis of priority and age. The system first searches for free voices and allocates any that are found. If no free voices are available, the system searches for the voice occupied by the lowest priority Sound.

If two voices share the same ‘lowest’ priority, allocation is based on age. **MusyX** contains an ‘age counter’ that decreases over time to determine age. The user can control and change both priority and age by assigning a priority to instruments, or by manipulating the age counter with MacroStep commands.

New allocations occur only if free voices are available or if the new instrument has an equal or greater priority than an instrument currently playing. If neither is the case, the new instrument cannot be allocated.

Game Applications

For game applications, ***MusyX*** offers a full-blown sound effect API including all the functions a programmer could possibly want. In a 3D game, for example, the programmer simply places the sound effect somewhere in 3D-space and the system takes care of all volume, panning, or surround sound changes that occur when the listener is changing his position in the environment of the game. For the programmer's convenience a low level API is also included in case he prefers to handle these higher level tasks on his own.

Data Conversion

With the tools ***MusyX*** provides, the programmer can quickly and easily transfer the generated data from his PC into the target platform's specific format. This gives the musician the ability to truly concentrate on the creative aspects of his work.

The Installation Process

Hardware Setup

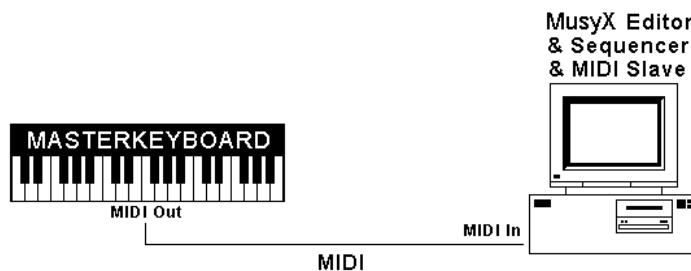
A Single System Setup is the standard configuration of **MusyX**. The connections must be as follows:

- Connect an external Keyboard's Midi-out to Midi-in of the computer.
- TCP/IP MUST be installed for the communication between **MusyX** Editor and MIDI-Slave. Although an external network connection is possible, it is not necessary.

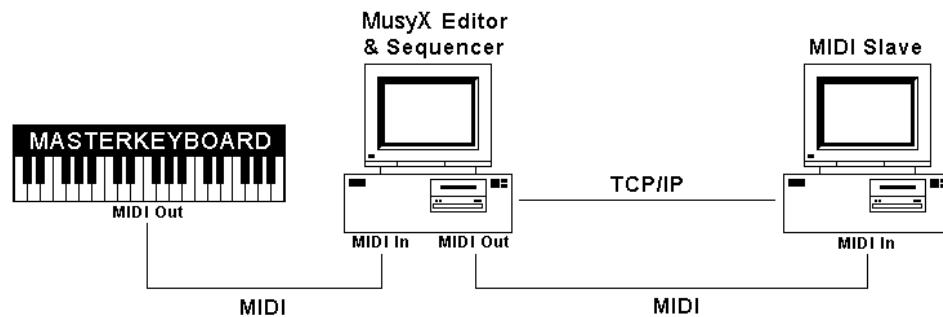
Although a One-Computer setup is the standard configuration, other configurations are possible. The three tools (**MusyX** editor, MIDI Slave and Sequencer) may run on three separate machines or in any combination on two machines. In all cases, the following connections must be maintained:

- Be sure to connect the Sequencer's Midi-out to Midi-in of the Slave. (*on a single system setup this will require the use of a "MIDI loopback device"*)
- Connect an external Keyboard's Midi-out to Midi-in of the Sequencer.
- The Midi-out from the slave is not used and can be left unconnected.
- TCP/IP MUST be installed for the communication between **MusyX** Editor and MIDI-Slave.

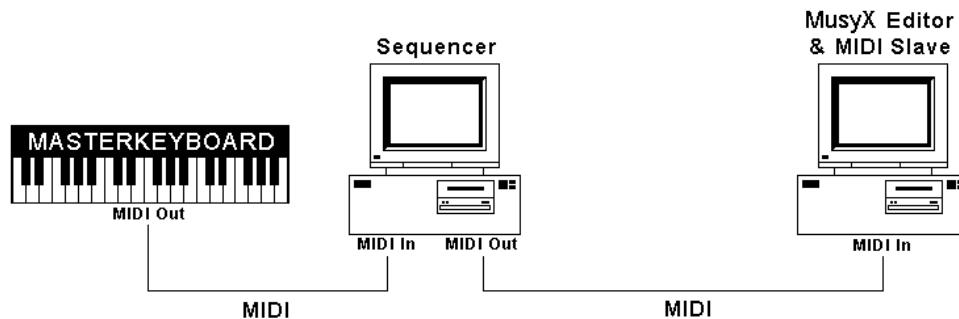
One Computer Setup:



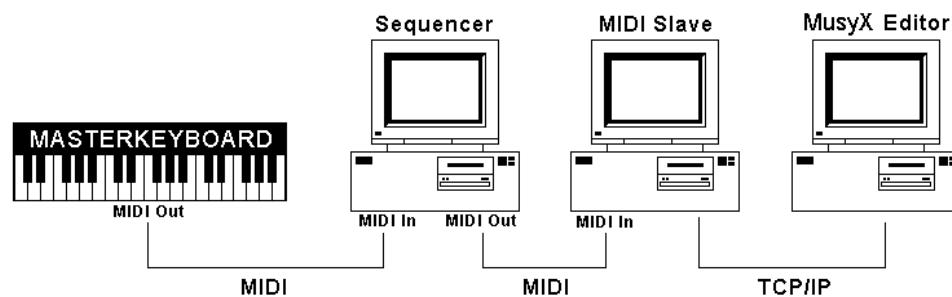
Two Computer Setup Version A:



Two Computer Setup Version B:



Three Computer Setup:



Installing *MusyX* on Nintendo 64

System Requirements

Windows 95/98 or
Windows NT 4.0 (Slave program requires Windows 95/98)
Pentium MMX 200Mhz (Pentium II 266Mhz or better recommended)
64Mbytes of RAM
800x600 resolution or higher
Winsocket & DirectX 3 or higher installed
16-bit sound card

How to Install the Software

Install all *MusyX* components from the CD via the SETUP.EXE program.

Consult the README.TXT file in the target directory of the installation for the newest updates and changes.

If you want to run your sequencer program on the same machine as the slave program, you should now install a MIDI loopback device, if you have not already installed one (See '**MIDI Loopback Devices**' on page 28, for details).

The Slave Configuration Tool

Start the slave configuration program, "MusyX Slave Configuration", which can be found in the MusyX Program folder. This program will let you configure all aspects of the slave's behavior.

The default settings assume a single system setup, which is the most likely configuration. You will need a dual system setup only when running the editor under Windows NT. So you will probably want to keep the link configuration the way it is (See '**Installing *MusyX* on a Dual System Setup**' on page 21, for details about the dual system setup).

Select the MIDI port you want to use to input MIDI data to the slave (e.g. a port provided by a MIDI loopback device).

In the sound settings you may select the master mixing frequency to be used (probably 22050Hz) and the default maximum number of voices available at one time. The latter can be changed at runtime, too.

The latency value determines the speed with which the slave program can react to your keyboard input via MIDI. The lower the number, the shorter the delay or latency. Since not all systems are reliable enough in respect to timing to allow for short latencies, you may be forced to use a higher latency value. If you experience any problems, start this program again and increase the latency value. (When setting the number of voices keep in mind that the N64's runtime library only supports a maximum of 32 voices, while the slave program supports up to 64 voices.)

Finally you have to decide whether to use 16-bit RAW samples or compressed samples with the slave. Compressed samples give you the advantage of a 100% accurate preview of the sound, so you can hear how it will sound when played on the N64. Compressed samples have been ADPCM compressed, then decompressed, in order to simulate the realtime decompression performed by the hardware. On the other hand, converting the samples will take some initial setup time.

The slave program uses a cache directory on the local hard drive to store compressed samples for later use. The user can define the size of this cache by setting the cache size value. We found that the difference in sound quality between raw (uncompressed) samples and compressed samples is minimal, and makes it therefore possible to mainly use raw samples. But we leave this decision up to the individual musician.

Exit the configuration tool by clicking on "Ok".

You now have finished the single system setup.

System Requirements for Dual System Setup

Master

Windows 95/98 or Windows NT 4.0
Pentium 200Mhz or better
32Mbytes of RAM
800x600 resolution or higher
Winsocket installed

Slave

Windows 95/98 (NT is not supported)
Pentium MMX 200Mhz (Pentium II 266Mhz or better recommended)
32Mbytes of RAM or better
800x600 resolution or higher
Winsocket & DirectX 3 or higher installed
16-bit sound card

Installing *MusyX* on a Dual System Setup

Install all *MusyX* components from the CD via the SETUP.EXE program.

Consult the README.TXT file in the target directory of the installation for the newest updates and changes.

Installing the Slave Program:

Install the *MusyX* "Musician's Tools" from the CD via the Setup program.
The other items are not required to be installed on the slave PC.

Configuring the Slave:

Start the slave configuration program, "MusyX Slave Configuration", which can be found in the MusyX Program folder. This program will let you configure all aspects of the slave's behavior.

The default settings assume a single system setup. To change to dual system setup simply tag the "dual system" checkbox. Normally it will not be necessary to deactivate the "automatic IP configuration". If you wish to do so, you will have to specify the local IP of the slave manually.

Select the MIDI port you want to use to input MIDI data to the slave (e.g. a port provided by a MIDI loopback device).

In the sound settings you may select the master mixing frequency to be used (probably 22050Hz) and the default maximum number of voices available at one time. The latter can be changed at runtime, too.

The latency value determines the speed with which the slave program can react to your keyboard input via MIDI. The lower the number, the shorter the delay or latency. Since not all systems are reliable enough in respect to timing to allow for short latencies, you may be forced to use a higher latency value. If you experience any problems start this program again and increase the latency value. (When setting the number of voices keep in mind that the N64's runtime library only supports a maximum of 32 voices, while the slave program supports up to 64 voices.)

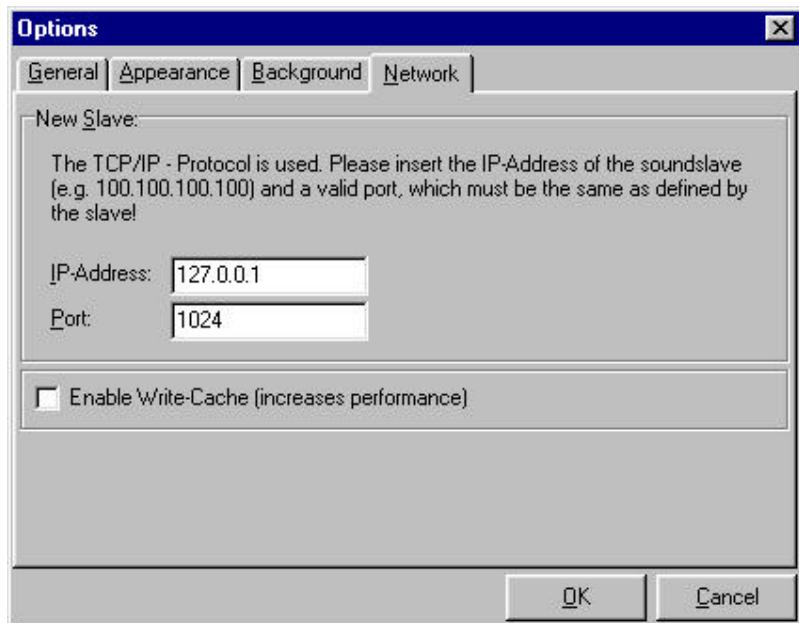
Finally you have to decide whether to use 16-bit RAW samples or compressed samples with the slave. Compressed samples give you the advantage of a 100% accurate preview of the sound, so you can hear how it will sound when played on the N64. Compressed samples have been ADPCM compressed, then decompressed, in order to simulate the realtime decompression performed by the hardware. On the other hand, converting the samples will take some initial setup time.

The slave program uses a cache directory on the local hard drive to store compressed samples for later use. The user can define the size of this cache by setting the cache size value. We found that the difference in sound quality between raw (uncompressed) samples and compressed samples is minimal, and makes it therefore possible to mainly use raw samples. But we leave this decision up to the individual musician.

Exit the configuration tool by clicking on "Ok".

Configuring the **MusyX** Editor:

Start the **MusyX** editor program and select "environment" from the "options" pull down menu. In the dialog box which appears, select the "network" panel.



The default configuration assumes a single system setup. Replace the 127.0.0.1 with the IP address of the system running the slave program. Make sure that the port is set to the same port as in the slave configuration program. (By default, they are both 1024.)

Exit the dialog by clicking on "Ok".

You now have finished the dual system setup.

Installing **MusyX** on Game Boy

System Requirements

Windows 95/98 or
Windows NT 4.0 (Slave program requires Windows 95/98)
Pentium MMX 200Mhz (Pentium II 266Mhz or better recommended)
64Mbytes of RAM
800x600 resolution or higher
Winsocket & DirectX 3 or higher installed
16-bit sound card

How to Install the Software

Install all **MusyX** components from the CD via the SETUP.EXE program.

Consult the README.TXT file in the target directory of the installation for the newest updates and changes.

If you want to run your sequencer program on the same machine as the slave program, you should now install a MIDI loopback device, if you have not already installed one (See '**MIDI Loopback Devices**' on page 28, for details).

Windows 95/98 and Windows NT4.0:

Connect your Game Boy Color with the supplied custom link cable to any unused parallel port of your PC. Configure the slave to use a loopback device.

Windows 95/98 ONLY:

Configure **MusyX** to use this parallel port (you need to know the port address of the parallel port you wish to use) using the Game Boy Data Link section of the Game Boy slave configuration.

Windows NT4.0 ONLY:

After installing all **MusyX** components onto your PC hard drive, you will find a folder "Windows NT Driver" in your installation directory. Contained therein is a device driver for Windows NT to allow access to the parallel port.

Copy the file GAMEBOY.SYS into the Windows NT system folder "system32\drivers".

To setup the driver for the port address of the parallel port you wish to use, double-click on one of the provided registry key files. Typical addresses are 0x3bc, 0x378 and 0x278. If you are unsure about which port address to use please contact your system administrator.

Next open the system control panel and double-click on "Devices". Locate the "parport" device and select it. Click on the "Startup" button and select "Disabled" from the list of startup options.

Close the control panel and reboot your machine.

Open the "Devices" control panel once more and verify that the new device "Gameboy" has a "started" status and an "automatic" startup state. If the startup state does not state "automatic" change it using the startup options as described above. If the service is not yet started, select it and press the "Start" button. If for some reason the device fails to start please contact your system administrator.

Configuring the Slave

Start the slave configuration program, "MusyX Game Boy Slave Configuration", which can be found in the program menu under the MusyX Game Boy folder. This program will let you configure all aspects of the slave's behavior.

The default settings assume a single system setup, which is the most likely configuration. So you will probably want to keep the link configuration the way it is (See '[Installing MusyX on a Dual System Setup](#)' on page 26, for details about the dual system setup).

Select the MIDI port you want to use to input MIDI data to the slave (e.g. a port provided by a MIDI loopback device).

Configure the parallel port that **MusyX** will use to communicate with Game Boy, by entering the parallel port address in the Game Boy Data Link section.

Exit the configuration tool by clicking on "Ok".

You now have finished the single system setup.

System Requirements for Dual System Setup

Master

Windows 95/98 or Windows NT 4.0
Pentium 200Mhz or better
32Mbytes of RAM
800x600 resolution or higher
Winsocket installed

Slave

Windows 95/98 or Windows NT 4.0
Pentium MMX 200Mhz (Pentium II 266Mhz or better recommended)
32Mbytes of RAM or better
800x600 resolution or higher
Winsocket & DirectX 3 or higher installed

Installing **MusyX** on a Dual System Setup

Install all **MusyX** components from the CD via the SETUP.EXE program.

Consult the README.TXT file in the target directory of the installation for the newest updates and changes.

Installing the Slave Program:

Install the **MusyX** "Musician's Tools" from the CD via the Setup program.
The other items are not required to be installed on the slave PC.

Configuring the Slave:

Start the slave configuration program, "MusyX Game Boy Slave Configuration", which can be found in the program menu under the MusyX Game Boy folder. This program will let you configure all aspects of the slave's behavior.

The default settings assume a single system setup. To change to dual system setup simply tag the "dual system" checkbox. Normally it will not be necessary to deactivate the "automatic IP configuration". If you wish to do so, you will have to specify the local IP of the slave manually.

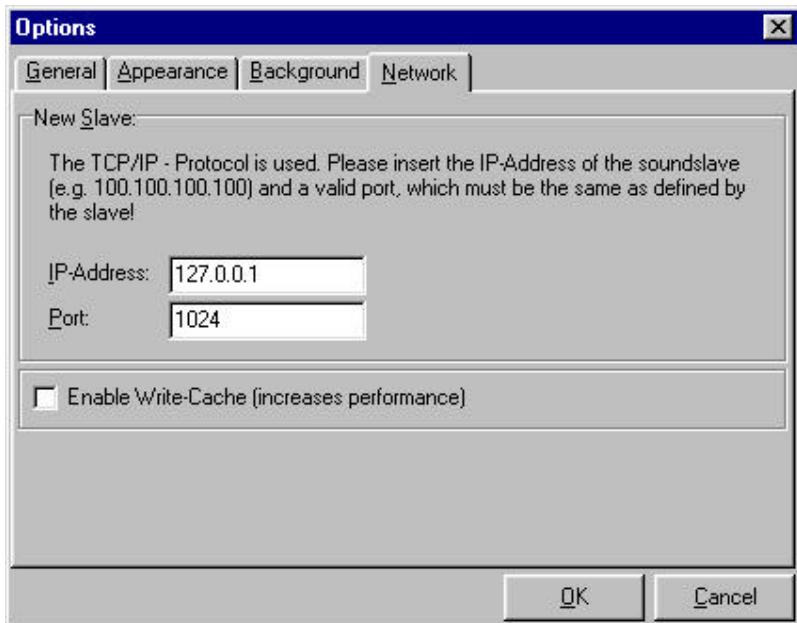
Select the MIDI port you want to use to input MIDI data to the slave (e.g. a port provided by a MIDI loopback device).

Configure the parallel port that **MusyX** will use to communicate with Game Boy, by entering the parallel port address in the Game Boy Data Link section.

Exit the configuration tool by clicking on "Ok".

Configuring the **MusyX** Editor

Start the **MusyX** editor program and select "environment" from the "options" pull down menu. In the dialog box which appears, select the "network" panel.



The default configuration assumes a single system setup. Replace the 127.0.0.1 with the IP address of the system running the slave program. Make sure that the port is set to the same port as in the slave configuration program. (By default, they are both 1024.)

Exit the dialog by clicking on "Ok".

You now have finished the dual system setup.

MIDI Loopback Devices

Since it is not possible to loopback MIDI data within the system by default when using Windows, MIDI programmers and users devised a way to make this possible. So-called "MIDI loopback devices" are available on the internet (e.g. "Hubi's MIDI loopback device" [Freeware]). Your **MusyX** CD contains a version of this MIDI loopback device, for N64 in the directory called "MusyX\Gifts", and for Game Boy in a directory called "MusyX\MIDI loopback device".

These devices interact with the system like hardware MIDI ports and are capable of routing MIDI data, eliminating the need for a real hardware MIDI port, when data is looped back internally.

In the context of **MusyX** this will let you run the slave program, the editor, and your sequencer program on one system with only one sound card, where normally you would need two soundcards to setup a hardware-based MIDI loopback.

Follow the directions in the file, MIDLPBK.TXT, under HLD_25.ZIP, to install the MIDI loopback device.

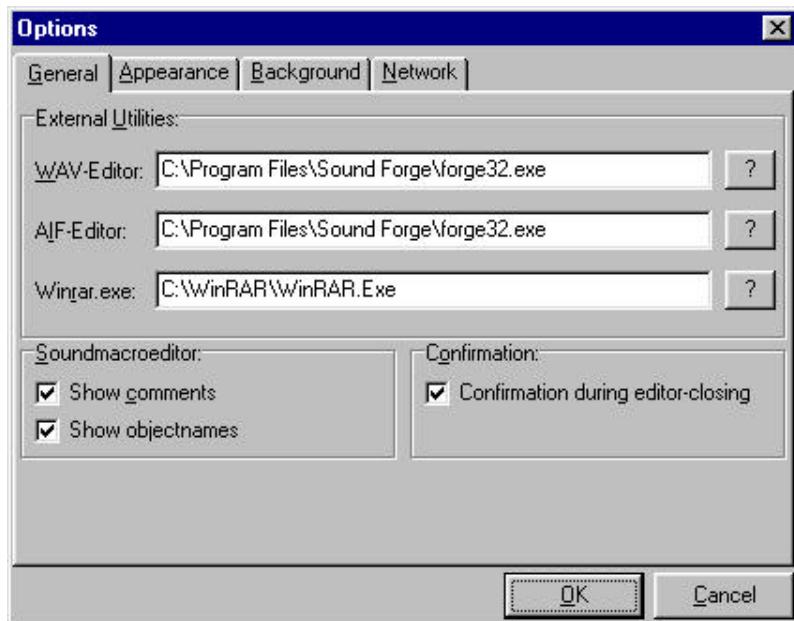
To run the MIDI loopback device, run the HWMDCABLE.EXE. You can also make a shortcut to this program, to make it easier to start it.

The MIDI loopback program will add a program to the Windows task bar. To configure it for use with **MusyX**, right click the program on the task bar. Then select LB1 for the input device and LB1 for the output device.

Note: Make sure that your sequencer program uses LB1 for its MIDI output device. Also make sure that LB1 is configured in the Windows Multimedia Properties panel.

Changing General Settings

Start the **MusyX** editor program and select "environment" from the "options" pull down menu. Select the "general" panel in the dialog which appears.



This menu allows you to define external utilities and to choose your favorite WAV and AIF Editor. "Soundforge" is one possibility for a sample editor for use with **MusyX**, though any WAV and AIF editor can be used. (Soundforge is a professional Sample editing software and not included in the package). The third selection under External Utilities lets you add WinRAR as an archive. "WinRAR" provides an easy way to make backups of your projects; **MusyX** supports the WinRAR Archiver directly from within the program (WinRAR is Shareware and not included in the package).

The 'General' screen contains two switches for the SoundMacro editor:

- | | |
|--|--|
| "Show comments"
"Show object names" | This enables/disables the comments under the value fields.
This enables/disables the names shown with the Object Ids. |
|--|--|

You will also find a switch for enabling/disabling the confirmation requester after closing internal editors.

To customize the appearance of the Editor environment you can choose your own icons and wallpaper. To do so, use the "Appearance" and "Background" tabs in the Options screen.

The "Network" tab allows you to set the IP address of the slave and the port to be used.

Software Start-up Routine

Now that your system is properly configured, perform system start-up in accordance with the appropriate paragraphs below.

N64 Start-up Routine

If this is the first time that you have run MusyX, you will find it helpful to refer to the "Walk Through" on page 69.

Start "MusyX Slave" and "MusyX Editor" from your PC.

Then, start your external sequencer program.

Game Boy Start-up Routine

Burn a copy of "Slave ROM" onto a flash ROM. This can be found on the CD under "MusyX_GB\Slaverom\gbc_slave.com".

Plug the flash ROM into your Game Boy and turn on the Game Boy's power switch.

From your PC, start "MusyX Game Boy Slave". Please refer to the Musician's Reference, Appendix 2.

Start "MusyX Editor".

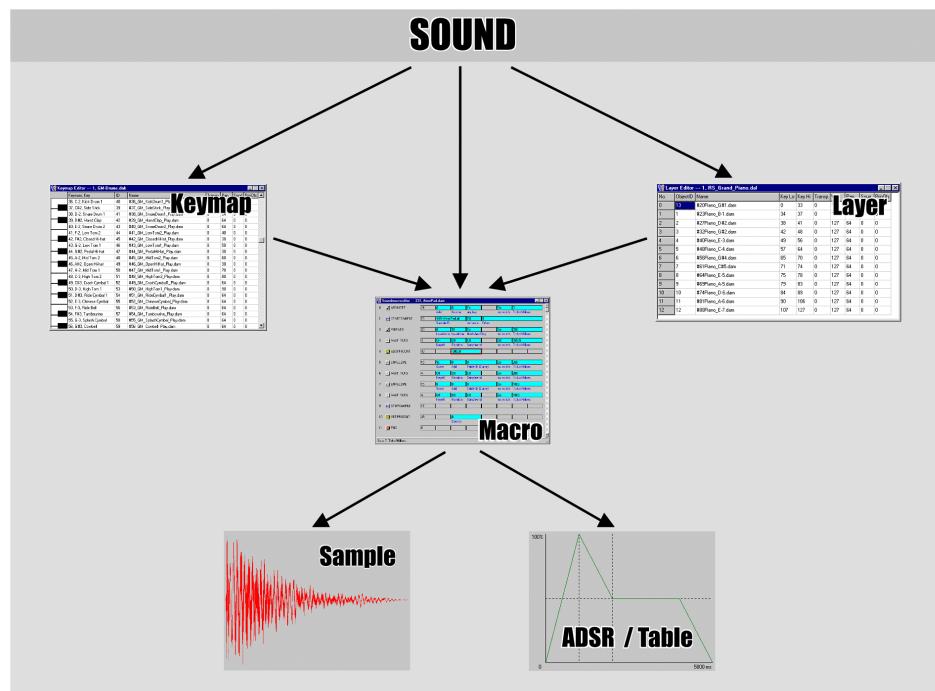
Then, start your external sequencer program.

Overview

How *MusyX* Works

The purpose of *MusyX* is to produce sound. Both musical instruments and sound effects are based upon what is simply called a **sound**.

The composition of a sound within *MusyX*:



A sound in itself is made up of a group of underlying data structures. A sound can be constructed from these data structures in different ways using three different tools.

Every sound must contain at least one **SoundMacro**. A SoundMacro is nothing more than a very simple and easy-to-understand program that defines what is being done to produce a sound over time. The programming language used in a SoundMacro is called **SMaL** (Sound Macro Language).

Another way of looking at the mechanism of a SoundMacro would be to consider a sound as being made up of a number of events that take place on a time line. Encoded in SMaL, the actions needed to define this sound are listed in the SoundMacro.

Every sound contains at least one SoundMacro, though it could very well contain more than one. Some sounds only need one sample at a time, but every MIDI key has to be occupied by a different sample.

More complex sounds may demand the use of more than one sample at any one given time. To organize multiple SoundMacros running within one sound, **MusyX** offers two data structures: Keymaps and Layers.

SoundMacros and the SoundMacro Editor

General

SoundMacros are simple, straightforward ‘programs’ that are used to define sounds. A very simplistic SoundMacro, on a sample-based system like the Nintendo 64, might say something like this (not formulated using the SMaL programming language, but in simple English):

- 1.) Use a specified ADSR
- 2.) Start a specified sample
- 3.) End

SoundMacros do far more than simply start samples or oscillators. SoundMacros have the ability to reference data, and thereby, to bind that data to the sound. This example references data defining an ADSR and a sample.

In addition, a SoundMacro does not necessarily need to start sound reproduction immediately. As mentioned above, starting a sound with a SoundMacro could simply reserve a voice.

The SoundMacro may do a number of things before any sample or oscillator is started, e.g. create a set waiting period among other things.

Creating a SoundMacro

To be able to use the full potential of SoundMacros, it is helpful to understand the basics of how the Sound routine really handles a “sound”.

If a new note is received via MIDI, a logic routine searches for a free voice and starts a SoundMacro (some lower performance systems, such as Game Boy, may use a fixed voice allocation scheme instead).

A SoundMacro is a small program made up of special commands comparable to a very simple form of the BASIC language. SoundMacros can be quickly and easily programmed from scratch, but you may also use and enhance existing SoundMacros either by template or by library functions.

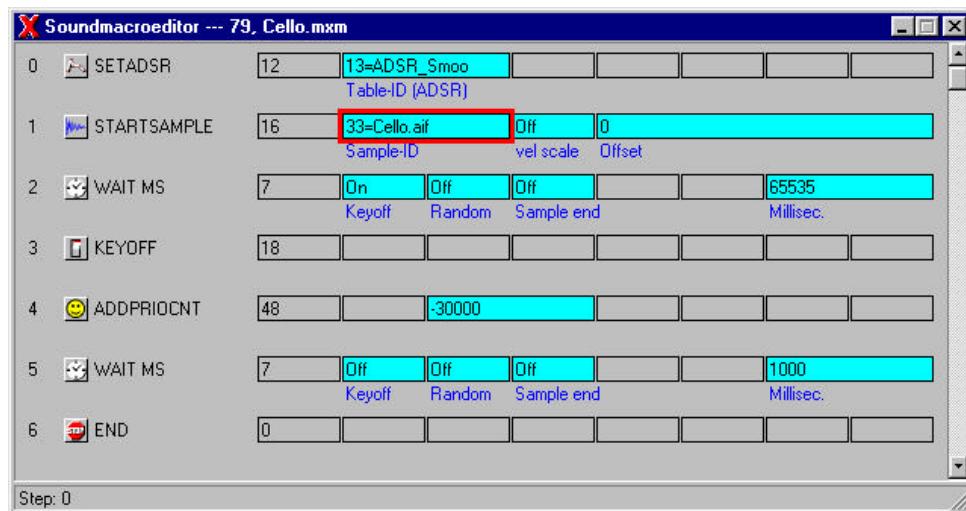
To create a new SoundMacro, select the folder “SoundMacros” of the Object-pool. Then double-click inside the view area of the project window or use the “new” entry from the pop-up menu [click right mousebutton]. Be sure not to double-click on the name of an existing SoundMacro, because this would open the SMaL-editor.

It is also possible to transfer a SoundMacro from one project to another. Both the source and destination Projects must be opened. Then drag the desired SoundMacro from the source Project to the target Project and drop it into the “SoundMacros” folder. All dependent Objects will then be copied to the new Project automatically.

Creating SoundMacros by Using Templates

The easiest way to create SoundMacros is by using one of the templates from the pop-up menu inside the sample-pool. Simply select a Sample, click [right mousebutton] and select “Templates - Generate SoundMacro” from the popup menu.

Editing SoundMacros



SoundMacros are built by combining MacroStep commands into a SMaL program. All SoundMacro commands are available in and defined by a plain-text Macro definition file. The MacroStep commands are different for each different target platform, and there is a separate Macro Definition File for each platform.

The Command Pool

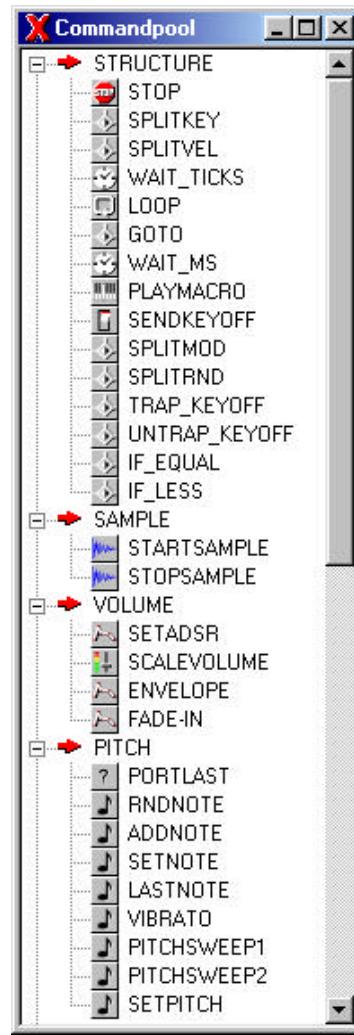
To spare the programmer the effort of typing in each MacroStep Command, MusyX comes with a graphical interface for the creation of SoundMacros.

The Command Pool Window contains all MacroStep commands that are available for a particular platform. To easily create a SoundMacro, drag these commands from the Command Pool and drop them into the SoundMacro Editor.

A new command dropped on top of another will be inserted before the existing command. To delete a command, select it with either the cursor keys or the mouse and press "Delete" on your computer keyboard. A dialog box will ask you to confirm the deletion (this can be turned off in the Options screen).

All SMaL programs must finish with the "END" command. This command cannot be moved to a different place in the program or deleted.

A complete listing of MacroStep commands and their features can be found in the Musicians Reference, along with a selection of "sample" SoundMacros.



Editing Values

Most MacroStep commands contain parameters. These parameters can be divided into three different groups: Numerical, Switch, and Reference. All parameters can be selected for editing with either the cursor keys or the mouse.

To adjust numerical parameters, select a parameter box and type over the value. The value is updated the moment the parameter is de-selected. De-selecting a parameter by using the “Enter” key on the computer keyboard allows value updates while leaving the cursor on the parameter, making it easier to quickly re-edit the parameter.

Numerical parameters can also be entered as a hexadecimal value if a “\$” is typed before the number. After pressing “enter” the value will be saved and displayed as a decimal number again.

Switch parameters contains only two settings: On and Off. To toggle these either double-click or use the return key.

Reference parameters are similar to numerical parameters except that the numbers here represent ID's instead of values. Each ID references another piece of data within the system, like a sample or a table. The visual difference between IDs and values is that the name of the referenced object is shown behind the number.

Parameters can also be increased or decreased using the +/- keys (Switch values will change between on and off).

Loops and Jumps in SMaL

With some commands you can build loops and jumps, either conditional or unconditional. For more details refer to the Macro command-reference.

While jumps to illegal locations will be detected by the slave program and cause the macro program to be stopped, the runtime libraries do NOT perform such tests. So you have to be careful when using jumps.

Calling other SoundMacros in SMaL

It is possible to call other SoundMacros from a running SoundMacro. This can be helpful to define subroutine SMaL-programs or to further structure your work.

For more details refer to the Macro command-reference.

Keymaps and Layers

General

MusyX provides two structures to allow the building of even more complex sounds, keymaps and layers.

Keymaps and layers allow you to start or reference SoundMacros, as well as other keymaps and layers, depending on which key is being depressed/received via MIDI.

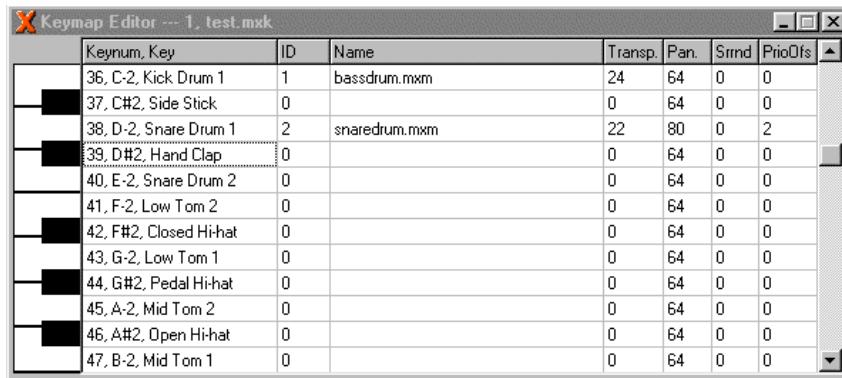


Because of the performance of the Game Boy / Game Boy Color, keymaps and layers are NOT supported on this system.

Keymaps

A keymap is basically a large table containing an entry for each of the 128 MIDI keys that exist in the MIDI standard. When a specific key is used, **MusyX** looks at the entry belonging to the current key and starts the specified SoundMacro or Layer. In addition, other parameters like panning offsets, volume and so on can be specified for each entry.

A keymap can be used to build a simple multi-sample instrument or to design a drum set.



The screenshot shows a software window titled "Keymap Editor --- 1, test.mzx". The window contains a table with the following data:

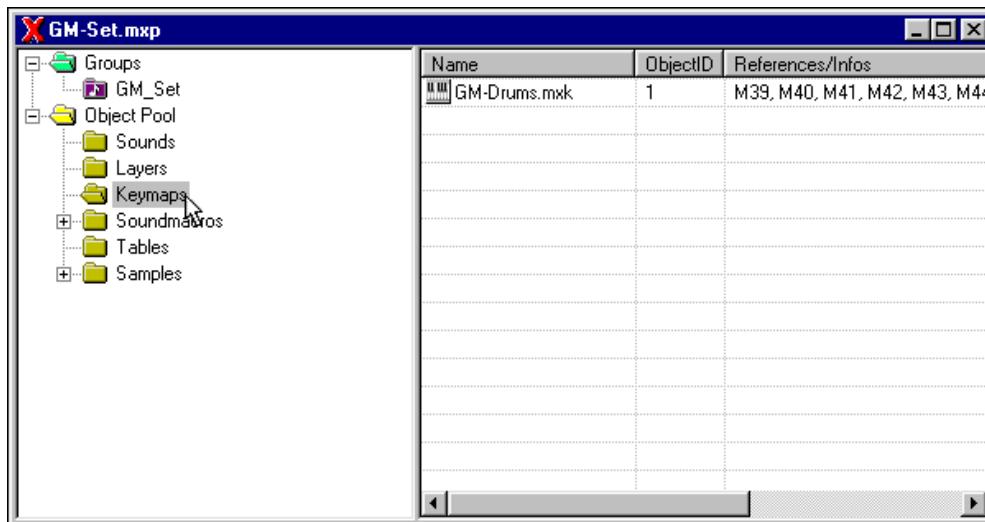
Keypad, Key	ID	Name	Transp.	Pan.	Srrnd	PrioOfs
36, C-2, Kick Drum 1	1	bassdrum.mxm	24	64	0	0
37, C#2, Side Stick	0		0	64	0	0
38, D-2, Snare Drum 1	2	snaredrum.mxm	22	80	0	2
39, D#2, Hand Clap	0		0	64	0	0
40, E-2, Snare Drum 2	0		0	64	0	0
41, F-2, Low Tom 2	0		0	64	0	0
42, F#2, Closed Hi-hat	0		0	64	0	0
43, G-2, Low Tom 1	0		0	64	0	0
44, G#2, Pedal Hi-hat	0		0	64	0	0
45, A-2, Mid Tom 2	0		0	64	0	0
46, A#2, Open Hi-hat	0		0	64	0	0
47, B-2, Mid Tom 1	0		0	64	0	0

Parameters

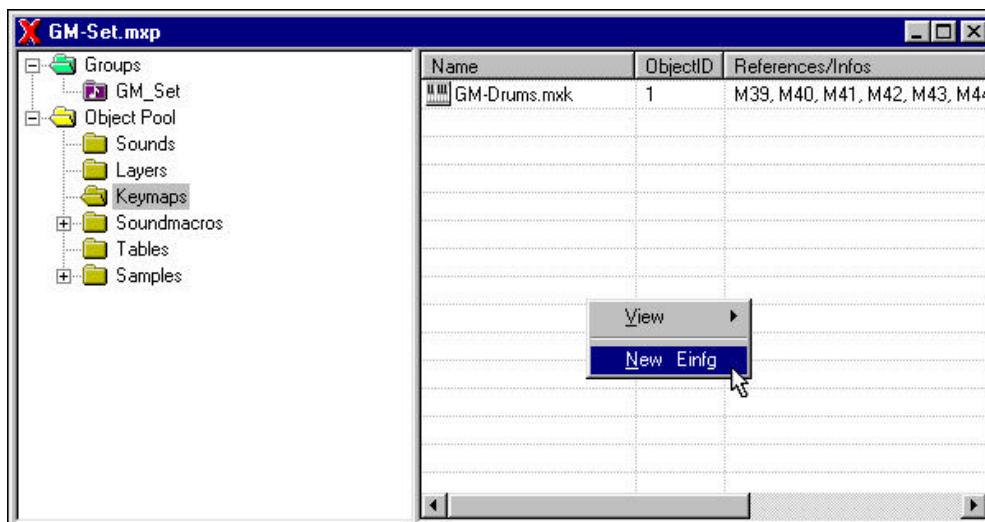
Keypad, Key	Rather than a real parameter, this information describes which key number this entry belongs to. <i>The GM-instrument usually mapped here is displayed as a reference.</i>
ID	The ID of the object assigned to this key. It can be a macro, keymap or layer.
Name	The name of the object assigned to this key.
Transp	Transpose the sound from a single key up or down in one-note steps.
Pan	Panning-Offset - adjusts the pan position of the sound relative to the one of the keymap.
Srrnd	If this is not zero, the sound on this key will be played through the surround-channel.
PrioOfs	Priority Offset - adjusts the priorities of the sounds from the keymap. This value is added to the priority which is used to "start" this keymap.

Create a New Keymap

To create a new keymap, select the "keymap" folder from the Object-pool.



Then go to the right side of the project window and double-click in an area without text, or press the right mouse button, and select "new" from the resulting pop-up menu. Don't double-click on an existing keymap (in the list), because you will edit that keymap, instead.



A keymap may be transferred from one project to another by dragging it from one project into the keymaps folder of the current project. All dependent objects will be copied automatically. The keymap is ready to use in the new project.

Layers

This structure is even more flexible. The length of a layer is flexible and, in contrast to a keymap, an entry here is not necessarily linked to a single key but rather to a key range. These ranges may even overlap between different layers.

When a key is being played, **MusyX** checks all entries in a layer and simultaneously starts all SoundMacros in the entries whose ranges contain the current key in their keyrange.

Since it is possible to let the Macros used for specific key ranges overlap, layers allow the musician to build a more elaborate multi-sample instrument.

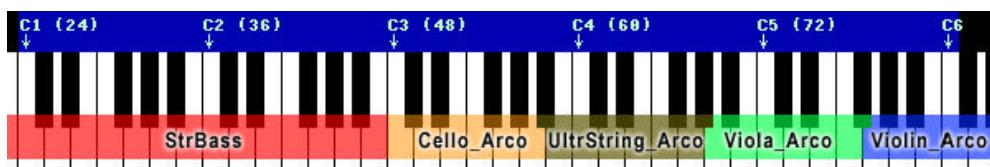
Layers can cause multiple Objects to be started simultaneously. It can also be useful for non-drum mappings of Objects to zones on the keyboard. You can reference SoundMacros or Keymaps here.

Layers Can be Used to:

- Make stacks of 2 or more sounds
- Create stereo sounds
- Map sounds to zones of the keyboard

Layer Editor --- 14, StringEnsemble2.mxl								
No.	ObjectID	Name	Key Lo	Key Hi	Transp.	Vol.	Pan.	Sourr.
0	23	StrBass.mxmx	0	47	0	127	64	0
1	79	Cello_Arco.mxmx	48	57	0	118	83	0
2	74	UltraString_Arco.mxmx	58	68	0	90	64	0
3	76	Viola_Arco.mxmx	69	78	0	127	60	0
4	77	Violin_Arco.mxmx	79	127	0	127	30	0

The example layer above, “StringEnsemble2.mxl”, would be presented on the keyrange like this:

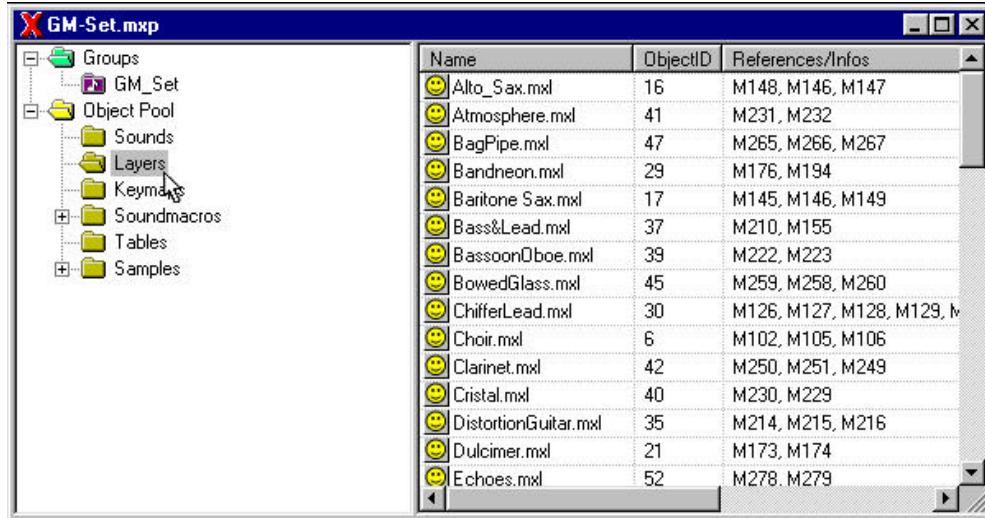


Parameters

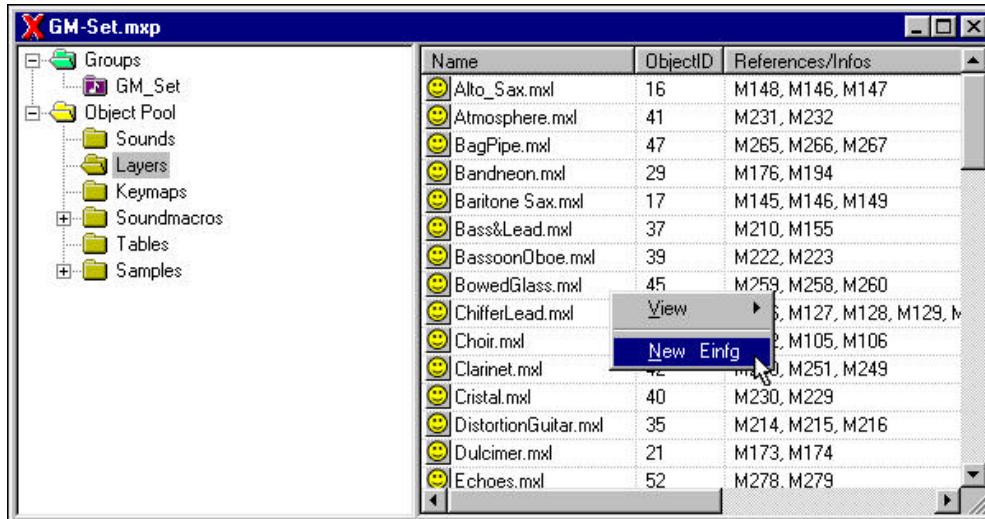
ObjectID	The objectID from the object on this key. It can be a Macro, keymap or layer.
Name	The name of the object from this key.
KeyLo	Specifies the lowest key of this range.
KeyHi	Specifies the highest key of this range.
Transp	Transposes the sound from this range in one-note steps up or down.
Vol	Volume setting defines volumes of instruments and sounds relative to the volume of the Layer.
Pan	Panning-Offset adjusts the panorama position of a sound relative to the one of the layer.
Srrnd	If this is not 0, the sound on this key will be played through the surround-channel.
PrioOfs	Priority Offset adjusts the priorities of the sounds from the layer among each other. This value is added to the priority which is used to "start" this Layer.

Create a New Layer

To create a new Layer, first select the “Layers” folder of the Object-pool.



Then go to the right side of the project window and double-click in an area without text, or press the right mouse button, and select "new" from the resulting pop-up menu.



Don't double-click on an existing layer (in the list), because you will edit that layer, instead.

Importing a Layer from another project is also possible. Both the source and the destination Project must be opened. Then simply drag the desired Layer from the source Project to the target Project and drop it on the “Layers” folder. All dependent Objects will be copied to the new Project too. The layer is ready to use in the new Project.

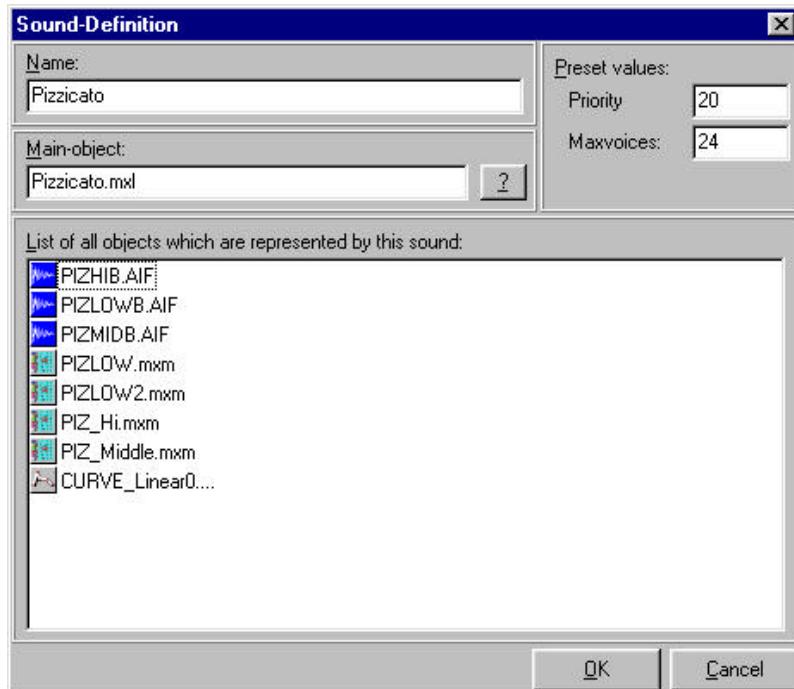
The Sound Editor

What is a Sound?

In **MusyX**, a **sound** is used to define an instrument.

Since an Instrument can consist of different Objects like Keymaps, Layers or SoundMacros, the Sound definition can be used to define the parental Object.

Once defined, Sounds can also be exported to a single file, including all dependent objects. Exported Sounds can easily be imported into other Projects.



Defining Sounds and Import/Export

To create a new Sound, first select the "Sounds" folder of the Object-pool. Then go to the right side of the project window and double-click in an area without text, or press the right mouse button, and select "new" from the resulting pop-up menu. Don't double-click on an existing sound (in the list), because you will edit that sound, instead.

This Object can then be easily exported to a single file including all dependent Objects (also Samples).

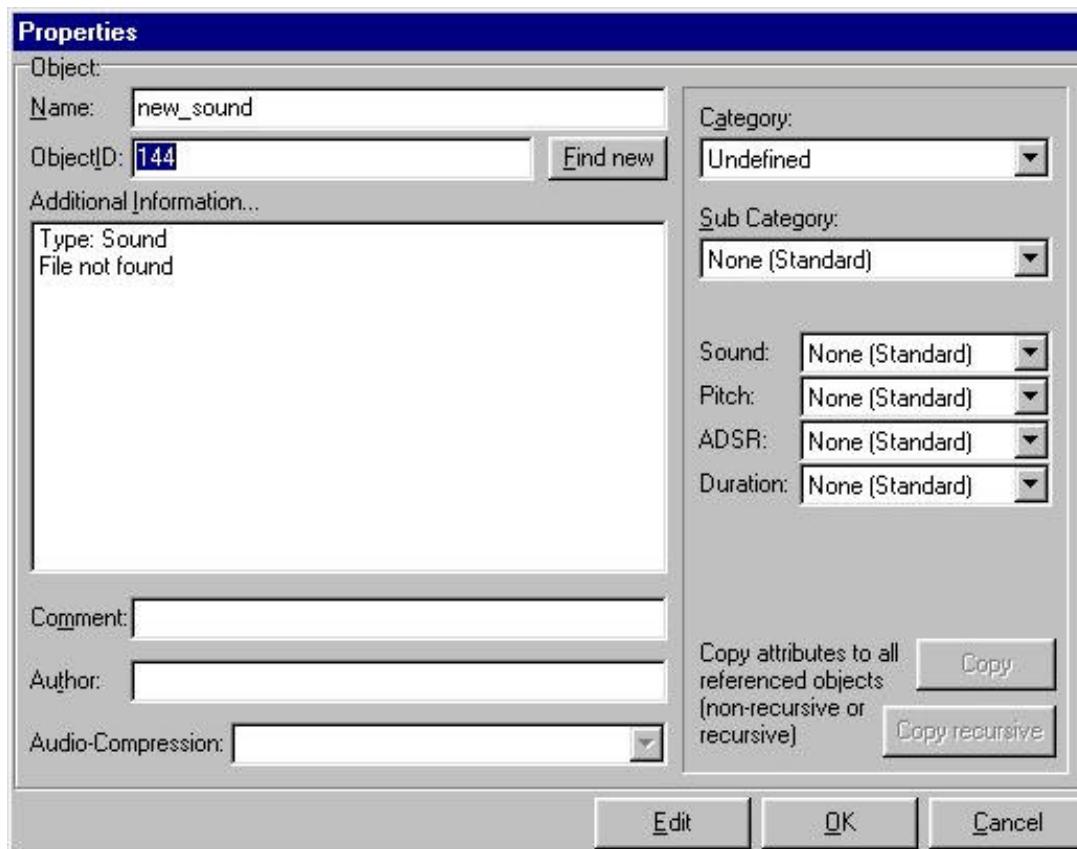
To import a Sound Export File select the “Sounds” folder and choose “Import” from the pop-up menu [right mousebutton] inside the view area.

Transferring a Sound from another project is also possible. Both the source and the destination Project must be opened. Then simply drag the desired Sound from the source Project to the target Project and drop it on the “Sounds” folder. All dependent Objects will be copied to the new Project, too.

The Sound Object Properties Window

After creating a Sound object you can view the object properties window. Here you can give the new Sound a name and edit some additional information. This information includes the author's name, comments and some attributes to describe the Sound. These definitions do not change the sound, they are only used to categorize a sound object.

These properties are not only available for Sounds, but for all objects (Layers, Keymaps, Soundmacros, Tables and Samples). In combination with the search objects window, this can be used to search for a specific sound in a large project.

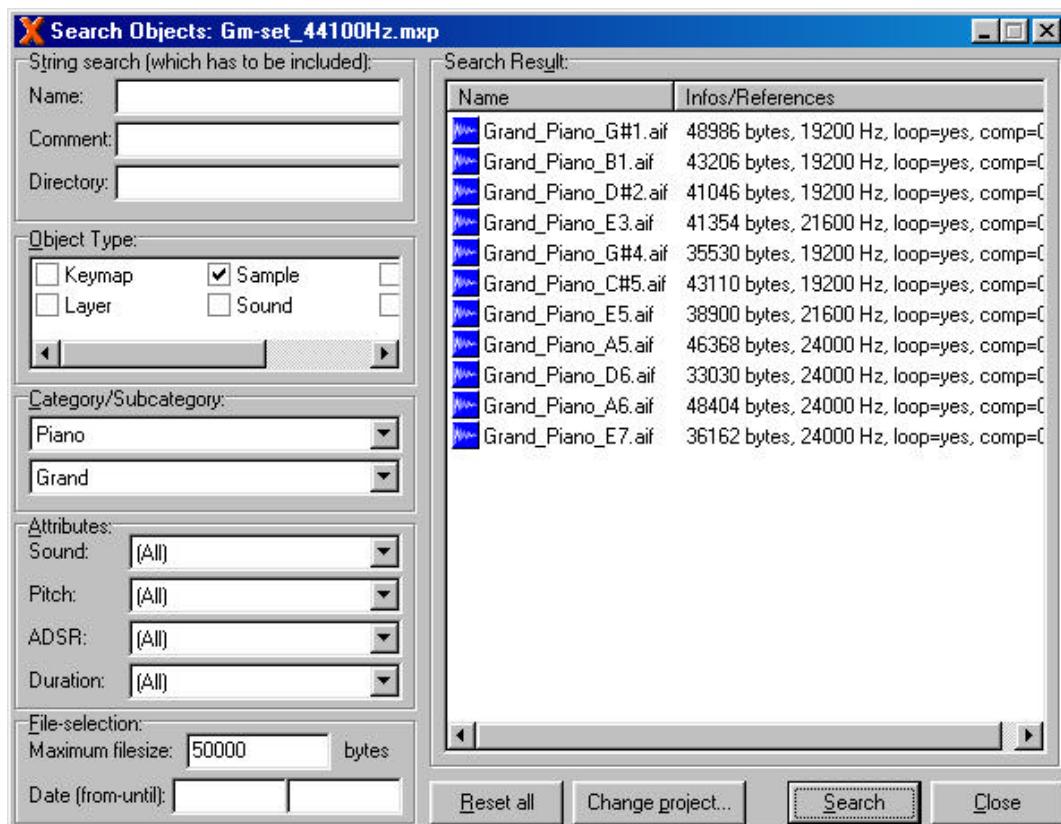


A description of the fields and buttons is included below.

Name	Change the name of the object
ObjectID	Change the ID number of an object
"Find new"	Used to find a new available ID for the object
Comment	Used for a short description of the sound
Author	Author's name can be entered here
Audio compression	Type of compression used for a sample object
Category	Choose from a list of instrument types
Sub category	Some instrument types have a sub type
Sound	Sound character
Pitch	Frequency range
Duration	Length of the sound
"Copy"	Sets the same attributes to the defined parent object of the sound (a layer, keymap or a soundmacro)
"Copy recursive"	Sets the same attributes to ALL child objects of the sound including dependent objects

The Search Window

This window is used to find objects in a large project. Here, you can search for one or more of the attributes for object properties. Additionally, there are fields for the time & date and file size (important for samples) of the objects.



Before you can start searching, you have to choose the project using the "change project" button. Only opened projects can be searched.

If you edited some or all of the fields for your desired object, simply hit the search button and all matching objects will be listed.

Samples can be auditioned by selecting them and then hitting the space bar on the keyboard. Found objects can also be dragged into a new project, if you drop them on their destination folder in the project window.

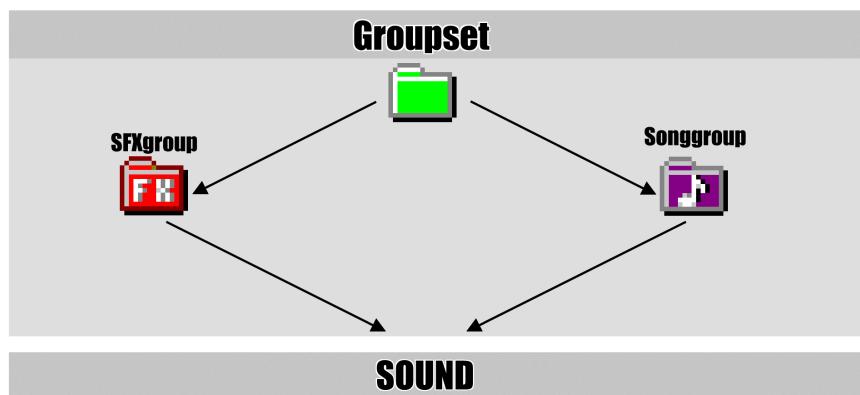
Hint: You can also create your own sound and object library, using a normal project that can be opened at the same time as your working project. Finished objects can be copied by drag & drop, to the library project for later use in other projects.

Organizing Data

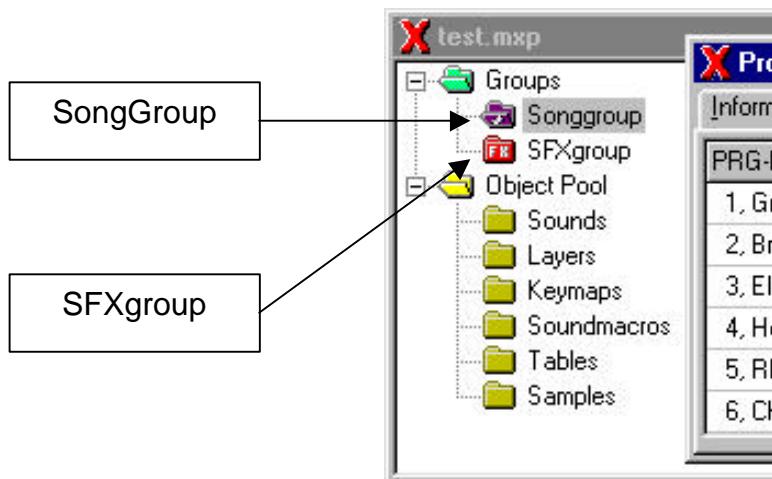
General

The basic structure **MusyX** uses to organize the data of whole songs or a set of sound effects is a **group**. For structuring the data even further, groups may be sorted into **groupsets**. These structures constitute sub-directories and are simply a means to get your project organized.

By splitting a whole project into small units, the amount of data which has to be present at a given time can be limited. Keep in mind that memory is limited. Sound usually is one of the first things where people are trying to save some space when memory space gets tight.



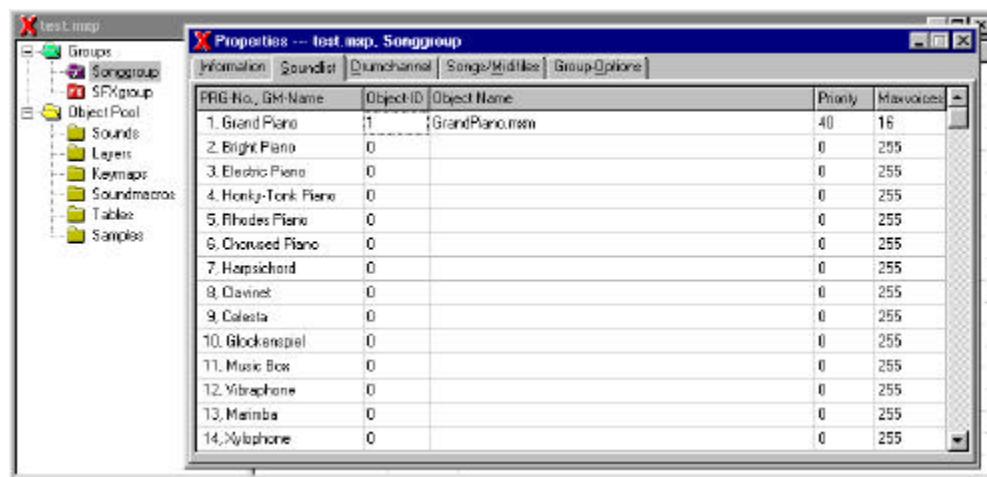
MusyX uses two types of groups: **SongGroups** and **SFXgroups**.



Note: The example project's "SongGroup" is named "Demo". The actual name is arbitrary.

SongGroups contain all data that is necessary to play back one or more songs (useful to keep a level-music and a boss-music together in the memory). They reference all song data generated with an external sequencer program, as well as all sounds that are used as instruments within the songs.

MIDI uses program numbers from 1 to 128 to identify instruments. **MusyX** maps sounds to these numbers using the Soundlist or Drumchannel from the SongGroup Properties, which contain one entry for each MIDI program number. To access the songgroup's properties, simply double-click the songgroup in the project window.



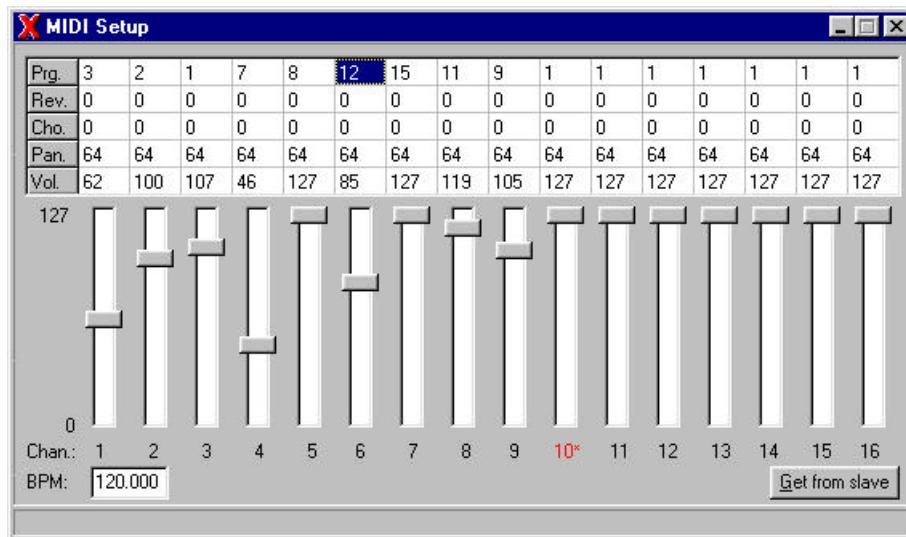
The Soundlist and the Drumchannel, together with some other data, represent a SongGroup. This SongGroup can contain more than one song, as long as those songs use the same instrument definitions.

In the tab "Songs/Midifiles", MIDI-files can be entered. This does not serve an immediate purpose during development, but is necessary for the conversion of the data. When converting the data, the Editor uses the entry to determine which file has to be included in the conversion.

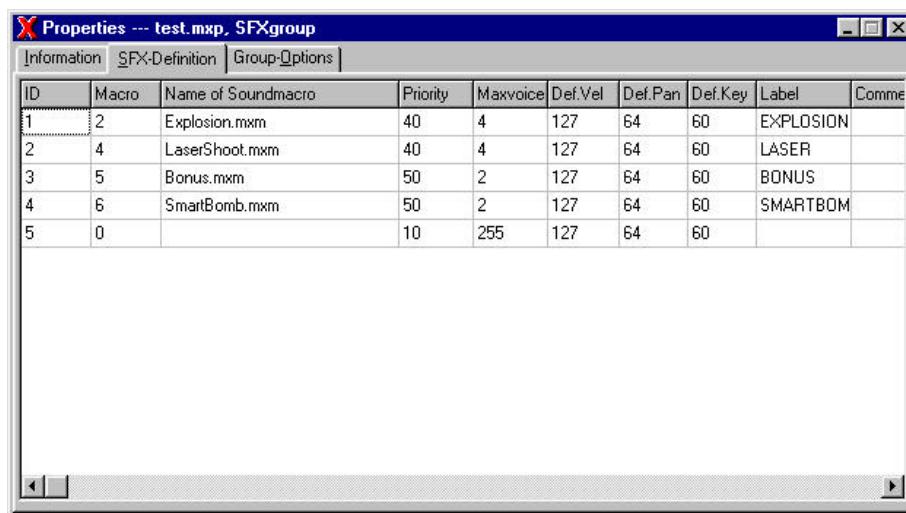
Each song referenced by a specific SongGroup has its own **MIDI-setup**. This structure consists of 16 channels that are referenced by the SongGroup. The MIDI-setup contains initial settings for some major MIDI controllers and allocates the programs (i.e. instruments) that are to be initially played to their MIDI channels.

The settings can be changed while the song is in progress, allowing for instruments to be changed in mid-song.

The Drumchannel, e.g., is for MIDI channel 10, which is defined as the drum channel in the General MIDI standard. The other 15 MIDI channels can be referenced with sounds from the soundlist the way the musician chooses.



SFXgroups basically consist of just one large table containing one entry for each sound effect to be defined. Each sound effect uses a specific sound. Since several sound effects can start with the same sound, it is imperative that sound effects define other parameters like priority, Maxvoices and Default Panning to differentiate the sound effects.



To further simplify the teamwork between sound artist and programmer, each sound effect is assigned a Label.

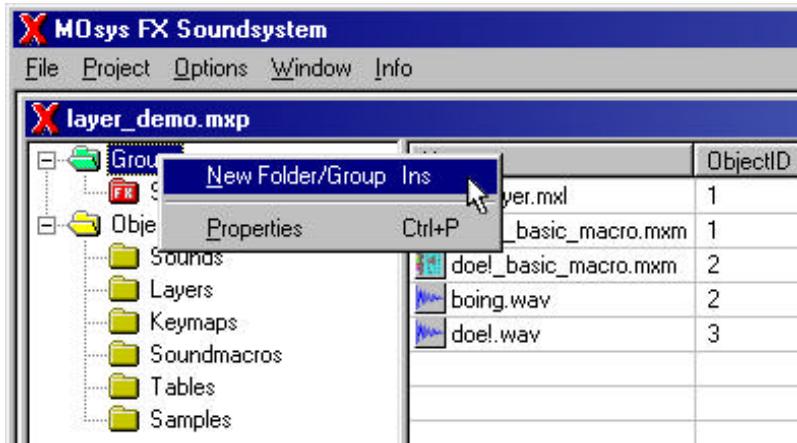
MusyX automatically generates a header file that defines constants, which the programmer can use to reference the sound effects at runtime. These constants contain the Label that is assigned to the sound effect.

As a result of this labeling, musician and programmer no longer have to pass cryptic sound effect ID numbers to each other in order to synchronize their work.

Because they gather together a predefined set of sounds to be used as sound effects, SFXgroups are the perfect tool for organizing multiple groups of sound effects, e.g. all sound effects that are to be used in a specific level of a game.

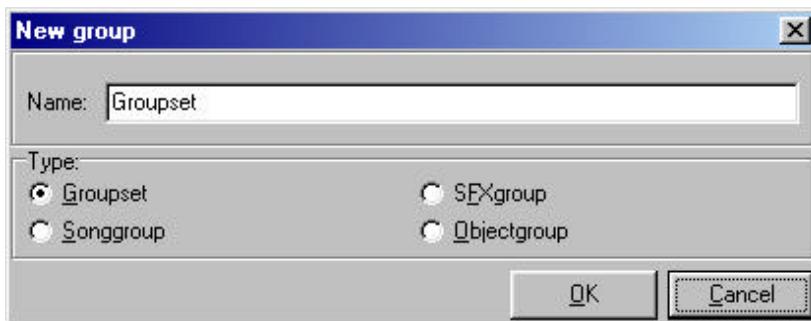
Managing Groups

To create a new Group, click [right mousebutton] on the Groupfolder and select "New" from the pop-up menu.



You have the following choices:

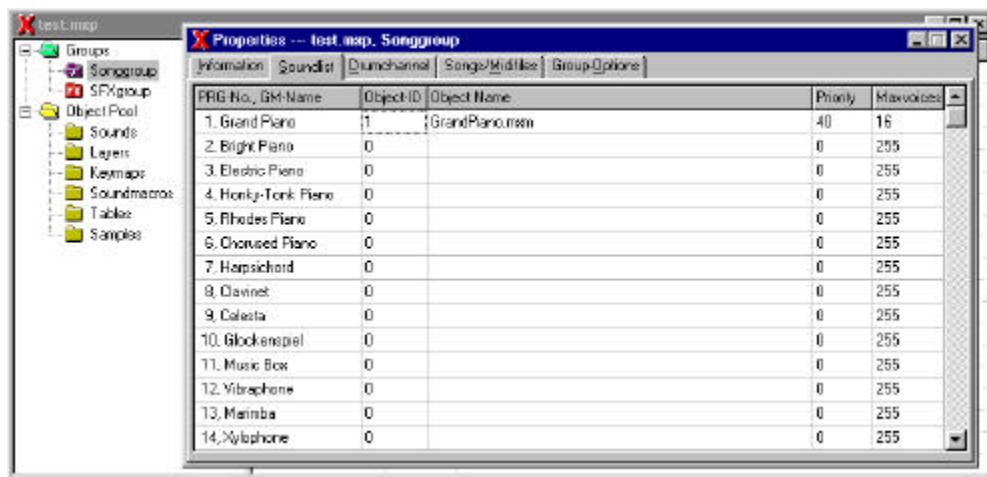
- SongGroup is used for one or several Songs
- SFX-Group for one or several Sound-FX
- Object-Group to collect a bunch of Objects
- Groupset similar to a Folder which can contain several other Groups



SongGroups and Their Parameters

A SongGroup is comparable to a set of soundprograms and multisets of a synthesizer or sampler. For every SongGroup that represents a level or stage in a game you have to sort the required "Sounds" from the Object Pool into a *Soundlist* to assign them to the correct MIDI program number.

To edit a SongGroup, double-click on it to open the Properties window. Inside the Properties window, you can edit the "Soundlist" or select one or several "MIDI files" for this SongGroup. This "Soundlist" represents MIDI-programs that reference the instruments for channel 1~9, 11~16 and the "Drumchannel" for MIDI-channel 10 in the MIDI setup.

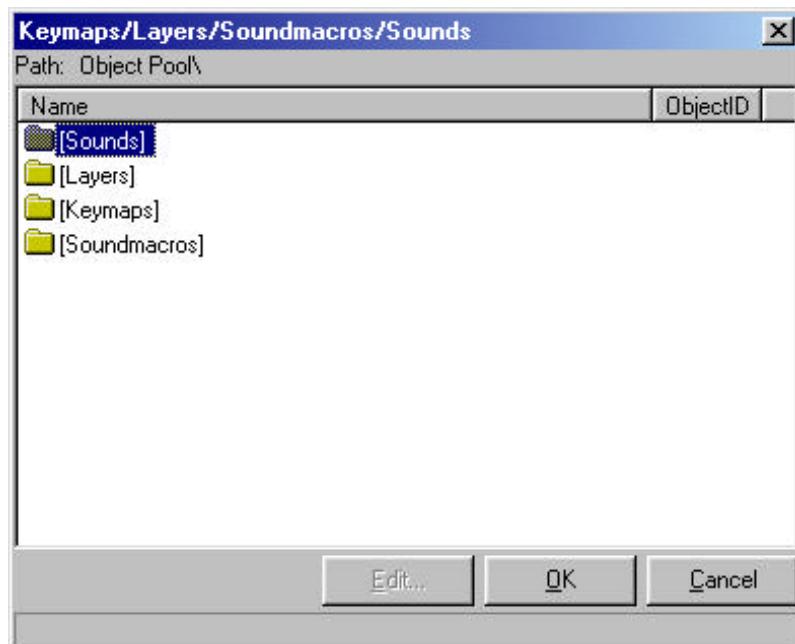


Take a closer look at the Soundlist. On the very left you see the MIDI specific program change number from 1 to 128.

The GM equivalent soundprogram is listed behind the number, but it is not obligatory to follow this "guideline". You can put a Bass sound on MIDI program #1 if you want to, but realize that it will be a Bass and not a Piano if you later select this soundnumber from your Sequencer.

The Drumchannel is very similar to the Soundlist, but for GM-compatibility reasons it represents an alternate list of Sounds (in this case Drumkits) always solely for MIDI channel #10.

To enter a Sound into the Soundlist or Drumchannel, select the desired number and double-click on it or press the “Enter“ key on your computer keyboard. A dialog box will show all objects from the pool that fit in here, such as Sounds, Layers, Keymaps or SoundMacros.



If you do not use program changes in your MIDI-file, please also read the section of this manual concerning MIDI Setups, because you need to preset the MIDI programs to the MIDI channels.

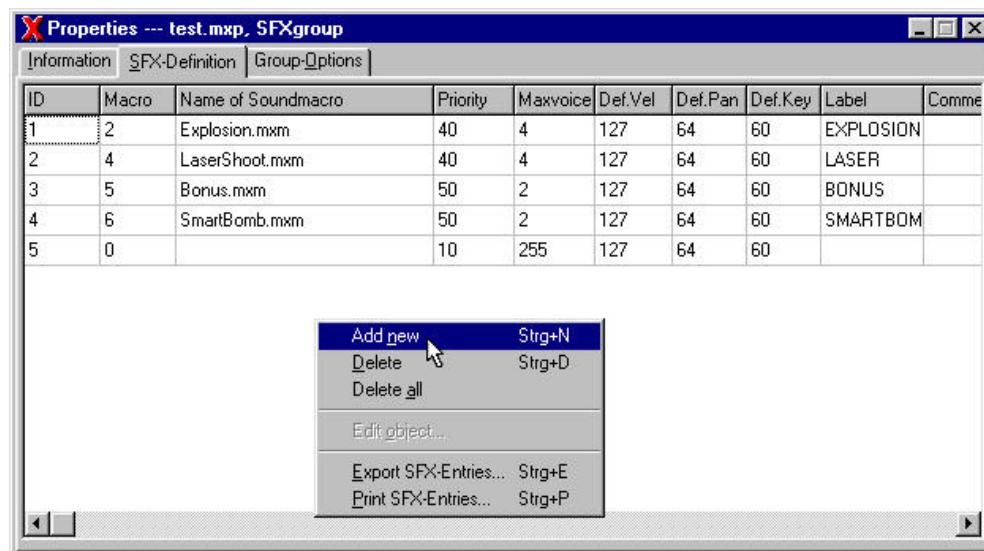
SFXgroups and Their Parameters

In the simplest case, a Soundeffect is just a Sound or a SoundMacro. Within the system, of course, SFX are handled somewhat differently than Sounds or SoundMacros.

Because the programmer of a game wants to be able to insert SFX into his program as easily as possible, the sound designer has to define one or more special tables called SFX-Groups.

The first major difference between SongGroups and SFX-Groups is the length of the Table. An SFX-Group can handle many more Sound IDs than a SongGroup can. Thus the table-length varies and in contrast to a SongGroup, a newly created SFX-Group has no entries to edit.

To add new entries, use the “Add new” function of the pop-up menu by right-clicking inside the SFX-Group properties window.



Here you can now enter, for example, the name of the Macro, or you may enter a Layer. You can also adjust parameters, if you wish to do so.

Testing Sound Effects

It is possible to test SFX with the *MusyX* editor and slave before incorporating them into the game. This is useful to check priority settings or volume levels.

First you have to send the desired SFX-group to the slave the same way you would send a SongGroup. If the slave receives a SFX-group, it automatically switches to a different playmode.

Beginning with MIDI-channel #1 every keynumber represents one SFX-ID and can be played by a connected MIDI keyboard or the virtual MIDI keyboard. A connected external keyboard has the advantage that more than one SFX can be triggered at the same time.

If there are more than 128 SFX, MIDI channel #2 represents IDs 128-255, MIDI channel #3 represents IDs from 256 to 383 and so on.

The Parameters

ID	The number that the programmer sends to the SFX-API to start a SFX using the defined label (see below)
Macro	SoundMacro number
Name	Name of the SoundMacro
Priority	A Priority number for each SFX, if voices are shared
Maxvoices	The maximum of voices allowed for this SFX
Def.Vel	Default velocity or Start-volume (the programmer can override this value)
Def.Pan	Default panning (the programmer can override this value)
Def.Key	The default key (the programmer can override this value)
Label	The programmer uses this label to reference a SFX
Comment	Comments for each SFX

Managing the Object Pool

You can create unlimited sub-folders for any kind of Object inside the Pool. To do this, select an existing folder and use the “Create folder” entry of the pop-up menu [right mousebutton]. This is useful for organizing your work.

Adding Samples

To add a sample to your project, first select the samplefolder from the Object-pool. You can either double-click inside the view area of the Sample-pool or choose “new” from the pop-up menu [right mousebutton].

You can now select a sample you want to import through the appearing file dialog box. If you select a sample using the mouse or the cursor keys, you can play it through your Soundcard from within the editor by pressing “space” on your computer keyboard. This can be done any time you select a sample in any window or dialog box.

MusyX supports both Microsoft WAV and Macintosh AIF files. WAV files, however, often do not support loops and **MusyX** then uses them only for one-shot samples. AIF files support both loops and one-shot format and are preferred for use with **MusyX**.

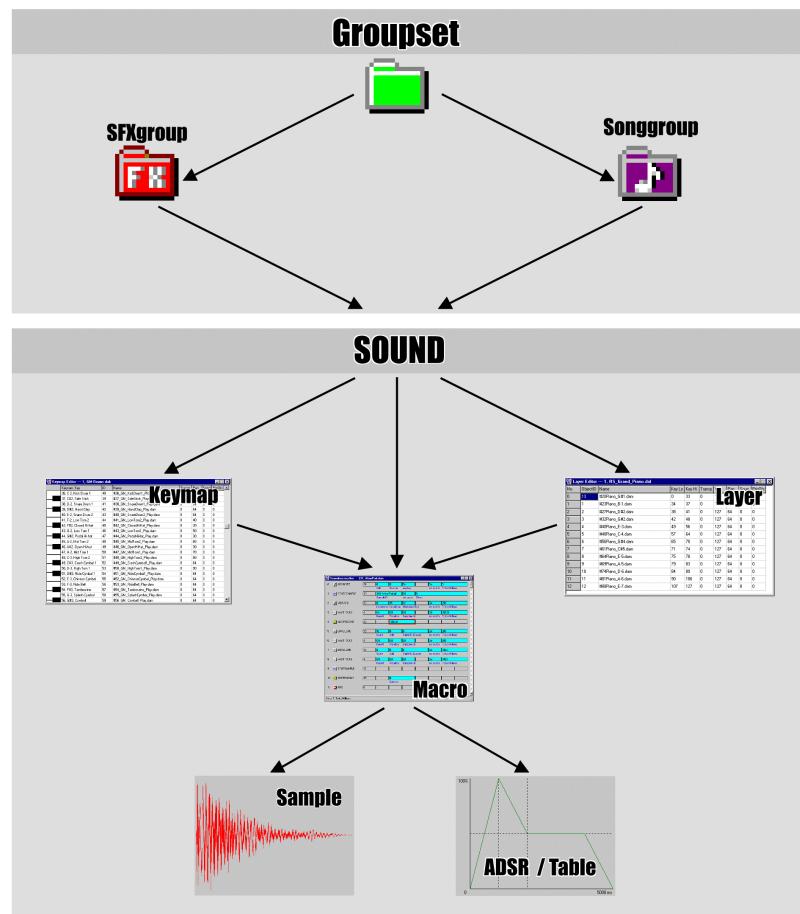
The Different Parts of a Project

All of the data of one game soundtrack is called a *Project*.

Within a project the system distinguishes between two different basic data types, *Groups* and *Objects*.

A *Group* is a collection of *Objects* belonging to the same level or stage of a game. It can be a *SongGroup*, *SFX-Group* or a *Groupset* of Groups.

Which one of these it is going to be depends on the need of the project or the game. To choose a way to organize the data for a game is at the user's discretion.



When contemplating the sounddefinition side, one will find a more complex data hierarchy. In this context, a *Sample* is the lowest part of a Sound. A *Sound* on the other side represents the highest kind of data in a sounddefinition.

Hint: *Layers* and *Keymaps* can refer to each other **or** to *SoundMacros*.

How the Structure Relates to the Actual Game Data

There is no major structural difference between the Project data used inside the Manager and the final converted Game data. The Project data is of course more comprehensive, since it contains additional information, e.g. names of Objects. Samples may also be stored in a different format from the one needed for the destination platform.

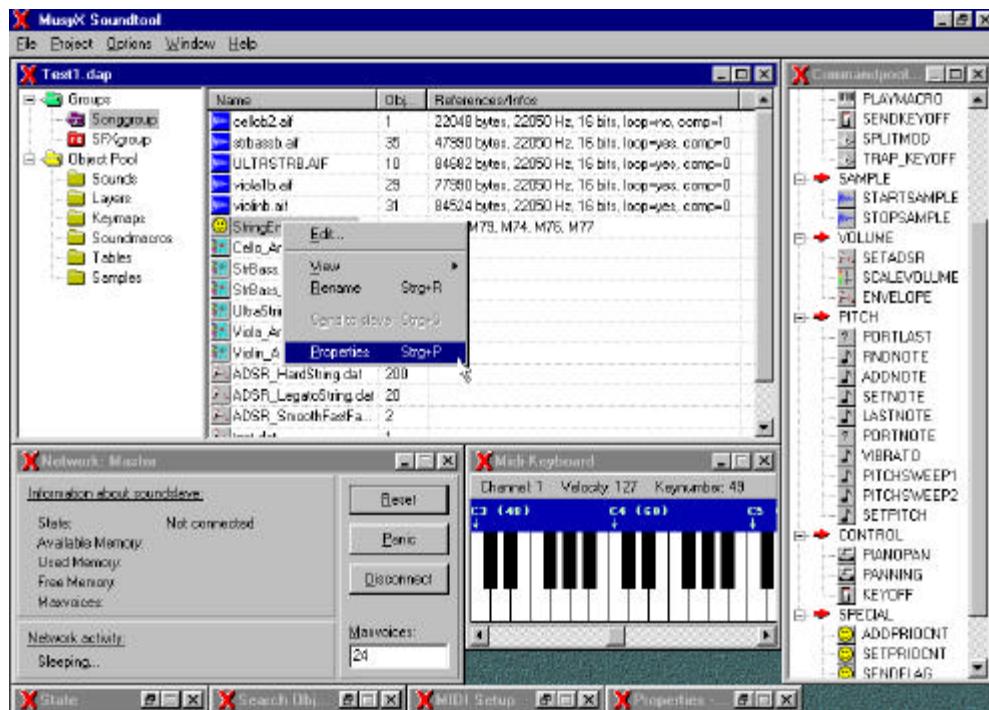
Transferring Data Between Projects

Objects can be transferred between Projects either by exporting and importing Sound-objects, or simply by dragging any desired Object from one Project window to another.

The Project Manager

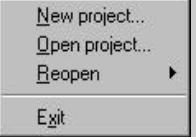
General

The Project Manager is the main working environment of **MusyX**. It is used to edit and manage all data related to the project.

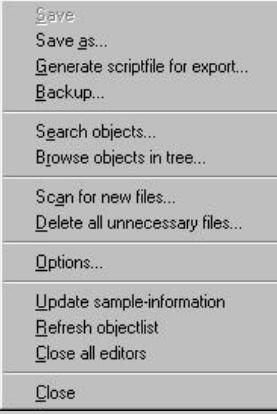


A typical screen layout of the Project Manager.

Project Manager Menus

File Menu	Description
	
New	Creates a new project
Open	Opens a saved project
ReOpen	Chooses a saved project from a list of the last 10 recent projects
Exit	Exits MusyX -Manager

After you open a project, the Project Menu becomes available.

Project Menu	Description
	
Save	Saves the active project
Save as	Saves the project under a different name
Generate scriptfile for export...	Scriptfile for the converter

Backup	When a valid path to a WinRAR archive program is entered in the Options menu, this menu item will be activated. When selected, it will use this program to "RAR" all data of the project together into one file. This may not be used to archive or transport data easily.
Search objects	Searches for objects using the library functions
Browse objects in tree...	Opens an additional project window with the complete tree of all objects and their hierarchy.
Scan for new files...	Files that are copied or saved to the project directories by external tools can easily be imported into the project with this function
Delete all unnecessary files...	Deletes all files that are not related to or depending on objects within the project.
Options...	Setup options relating to the project.
Update sample-information	Updates the cached sample information if samples were changed by external tools.
Refresh objectlist	This causes the object reference list (in the right-hand part of the project window) to be refreshed manually. Not only the display will be refreshed, but also the internal data structures. During normal use of the editor, there should be no need to use this function.
Close all editors	Closes all open editors at once
Close	Closes the project

Options Menu Environment...	Description
Environment	Options and setup of the editors environment

Window Menu	Description
Undo last command Cascade > Tile > Align Commandpool Window(s) 1 MIDI Setup 2 State 3 Network: Master 4 Properties ... (empty) 5 Search Objects (disabled) 6 Midi-Keyboard 7 Commandpool - Testproject.mxp ✓ 8 Testproject.mxp	
Undo last command	Undoes any window arrangements
Cascade	Re-arranges windows in an overlapping diagonal line. You can choose between "all windows" and "Project windows" or "Editors" only.
Tile	Re-arranges windows. You can choose between "Project windows" or "Editors".
Align commandpool	Aligns the commandpool-window with the right side of the main window
1 Midisetup	Quickselect for the Midisetup-window
2 State	Quickselect for the Status-window
3 Network: Master	Quickselect for the Network/Master-window
4 Properties	Quickselect for the Group-/Object- Properties-window
5 Search Objects	Quickselect for the Librarian search window (disabled until a project is opened)
6 Midi-Keyboard	Quickselect for the Virtual MIDI-keyboard window
7 Commandpool	Quickselect for the Commandpool window
8 Testproject.mxp	Quickselect for an open project

Info Menu	Description
About <i>MusyX</i>	Developers information & copyright notice

Walk Through

General

This chapter provides a step by step example of how to design a simple instrument and a simple sound effect and export it so that the programmer can include it into his or her game.

The steps are:

- Start the *MusyX* Editor
- Create a Project
- Import a Sample
- Create an Instrument using a SoundMacro
- Create and setup a SongGroup
- Make a short MIDI sequence
- Create a Sound-FX
- Convert the data

Start the Soundslave

Start the slave manually from Win95/98-Startmenu/Programs/ **MusyX**.

Launch the **MusyX** Editor

Start the “**MusyX** Editor” from the Win95-Startmenu or directly by double-clicking the **MusyX** -Icon from your desktop if available.

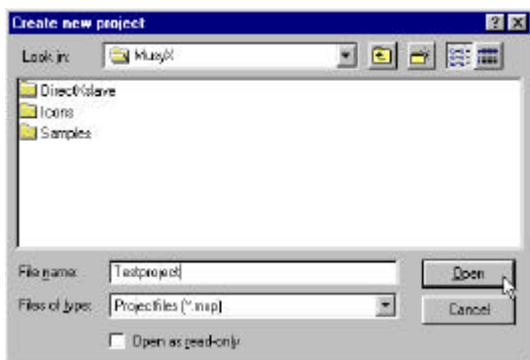


The Manager screen after startup

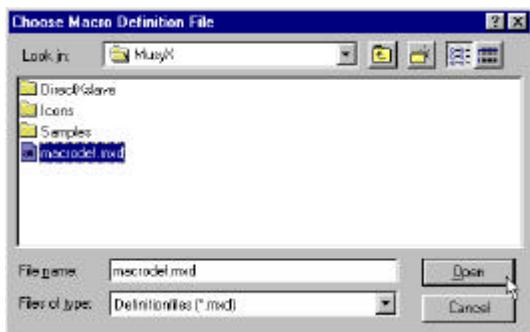
Creating a New Project

Select “New project” from the ‘File’ pull down menu.

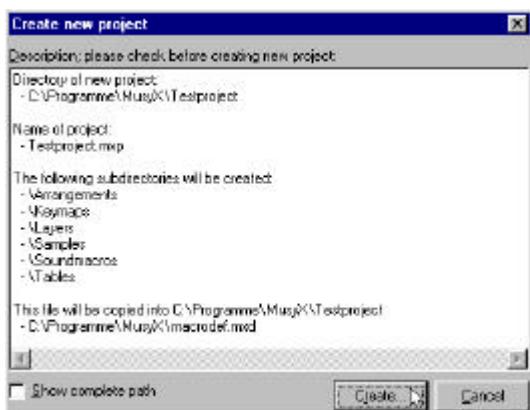
This opens a regular File dialog box, where you name the Project. For this example name it “Testproject”.



Next, choose a Macro definition file. This file can be found in your **MusyX\bin\misc** Folder (“macrodef.mxd”). This file defines the set of commands used by the SoundMacro language (SMaL). The Game Boy macro definition file is located in **MusyX-GB\macrolibrary**.



Finally a confirmation window appears and you have the option to create the project (click on the “Create”-button].



The Project Window

The newly created Project looks like this:



From this window you have access to all data related to the project. It is split into a left and right area. On the left side is a folder structure, the "Groups" and the "Object Pool". On the right side of the window is the view area where the Sound-Objects are displayed. (Since a new Project contains no Sound-Objects, this area is empty at the moment.)

The "Object Pool" contains Subfolders for sound-related Objects like Samples, SoundMacros, Layers, Keymaps, Tables and sounds, but more about this later. For now we need only Samples and SoundMacros.

The "Groups" are mainly used to organize the different sounds into SongGroups (e.g. for the different levels of a game) and SFX-Groups (lists of Sound effects).

The next step is to add a Sample and build an Instrument using a SoundMacro. *MusyX* supports both Microsoft WAV files and Macintosh AIF files.

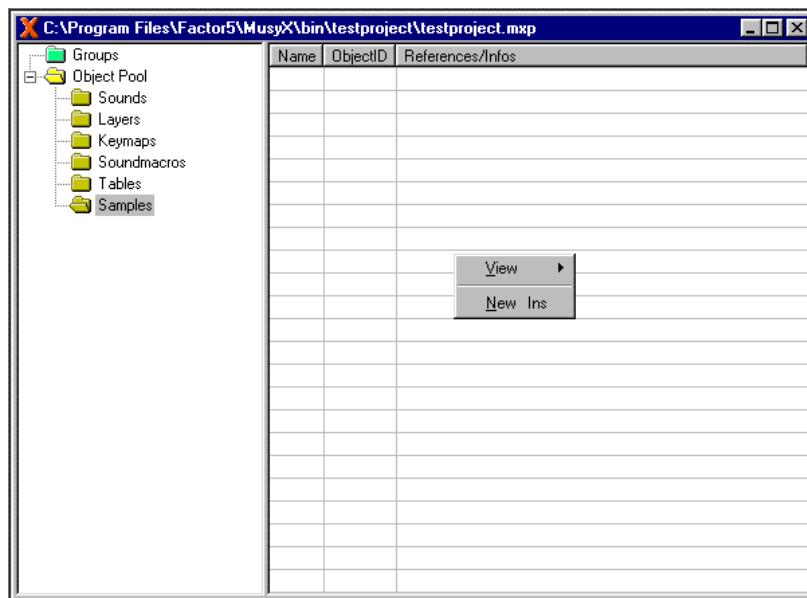
Keep in mind that WAV files often do not contain loop information.

Adding Samples

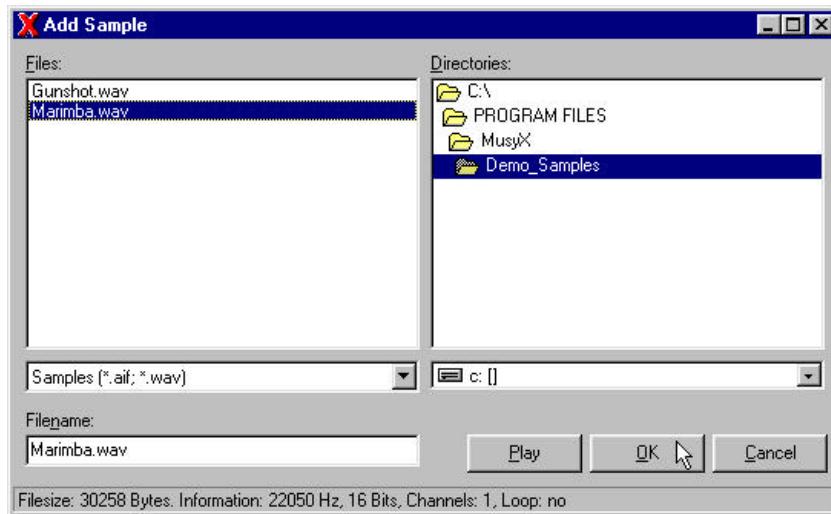
To add (import) a sample to your project, first expand the “Object Pool” folder by clicking on the “+”-symbol.



Select the “Samples” folder:



Then go to the right side of the project window and double-click in an area without text, or press the right mouse button, and select "new" from the resulting pop-up menu. Don't double-click on an existing sample (in the list), because you will edit that sample, instead.



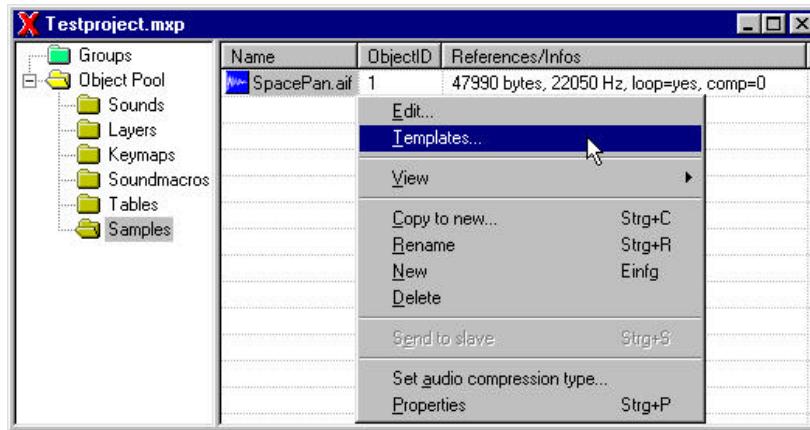
Select the Sample “Marimba.wav” from the Directory “Demo_Samples” of your **MusyX** Folder and click the OK-Button. Now the Sample has been imported into the project.

Before we can use the Sample as a sound, we have to create a SoundMacro.

A Very Simple SoundMacro (SMaL Program)

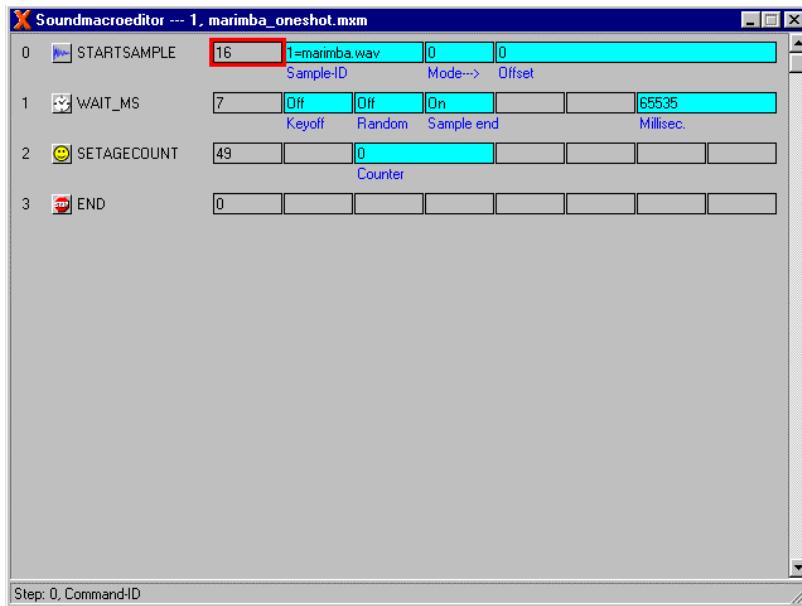
Starting with a Sample, the smallest building block in this context, the easiest way to create a Sound Macro is using a “Template”. “Templates” are nothing more than predefined SoundMacros.

Click [right mousebutton] on the “Marimba.wav” Sample and select “Templates” from the appearing pop-up menu.



This will bring up a window with some SoundMacro-templates. For our “Marimba.wav”-Sample we can use the “Oneshot” Template.

Now we have created a SoundMacro that plays our Sample. To see what this SoundMacro looks like: Select the SoundMacros Folder in the Object Pool and open the SoundMacro-editor by double-clicking on the marimba_oneshot1.mxm Object.

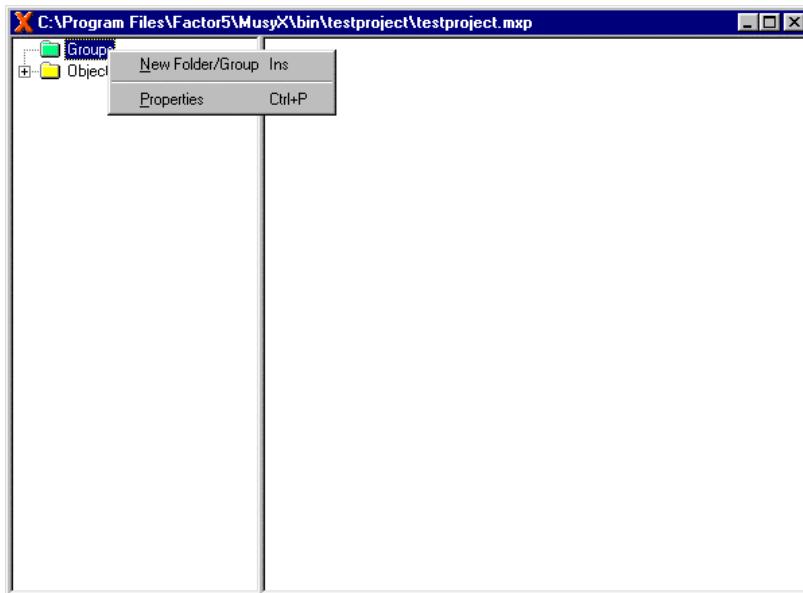


To actually play this sound we need to tell the system when to use it. Here we are going to set the sound up to be used as an instrument.

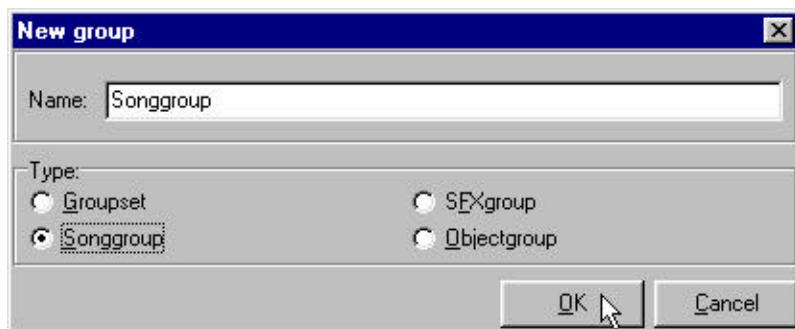
Hence the next step is to define a song group.

Defining a SongGroup

To create a SongGroup click [right mousebutton] on the “Groups” folder in the Project window and a pop-up menu appears. Select “New Folder/Group“.



From the following dialog box choose “SongGroup“ and click OK.

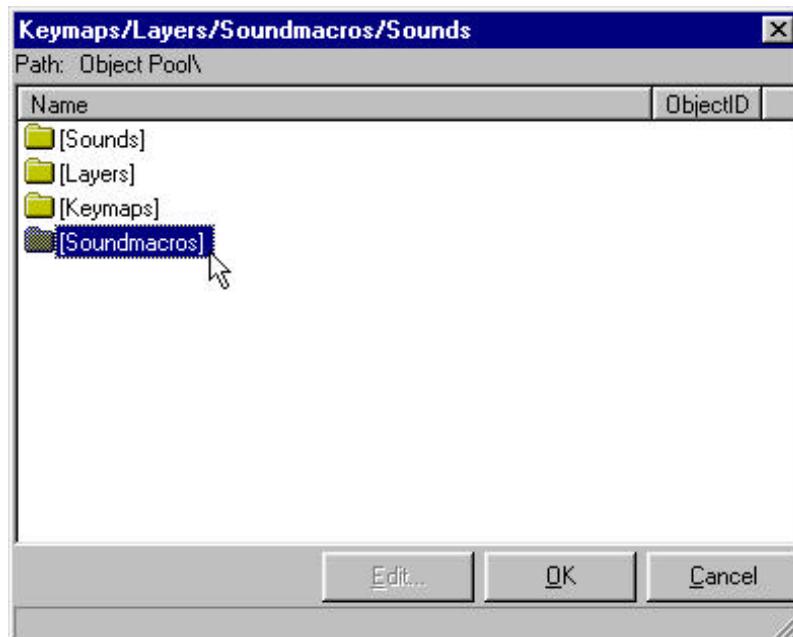


To edit the new SongGroup double-click on its name and it will open the Properties window. Inside the Properties window you have to enter our new SoundMacro into the “*Soundlist*“.

Properties --- Testproject.mxp, Songgroup				
Information	Soundlist	Drumchannel	Songs/Midifiles	Group-Options
PRG-No., GM-Name	Object-ID	Object Name	Priority	Maxvoices
1, Grand Piano	0		0	255
2, Bright Piano	0		0	255
3, Electric Piano	0		0	255
4, Honky-Tonk Piano	0		0	255
5, Rhodes Piano	0		0	255
6, Chorused Piano	0		0	255
7, Harpsichord	0		0	255
8, Clavinet	0		0	255
9, Celesta	0		0	255
10, Glockenspiel	0		0	255
11, Music Box	0		0	255
12, Vibraphone	0		0	255
13, Marimba	0		0	255
14, Xylophone	0		0	255

The Soundlist is used to map SoundMacros or other Objects, Keymaps or Layers for example, to a MIDI program-number that can be accessed by MIDI program-changes. The names beside the numbers on the left side are only shown as a reminder for General MIDI purposes. More detailed information about the Soundlist is coming up in a later chapter.

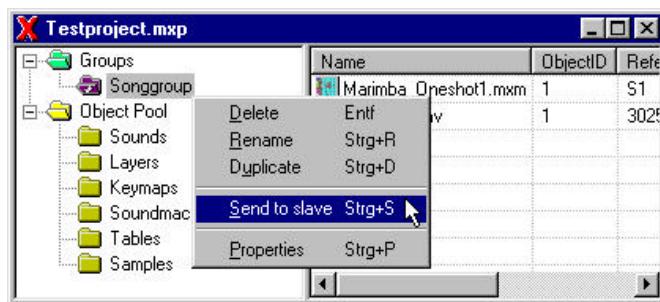
Next double-click on the blank field below "Object name", of row No 1. A dialog box is shown with different types of Objects from the "Object Pool".



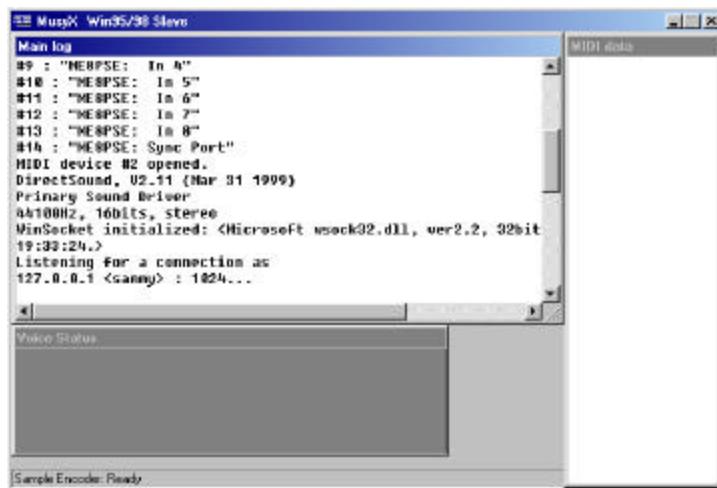
Open the "SoundMacros" Folder and choose the new "Marimba_Oneshot1"-Soundmacro. Click Ok.

Playing the Instrument for the First Time

The SongGroup is now ready to be sent to the Slave. Select the SongGroup and click the right mousebutton to open a pop-up menu. Choose "Send to slave" and all data belonging to this SongGroup will be transferred to the Soundslave.

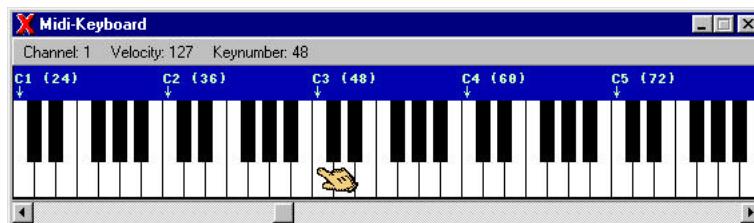


The transfer should last only a few seconds for our small testproject. On the status-window of the Soundslave PC you can watch the transfer.



After the transfer is completed you can play your first instrument using your sequencer and MIDI-keyboard. Make sure that your sequencer is set to the right MIDI-output and channel 1.

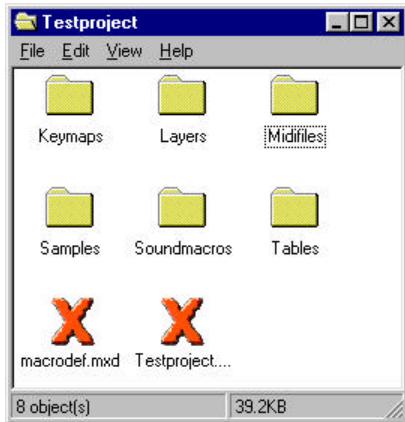
You can also play the instrument using the virtual MIDI-keyboard window inside *MusyX*. See later chapter for details.



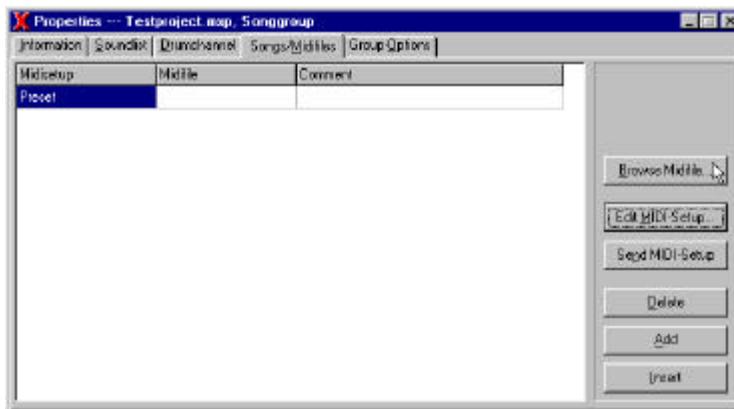
Recording a Sequence

Use your favorite MIDI-sequencer to record a short sequence with the new sound, using the slave program as synthesizer.

Save the finished Sequence as a MIDI-file (*.mid) into the Midifiles-folder of our Testproject.



Switch back to the **MusyX** Editor. Open the Properties window of the SongGroup and click [left mousebutton] on the panel called "Songs/Midifiles". Assign the Midifile to the Midisetup by double-clicking on the midifile field or by clicking on the "Browse Midfile" button.



A regular File dialog box appears. Here you have to change the directory, select your *.mid sequence that you recorded earlier, and click "Open". Your sequence is now attached to the SongGroup and you can close the Properties window.

Attaching midifiles to a SongGroup is not necessary as long as you work only with the slave, but it must be done before we convert the data for the game. Otherwise, the converter will not be able to find any midifile to include into the final data.

Looped Midi-files

Midifiles can be played like samples as oneshot (meaning only once) or they can be looped. If you do not want to loop the whole sequence or if your music contains a pause at the end before it starts from the beginning, you need to define two custom controllers inside your MIDI sequence.

These controllers determine, between them, the start and the end of the loop.

Controller 102 marks the left locator position or start of the loop.
Controller 103 marks the right locator position or end of the loop.

Controller 104 marks a position from where certain actions are possible, e.g. fading out, or beginning one song while the one playing is faded out. This controller makes it easier for the programmer who needs to combine various Midifiles in a game. Due to this controller, he does not have to worry about finding the exact right spot from where to fade out for example.

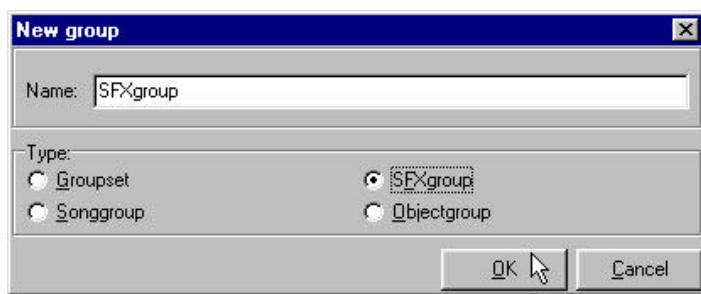
The values of the controllers are ignored for now, but should be kept zero.

Defining a Sound Effect

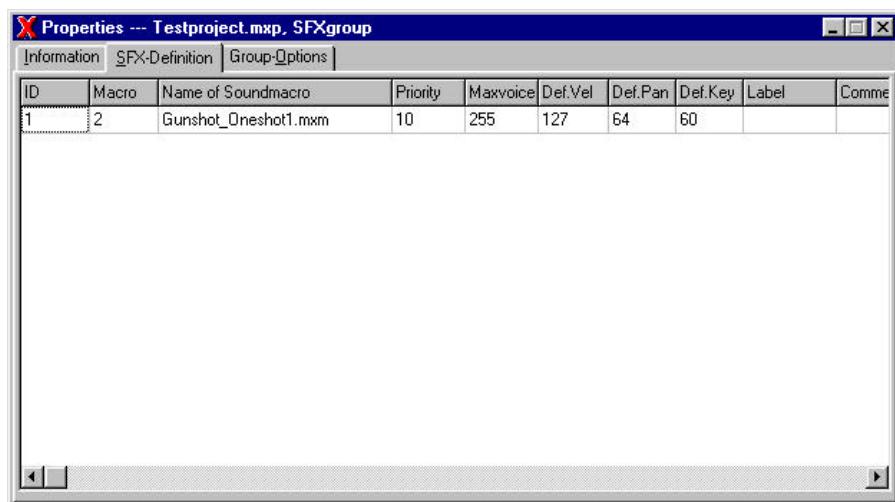
Sound effects (SFX) can be designed by importing a sample into your project and creating a SoundMacro, just the way you would create any instrument. The only difference is that unlike instruments, SFX are organized in so-called SFX-Groups. In these Groups you can edit a list of special settings for creating a SoundMacro.

Import the Sample “Gunshot.aif” from the Samples Directory (MusyX\GM-Set\GM-Set_44100\Samples\SFX) and create a SoundMacro using the Oneshot-Template. This is done exactly the same way you did the “Marimba” Sound.

Now create a new Group using the Groups pop-up menu. Logically, this time you select “SFX-Group” instead of a SongGroup.



Open the SFX-Group by double-clicking on its name and add the new SFX to the list. Use the pop-up menu [right mousebutton] to add a new entry first. Then double-click on the empty field under “Macro” and choose the “Gunshot_Oneshot.mxm” from the following dialog box. (This is the same procedure you followed in the SongGroup properties.)



Close the Properties Window and send the SFX-Group to the slave the same way you did the SongGroup.

How to Test a Sound Effect

After having sent the SFX-Group to the Slave, you can now easily play the SFX from the virtual MIDI-keyboard. The keynumber now represents the FX-ID, so you have to move the scrollbar of the MIDI-keyboard Window to the left to reach the appropriate key.

Every MIDI channel consists of 128 IDs, meaning that 128 different sound effects can be played at once. Since there is a total of 16 MIDI channels, you can directly test 2048 sound effects.

Saving Your Work

To save your Project select “Save” from the Project menu.

Finish!

The next step would be to convert the project using the external *MusyX* commandline tools.

Additional Tools

The Table Editor

The Table-object is an additional database that the user can prepare for specific SoundMacro-commands. These can access the table to obtain large amounts of data quickly and conveniently. Tables can only be referenced by specific SoundMacro-commands.

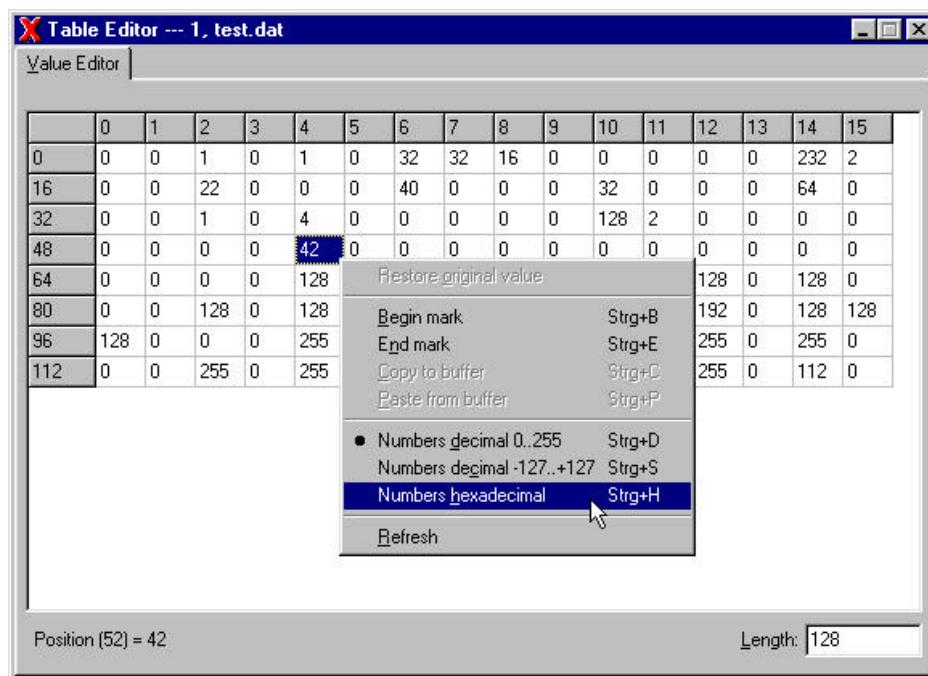
Tables can be used as curves for scaling the volume or to define ADSR Envelopes.

To create a new table, double-click on an empty space in the Tables directory. Then type in a name of a non-existing file. This will create the table. Next, double-click on the table to enter the table editor.

Using the Table Editor

To open the table editor, select the tables folder from the project window. Then right-click in the right view window and create a new table. Open the new table by double-clicking on its name in the right view window.

The table editor displays the data in 16 columns. By entering a new value in the length-field on the bottom right, the length of a table can be changed anytime.



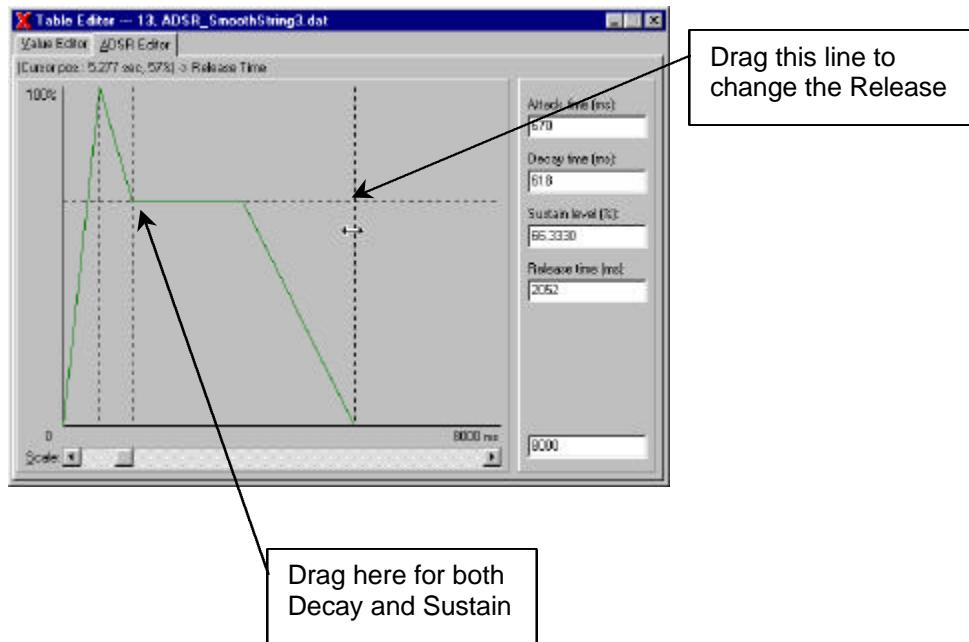
To change the data in a table, simply select the field using the mouse or the cursor keys and enter a value. The numerical Input and Output can be changed from the pop-up menu [right mousebutton].

Refer to the musician's reference of the SMaL language description concerning the use of tables in SoundMacros.

Using the ADSR Envelope Editor

If the length of a table equals 8 bytes exactly, you can access the ADSR section of the Table editor. This is the most common use of tables. Set the table length to 8, using the "length" option at the bottom of the Value Editor.

Here you can edit Attack, Decay, Sustain, and Release either by typing the values into the appropriate fields or by dragging the lines of the graphic display. The scale slider can be used to scale the graphical appearance of the Envelope.



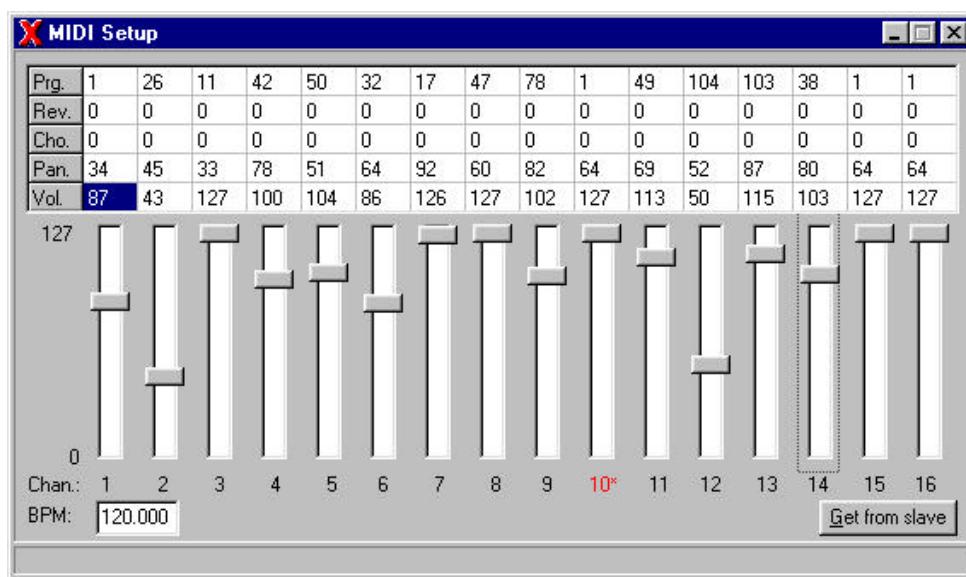
The MIDI Setup Window

What is the Purpose of this Window?

The MIDI setup is used to pre-set MIDI-programs, volumes, panning and other data for all 16 channels. This is comparable to the multimode setup of a Synthesizer. Every SongGroup of a project can contain several MIDI-setups, one for each song in this SongGroup.

How to Use the Window in Everyday Work

If you open the properties of a SongGroup you'll find a sub-window called "MIDI-setups/Songs", where you can add, remove and edit MIDI-setups. To open the MIDI setup window double-click on the name of the setup or use the "Edit" button, on the right side.



The Importance of the Window when Exporting Data

A MIDI-setup can be exported as simple data or as a MIDI standard file. This file can be used to import a setup into a MIDI sequencer.

Using Multiple MIDI Setups within One SongGroup

Each SongGroup can contain an unlimited number of Songs and MIDI Setups.

Please read also about looping MIDI-files in the section, “Recording a Sequence” of the *Walkthrough*.

The Virtual MIDI Keyboard

Why a Virtual Keyboard?

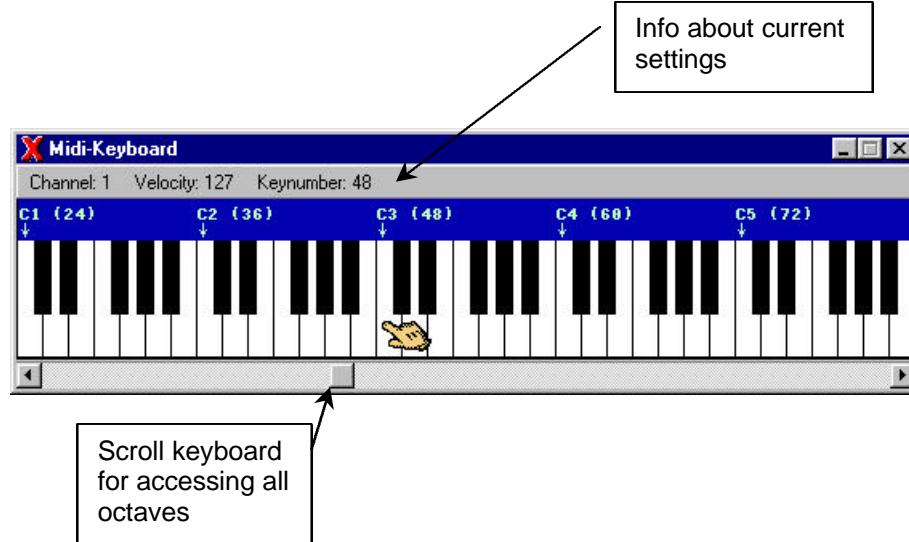
If you want to test a sound but you have no real MIDI keyboard connected, you can use the virtual keyboard.

In addition, it can help you check key numbers when building Layers or Keystrokes.

Using the Keyboard

When you move the mouse pointer over the virtual keyboard window the arrow will change into a hand. On the status bar above you can see the key number of the key where the hand is currently pointing.

Clicking the left mouse button on one of the keys sends this note to the slave. Clicking the right mouse button opens a pop-up menu where you can change the setup of the keyboard parameters.



Testing Sound Effects

One of the virtual keyboard's main purposes is to allow for quick tests of sound effects.

Sound effects are mapped to the keyboards – both the virtual and the physical one – the moment you send a SFX group over to the slave. The method in which they are mapped is both simple and effective.

First the key number and the MIDI channel are used to calculate the ID of the sound effect that is to be started. Once the ID is determined, the sound effect will be started.

The first seven bits of the ID represent the key number, bits eight to eleven the MIDI channel.

Every MIDI channel consists of 128 IDs, meaning that 128 different sound effects can be played at once. Since there is a total of 16 MIDI channels, you can directly test 2048 sound effects.

Limitations in Comparison to a Real Keyboard

The disadvantage of a virtual keyboard is that the user cannot play real music. Since only one key at a time can be pressed, there is no polyphony.

In addition, no controllers are available and the virtual keyboard cannot send MIDI commands to a sequencer.

The Network Master Window

What can be Controlled Using this Window?

The Network Master Window is used as a remote and displays status information about the Slave.

The Buttons and Fields:

- Connect Establishes a connection and resets the Slave
- Panic Resets all “hanging” voices on the Slave if needed
- Disconnect Break the connection to the Slave
- Voices Sets the maximum amount of Voices the Slave should play at once.

What Kind of Information is Displayed?

On the left side of the window, information about memory is displayed. You are informed about the available memory on the Slave system, about the space used by all sent objects and about remaining free memory space.

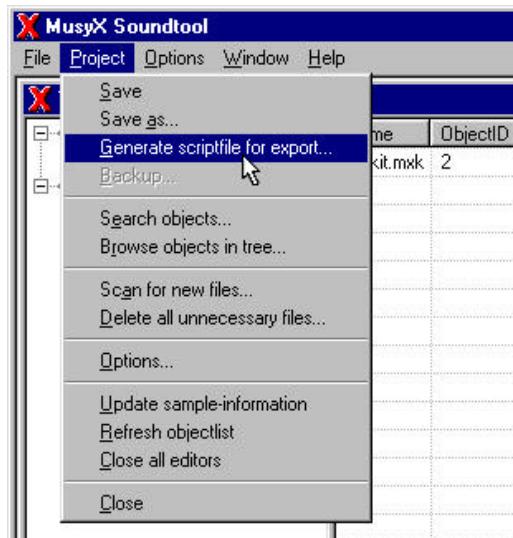
Data Conversion

General

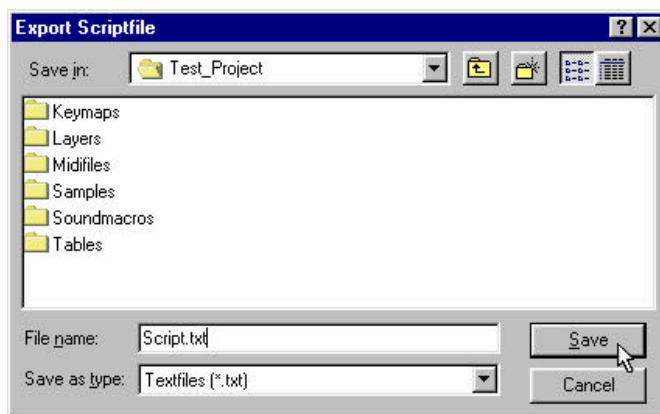
MusyX stores its data in a platform-independent and easy-to-work-with set of files. This format is not suitable to be used at runtime on the target platform. This is the reason why the data has to be converted before it can be used in the game application.

What the Musician has to do to Prepare the Data

The first step is to export the current project structure in a form that the command-line-based converter tools can understand. The so-called “script file” must be generated by the musician before handing the whole project's data structure over to the programmer.



Select the “Generate script file for export” item from the “Project” pull down menu.



A dialog box will appear that lets you select the name for the script file. If no extension is specified, .TXT will be added.

You are now ready to hand the whole project directory over to the programmer.

At the same time the musician should specify to the programmer which groups should be used at what time in the game application.

The Actual Data Conversion

There are two tools needed to convert the data.

GM2SONG.EXE is used to convert MIDI-1 files to the song file format used on the target platform.

MUConv.EXE is used to convert all other data. This program even calls GM2SONG.EXE. Because of this we will have a closer look at MUConv.EXE, only. GM2SONG.EXE must be in a location where it can be executed, but the user of *MusyX* will never come into direct contact with it.

In addition to a so-called description file (see below for details) and the export scripts defining the projects to be used as source, the tool can accept the following options.

-a

MUConv will generate an include file for easy reference to all IDs generated by the conversion process. By default the file will be written using C-syntax. You may use the **-a** option to write the file in assembler syntax instead.

-b

This option switches between little endian and big endian number representation. By default big endian is used (N64).

-p <path>

If the tool is not invoked from the project directory itself you may use this option to specify a search path for referencing all files from the project. This may be useful if you specify more than one project to be jointly converted. This option may be used multiple times and will add a search path to the search path list each time.

-t <sys>

This switch selects the target system for which the data should be converted. Be aware that this option does not set the appropriate endian mode on its own. Supported values for <sys> currently are either N64 or GB.

-s

Use this switch to disable the sample conversion step. If your project contains a lot of samples converting them may take a little while. You can avoid this by skipping this conversion step. You have to make sure though, that none of the samples have changed since you did the last complete conversion.

-d

Some platforms (e.g. N64) support different compression schemes. This switch causes MUConv.EXE to always use the default sample format no matter what the musician specified in the tool.

Tip:

On the N64, there are two formats supported: 16-Bit RAW and ADPCM compressed. The latter is the default format and should be used most of the time since the quality is usually very high and it imposes less performance overhead on the system.

-v

Enables the verbose mode. You'll get detailed information about every conversion step.

Example:

For a simple N64 project to be converted you may specify a command line like this:

```
MUConv -b -t N64 script.txt your.desc
```

You may specify multiple export scripts and search paths as source. All specified projects will be joined together and converted as one big project.

The description file may reference groups from all projects as if they were inside a single project. Be careful to avoid identical names across projects, though. This will cause the linking process to fail.

This feature can be used to allow a musician and a sound effects designer to work separately and still use both data sets as one big project in the game application.

The Description File

An important part of the information needed to do the data conversion comes from the “desc” or “description” file. This file specifies all parameters needed in addition to the project data to generate the final output files.

The file is divided into sections. A title line that looks like this marks each section: **[section name]**

Each line within the section specifies one parameter. Some sections just accept one parameter, some accept multiple parameters. The following is a list of sections and their parameters.

[project]

This section contains a case-sensitive list of all group names that should be included into the proj file.

[pool]

This section contains a case-sensitive list of all group names that should be included into the pool file.

[sampledir]

This section contains a case-sensitive list of all group names that should be included into the sdir file.

[samples]

This section contains a case-sensitive list of all group names that should be included into the samp file.

[outdir]

This section just contains one parameter. It specifies the output directory where all files produced by the tool will be stored.

[basename]

The base name, specified in this section, will be used to store all files except the song files and the include file. The section only contains one parameter.

[include]

The parameter of this section specifies the complete name of the include file to be written. The file will be stored in the output directory.

[stack]

This section contains the names of all the groups that should be taken into consideration for being in the “sound stack”, while considering which data is to be included in the various files (please see Programmer’s Reference).

It is necessary to specify the groups that are to be included in each output file.

For example:

One could try to save memory by simply putting all groups into the project file, while all other files just contain subsets of the data.

Together with the [stack] section it is possible to load only those files containing the groups really needed in a certain situation. In this case the [stack] section is used to tell the tool which groups to assume to be in the stack and in which order.

Multiple calls to the tool with different description files can now be used to generate all needed files.

Although this tool provides all these possibilities, in most cases you will not be forced to go to such extremes to generate your data. The simplest way is to include all needed groups in each output file and to skip the [stack] section all together.

Once you have edited the description file and selected the proper options, you are ready to start the tool.

It will produce four files that contain the project data and any number of song files that contain the data needed by the sequencer to playback the songs.

The files in the first group of files will all have the same base name, but different extensions. Here is a list of extensions and the type of data contained in the files.

.PROJ

This file contains all data about the structure of the project. This includes the information concerning what belongs to which group.

.POOL

The pool file contains all data except the samples. This includes, for example, all macros needed by the specified groups.

.SDIR

This file contains information about the location of all samples needed by the specified groups. The actual data is stored separately to make it possible to put it wherever the hardware allows. It may be stored in RAM where the CPU can reach it, but it may also be left in the ROM.

.SAMP

This file contains the actual sample data in the format needed on the target platform.

Once you have all these files converted you are ready to use them on your target platform.

MusyX Sample Program for N64

A complete **MusyX** project can be found in the "MusyX\example" directory. You will need to run the "Setup_Example.bat" file to copy the necessary libraries for your development environment. Supported environments are IRIX (5.3/6.2) and PC (Partner and SN 64).

The project directory is located in MusyX\Example\Data\Project.

To run the MUCONV data converter, use the makedata.bat file located in MusyX\Example\Data.

Sample output files (.proj, .pool, .sdir, .samp) are already located in MusyX\Example\Data\output.

Appendix 1 – N64 Musicians Reference

Table of Contents:

Appendix 1.1 - N64 Macro Commands	109
END	110
Structure Macros	111
STOP	111
SPLITKEY	112
SPLITVEL	113
WAIT_TICKS	114
LOOP	115
GOTO	116
WAIT_MS	117
PLAYMACRO	118
SENDKEYOFF	119
SPLITMOD	120
SPLITRND	121
TRAP_KEYOFF	122
UNTRAP_KEYOFF	123
IF_EQUAL	124
IF_LESS	125
Sample Macros.....	126
STARTSAMPLE	126
STOPSAMPLE	127
Volume Macros	128
SETADSR	128
SCALEVOLUME	129
ENVELOPE	130
FADE-IN	131
Pitch Macros	132
PORTLAST	132
RNDNOTE	133
ADDNOTE	134
SETNOTE	135
LASTNOTE	136
VIBRATO	137
PITCHSWEEP1	138
PITCHSWEEP2	139
SETPITCH	140

Control Macros.....	141
PIANOPAN	141
PANNING	142
KEYOFF	143
 Special Macros.....	 144
ADDAGECOUNT	144
SETAGECOUNT	145
SENDFLAG	146
REV_LEVEL	147
SETPRIORITY	148
ADDPRIORITY	149
AGECNTSPEED.....	150
AGECNTVEL.....	151
ADD_VARS	152
SUB_VARS	154
MUL_VARS	155
DIV_VARS	156
ADDI_VARS	157
SET_VAR	158
 Setup Macros	 159
PORTAMENTO	159
PITCHWHEELR.....	160
VOL_SELECT.....	161
PAN_SELECT	162
PitchW_SELECT	163
ModW_SELECT	164
PEDAL_SELECT	165
PORTA_SELECT	166
REVERB_SELECT	167
SPAN_SEL	168
DOPPLER_SEL.....	169
SETUP_LFO.....	170
 Appendix 1.2 - N64 Macro Templates	 171
?_ONESHOT	172
?_LOOPED.....	174

Appendix 1.1 - N64 Macro Commands

This part contains descriptions of all Macro Commands used by *MusyX* for the Nintendo64.

END

End of the Macro

\$00	END								
------	-----	--	--	--	--	--	--	--	--

Description:

This is always the last macro command. It cannot be deleted from the macro. It terminates the macro permanently.

Structure Macros

STOP

Similar to END, but can be used as a return command

Type: *Structure*

\$01		STOP	Return flag <i>mode</i>							
------	--	------	----------------------------	--	--	--	--	--	--	--

Description:

If mode is set to zero, this macro command has the same functionality as END. In contrast to END, it can be placed anywhere in the macro. If mode is set to a non-zero value, the macro will not be terminated and macro execution will continue from the last used GOTO command.

Parameters:

mode = Return flag (0 = on, 1 = off)

SPLITKEY

Splits the macro flow depending on the midikey

Type: *Structure*

	Keynumber	SoundMacro ID	SoundMacro step		
\$02	SPLITKEY	<i>key</i>	<i>macro</i>	<i>step</i>	

Description:

This command is used to conditionally change the flow of execution in the current macro. The macro program will jump to the given macrostep inside the specified macro, if the current key is higher than or equal to the key parameter.

Parameters:

- key = This parameter (0-127) specifies a key number to compare against. If the key you play is higher or the same as this key, the macro will jump, otherwise it resumes.
- macro = The ID of the macro to jump to
- step = The step number inside the macro to jump to

SPLITVEL

Splits the macro flow depending on the velocity

Type: *Structure*

	Velocity	SoundMacro ID	SoundMacro step		
\$03	SPLITVEL	<i>velocity</i>	<i>macro</i>	<i>step</i>	

Description:

This command is used to conditionally change the flow of execution in the current macro. The macro program will jump to the given macrostep inside the specified macro, if the current velocity is higher than or equal to the velocity parameter.

Parameters:

- velocity = Specifies the velocity to compare the current velocity against.
If the current velocity is higher or the same, the macro will
jump, otherwise it will resume.
- macro = The ID of the macro to jump to
- step = The step number inside the macro to jump to

WAIT_TICKS

Wait, depending on different conditions

Type: Structure

		Keyoff	Random	Sampleend		ms switch	Ticks/millisecond.
\$04	WAIT_TICKS	key release	random	sample end		ms flag	time

Description:

The execution of the current macro will be suspended until the given time is elapsed. By default, the time is specified in ticks. If the ms flag is set the time will be specified in ms. Independent from the selected time format a value of \$FFFF (65535) will cause an endless wait.

If one of the other flags is set, execution will resume as soon as one of the corresponding conditions becomes true or the time given has elapsed.

Key release	Wait until a keyoff is sent to the macro
Sampleend	Wait until the sample has reached its end
Random	The time will be used as a maximum to generate a randomized delay

Parameters:

key release	=	If this flag is set, the macro will resume after a keyoff is received
random	=	If this flag is set, the macro will resume after a random time is elapsed. In this case the ticks/millisecond. parameter defines the maximum wait time
sampleend	=	If this flag is set, the macro will resume when the sample reached its end (this works only with oneshot samples)
ms flag	=	This flag switches the mode of the time parameter
Ticks/Millisecl.	=	The wait time specified in ticks or milliseconds

LOOP

Loop back to a macrostep

Type: *Structure*

\$05	LOOP	Keyoff <i>key release</i>	Random <i>random</i>	Sampleend <i>sample end</i>	SoundMacro step <i>step</i>	Times <i>times</i>
------	------	------------------------------	-------------------------	--------------------------------	--------------------------------	-----------------------

Description:

Loop to the specified location within the current macro n-times. If one of the flags is set, the loop may be executed fewer than the specified number of times. A value of \$FFFF (65535) will cause an endless loop.

Key release	Loop until a keyoff is sent to the macro
Sampleend	Loop until the sample has reached its end
Random	The number of loops will be used as a maximum to generate a randomized counter

Parameters:

key release	=	If set to on and a keyoff is received the command will not loop but proceed with the next step in the macro
Random	=	If set to on the command will loop random times, where the times parameter specifies the maximal count of loops
sampleend	=	If set to on the command will not loop when the sample reached its end (this works only with oneshot samples)
step	=	This defines the macrostep, to which the command loops
times	=	The number of loops to be performed. A times value of 65535 will cause an endless loop, if none of the other conditions apply.

Hint: Loops cannot be nested!

GOTO

Jump to another macro

Type: *Structure*

		SoundMacro ID <i>Macro</i>	SoundMacro step <i>Step</i>		
\$06	GOTO				

Description:

Performs an unconditional jump to the specified location. Note that command \$01 STOP, has the option to jump back to the position after the GOTO command. This can be used to create a sub-macro.

Parameters:

macro = The macro ID to jump to

step = The step inside the specified macro to jump to

WAIT_MS

Wait, depending on various conditions

Type: *Structure*

\$07	WAIT_MS	Keyoff <i>key release</i>	Random <i>random</i>	Sampleend <i>sample end</i>			Millisec. <i>ms</i>
------	----------------	------------------------------	-------------------------	--------------------------------	--	--	------------------------

Description:

The execution of the current macro will be suspended until the given time has elapsed. By default, the time is specified in ms. A value of \$FFFF (65535) will cause an endless wait.

If one of the other flags is set, execution will resume as soon as one of the corresponding conditions becomes true or the time given has elapsed.

Key Release	Wait until a keyoff is send to the macro
Sampleend	Wait until the sample has reached its end
Random	The time will be used as a maximum to generate a randomized delay

Parameters:

key release	=	If this flag is set, the macro will resume after a keyoff is received
random	=	If this flag is set, the macro will resume after a random time has elapsed. In this case the ticks/millisec. parameter defines the maximum wait time
sampleend	=	If this flag is set, the macro will resume when the sample reached its end (this works only with oneshot samples)
ms.	=	The wait time specified in milliseconds

PLAYMACRO

Starts another macro

Type: *Structure*

	Addnote	SoundMacro ID	SoundMacro step	Priority	Max Voices
\$08	PLAYMACRO	<i>addnote</i>	<i>macro</i>	<i>step</i>	<i>priority</i>

Description:

Starts another macro in parallel to the current one. Since it will be started like any other macro, the normal delays etc. apply. The macro will be passed the priority and the maxVoc. value specified. The key is calculated by adding addnote to the original key of the macro starting the new one. The new macro may be started at any macrostep, using the step parameter. See also command \$09 SENDKEYOFF.

Parameters:

- addnote = A keyshift or transpose value (-128 to 127) can be added to start the new voice with a different note.
- macro = The ID of the new macro to start
- step = The step inside the new macro to start
- priority = Defines the priority of the new voice
- maxVoc. = The maximum count of voices that this new macro can allocate.

SENDKEYOFF

Send a keyoff to the specified macro

Type: *Structure*

	Addnote	SoundMacro ID					
\$09	SENDKEYOFF	addnote	macro				

Description:

Send a keyoff to the specified macro. Since the macro is only identified by its ID and the key value, which is calculated by adding the parameter addnote to the original key of the current macro, multiple macros may be found. In this case, the keyoff is sent to all macros encountered. This command is mainly used to signal a midi-keyoff to other previously started macros by the PLAYMACRO command.

Parameters:

addnote = Please see the description of the PLAYMACRO command
macro = The ID of the macro that will receive the keyoff

SPLITMOD

Splits the macro flow depending on modwheel

Type: Structure

	Mod value	SoundMacro ID	SoundMacro step		
\$0A	SPLITMOD	<i>mod.</i>	<i>macro</i>	<i>step</i>	

Description:

This command is used to conditionally change the flow of execution in the current macro. The macro program will jump to the given macrostep inside the specified macro, if the current modulation value is higher than or equal to the mod. parameter.

Parameters:

- mod. = This defines the point where the split occurs (0-127)
macro = The ID of the macro to jump to
step = The step number inside the macro to jump to

SPLITRND

Splits the macro flow depending on a generated random value

Type: *Structure*

RND value	SoundMacro ID	SoundMacro step		
\$13 SPLITRND	<i>rnd</i>	<i>macro</i>	<i>step</i>	

Description:

This command is used to conditionally change the flow of execution in the current macro. The macro program will jump to the given macrostep inside the specified macro, if the generated random value is higher than or equal to the *rnd* parameter.

Parameters:

- rnd = The higher this value is, the less is the chance that the jump will be performed
- macro = The ID of the macro to branch to
- step = The step number inside the macro to branch to

TRAP_KEYOFF

Registers a jump destination in a macro if a keyoff occurs

Type: *Structure*

	SoundMacro ID	SoundMacro step		
\$28	TRAP_KEYOFF	<i>macroID</i>	<i>step</i>	

Description:

Registers a jump destination in a macro if a keyoff occurs.

Parameters:

macro = The macro ID to jump to

step = The step inside the specified macro to jump to

UNTRAP_KEYOFF

Remove a keyoff trap

Type: *Structure*

\$29	UNTRAP_KEYOFF							
------	---------------	--	--	--	--	--	--	--

Description:

Removes a previously registered keyoff trap.

IF_EQUAL

Goto MacroStep if condition is true

Type: Structure

	Ctrl	A==	Ctrl	B	Not	Sound	Macro Step
\$70 IF_EQUAL	Var/Ctrl	A	Var/Ctrl	B	Not		MacroStep

Description:

If the condition is TRUE, the execution of the SMaL program will be continued at *MacroStep*. A jump outside the current macro is not possible. The condition evaluated is a simple comparison of the values of variables A and B. If the Not field is “Off”, the condition is TRUE as soon as both values are identical. If Not is set to “On”, the condition is TRUE if both values are not equal.

For details about variables see ADD_VARS.

Parameters:

- Var/Ctrl = Controller A switch (OFF = Variable, ON = extended controller)
- A = Variable / Controller A
- Var/Ctrl = Controller B switch (OFF = Variable, ON = extended controller)
- B = Variable / Controller B
- Not = logical not
- MacroStep = MacroStep number to jump to inside the current macro

IF_LESS

Goto MacroStep if condition is true

Type: *Structure*

	Ctrl	A<	Ctrl	B	Not	Sound	Macro Step
\$71 IF_LESS	Var/Ctrl	A	Var/Ctrl	B	Not		<i>MacroStep</i>

Description:

If the condition is TRUE, the execution of the SMaL program will be continued at *MacroStep*. A jump outside the current macro is not possible. The condition evaluated is a simple comparison of the values of variables A and B. If the Not field is “Off”, the condition is TRUE as soon as A < B is satisfied. If Not is set to “On”, the condition is TRUE if A >= B is satisfied.

For details about variables see ADD_VARS.

Parameters:

- Var/Ctrl = Controller A switch (OFF = Variable, ON = extended controller)
- A = Variable / Controller A
- Var/Ctrl = Controller B switch (OFF = Variable, ON = extended controller)
- B = Variable / Controller B
- Not = logical not
- MacroStep = MacroStep number to jump to inside the current macro

Sample Macros

STARTSAMPLE

Start a sample

Type: Sample

		Sample-ID	Vel. Scale	Sample Start Offset
\$10	STARTSMP	<i>smpID</i>	<i>Mode</i>	<i>Offset</i>

Description:

Starts the sample playback of the sample specified by *smpID*. An offset inside the sample may be specified, but is not supported on all hardware platforms. (Startsample playing from offset is not supported under Game Boy.) If it is supported, it is always specified in samples, not bytes. If no ADSR was previously specified, a standard ADSR will be used, that avoids click-sounds as much as possible but starts & stops the sample almost immediately.

Parameters:

- | | | |
|--------|---|--|
| smpID | = | The ID of the Sample to be started |
| mode | = | 0=apply offset directly,
1=scale offset with negative velocity (higher velocity results in smaller offset),
2=scale offset with positive velocity (higher velocity results in higher offset) |
| offset | = | The offset in sample-units. If mode = 1 or 2 the offset defines the maximal range |

Hint: Use an ADSR with at least 1ms fade-in time or the FADEIN command with 18ms fade time to avoid clicking if you use the offset function!

STOPSAMPLE

Stops the sample playback immediately

Type: *Sample*

\$11	STOPSAMPLE								
------	------------	--	--	--	--	--	--	--	--

Description:

Stops the sample playback immediately by sending a keyoff and setting the ADSR to a very short release time. Click-sounds will be avoided as much as possible.

Volume Macros

SETADSR

Hardware ADSR Envelope

Type: *Volume*

		Table-ID (ADSR)						
\$0C	SETADSR	Table						

Description:

The data from the specified table will be used to define an ADSR to be used with the voice. The editor presents a graphical edit dialog to define tables containing the needed data.

Parameters:

table = This references the ID of the table that contains the ADSR

SCALEVOLUME

Scales the velocity passed to the macro by the sequencer or the effect handler to calculate a new volume for the current voice

Type: Volume

	Scale	Add	Table-ID (Curve)	Org Vol.		
\$0D SCALEVOLUME	Scale	Add	curve	org.vol		

Description:

Calculates a new volume for the current voice by scaling the velocity. The velocity is either passed to the macro by the sequencer (via midi) or the effect handler. A scale of 127 equals 100%. Smaller values scale down and larger scale up. In addition to the simple scale, an offset can be specified in the 'add' parameter. The result of this calculation can be passed through a curve that will act as a translation table. A value of zero in the curve parameter will disable this feature. The new volume is calculated either using the current velocity (Org.Vel = 0) or the original velocity (Org.Vol = 1).

Parameters:

- scale = The scaling factor of the velocity
- add = A fixed offset can be added to the volume
- curve = This is a table ID of a volume translation curve (0=linear)
- org.vel = If this switch is set to on the original velocity (when the macro was started) is used instead of the current velocity/volume

ENVELOPE

Starts a software envelope

Type: Volume

\$0F	ENVELOPE	Scale	Add	Table-ID (Curve)	ms switch	Ticks/Millisec.
		Scale	add	curve	ms flag	ticks/ms

Description:

Starts a software envelope. The velocity of the current macro will be faded to the new one, in the time specified by the *ticks/ms* parameter. If the *ms flag* is set, ms will be used instead of ticks. The new volume is calculated just as described in SCALEVOLUME. The volume sweep may be of lower quality than the hardware ADSR.

Parameters:

- scale = The scaling factor of the velocity
- add = A fixed offset can be added to the velocity. Clipping is applied if the result exceeds 127
- curve = This is a table ID of a volume translation curve (0=linear)
- ms flag = This switches the following parameter from ticks to milliseconds
- ticks/ms = Specifies fade time in either ticks or milliseconds

FADE-IN

Starts a software fade-in envelope

Type: Volume

\$14	FADE-IN	Scale	Add	Table-ID (Curve)	ms switch	Ticks/Millisec.
		Scale	Add	Table	ms flag	ticks/ms

Description:

Starts a software fade-in envelope. The velocity of the current macro will be faded from zero to the new one in the time specified by the *ticks/ms* parameter. If the *ms flag* is set, ms will be used instead of ticks. The new volume is calculated just as described in SCALEVOLUME. The volume sweep may be of lower quality than the hardware ADSR.

Parameters:

- scale = The scaling factor of the velocity
add = A fixed offset can be added to the velocity. Clipping is applied if the result exceeds 127
table = This is a table ID of a volume translation curve (0=linear)
ms flag = This switches the following parameter from ticks to milliseconds
ticks/ms = Specifies fade time in either ticks or milliseconds

Pitch Macros

PORLAST

Not implemented yet

Type: *Pitch*

	Add	Detune		ms switch	Ticks/Millisec.
\$16	PORLAST	---	---	---	---

Description:

Not implemented yet.

Parameters:

RNDNOTE

Sets random pitch

Type: *Pitch*

	Note Lo <i>note-lo</i>	Detune <i>detune</i>	Note Hi <i>note-hi</i>	Fixed/Free <i>fix/free</i>			
\$17	RNDNOTE						

Description:

Sets random pitch. *Note-lo* is the lower end of the range, *note-hi* the upper end. The *detune* value will be added after the random pitch is calculated. It is specified in cents. If the *fix/free* flag is set, the pitch will be generated freely inside the range without respect to any key steps. If *rel/abs* is set, the *note-lo* parameter specifies how many keys below the current key the range should start, while *note-hi* defines the size of the upper range.

Parameters:

- | | | |
|----------|---|--|
| note-lo | = | Lower end of the range |
| detune | = | Applies a detune after the note is calculated |
| note-hi | = | Upper end of the range |
| fix/free | = | OFF = the random pitch is quantized to note values
ON = the pitch is calculated freely within the range |

ADDNOTE

Recalculates the current pitch by adding keysteps to the current key

Type: *Pitch*

\$18	ADDNOTE	Add <i>add</i>	Detune <i>detune</i>	Org Key <i>org.key</i>	ms switch <i>ms flag</i>	Ticks/Millisecl. <i>ticks/ms</i>
------	---------	-------------------	-------------------------	---------------------------	-----------------------------	-------------------------------------

Description:

Recalculates the current pitch by adding *add* keysteps to the current key (or original key, if the Original Key flag is set to on) and applying the *detune* value in cents. The *add* parameter is a signed value. The last two parameters are zero by default. If they are a non-zero value they will be used as in WAIT, to suspend the execution of the current macro for a given time interval.

Parameters:

- add = This can be used to transpose the midi-key to a new note
- detune = A detune of +/-99 cent can be applied
- org.key = If set to on the original midi-key is used to calculate the new pitch
- ms flag = This flag switches the mode of the time parameter
- Ticks/Millisecl. = A wait time specified in ticks or milliseconds. If 0 the macro proceeds directly to the next step

SETNOTE

Sets note by a fixed key

Type: *Pitch*

\$19	SETNOTE	Key key	Detune Detune	ms switch <i>ms flag</i>	Ticks/Millisecl. ticks/ms
------	---------	------------	------------------	-----------------------------	------------------------------

Description:

Calculating a new pitch by setting the current key to the new value specified by *key*. After this, the *detune* specified in cents will be applied. The last two parameters are zero by default. If they are a non-zero value, they will be used as in WAIT, to suspend the execution of the current macro for a given time interval.

Parameters:

- key = A fixed key in the normal midi key range (0-127)
detune = A detune of +/-99 cent can be applied
ms flag = This flag switches the mode of the time parameter
Ticks/Millisecl. = A wait time specified in ticks or milliseconds. If 0 the macro proceeds directly to the next step

LASTNOTE

Adds note to last note on current channel

Type: *Pitch*

\$1A	LASTNOTE	Add <i>add</i>	Detune <i>detune</i>	ms switch <i>ms flag</i>	Ticks/Millisecl. <i>ticks/ms</i>
------	----------	-------------------	-------------------------	-----------------------------	-------------------------------------

Description:

Recalculates the current pitch/key by adding *add* keysteps to the last key played on this MIDI (!) channel and applying the *detune* value in cents. The *add* parameter is a signed value. The last two parameters are zero by default. If they are a non-zero value they will be used as in WAIT, to suspend the execution of the current macro for a given time interval.

Parameters:

- add = This can be used to transpose the midi-key to a new note
- detune = A detune of +/-99 cent can be applied
- ms flag = This flag switches the mode of the time parameter
- Ticks/Millisecl. = A wait time specified in ticks or milliseconds. If 0 the macro proceeds directly to the next step

VIBRATO

Adds a vibrato to the voice currently used to play the macro

Type: *Pitch*

	Level note	Level fine	Modwheel flag	ms switch	Ticks/Millisecl.
\$1C VIBRATO	<i>Level</i>	<i>levelfine</i>	<i>mod.</i>	<i>ms flag</i>	<i>ticks/ms</i>

Description:

Adds a vibrato to the voice currently used to play the macro. Vibrato means that the pitch is modulated by a triangular waveform with a period specified by *ticks*. If the *ticks/ms* flag is set, the period is given in ms instead of ticks. A period of zero will disable the vibrato. The number of keysteps that it should go up or down gives the amplitude of the waveform. The *level* parameter is a signed value. If it is negative the pitch offset will go down first. If it is positive the offset will go up first. *levelfine* is specified in cents. If the *mod.* flag is set to on, the values from the modulation wheel will be used to scale the vibrato. "Off" disables any scaling by controllers.

Parameters:

- | | | |
|-----------|---|--|
| level | = | This is the level in note-steps (+/-12) |
| levelfine | = | The fine level (+/-99 cents) |
| mod. | = | On = Scale the level with the modwheel controller |
| ms flag | = | This switches the following parameter from ticks to milliseconds |
| ticks/ms | = | Specifies period time in either ticks or milliseconds |

PITCHSWEEP1

Add a sweep to the pitch

Type: *Pitch*

\$1D	PitchSWEEP1	Times	Add	ms switch	Ticks/Millisecl.
		Times	Add-value	ms flag	ticks/ms

Description:

Adds the add-value n-times to the current pitch. After this, the action starts again at the original pitch. To stop the effect, set add to zero. The last two parameters are zero by default. If they are a non-zero value, they will be used as in WAIT, to suspend the execution of the current macro for a given time interval. There are actually two commands of this kind (PITCHSWEEP1 & PITCHSWEEP2). They work independently from each other and combine to perform very nice effects, if used simultaneously.

Parameters:

- Times = This defines how many frames the add-value will be applied
- Add-value = The value to be added to the pitch per frame
- ms flag = This flag switches the mode of the time parameter
- Ticks/Millisecl. = A wait time specified in ticks or milliseconds. If 0 the macro proceeds directly to the next step

PITCHSWEEP2

Add a sweep to the pitch

Type: *Pitch*

\$1E	PitchSWEEP2	Times	Add	ms switch	Ticks/Millise.
		<i>Times</i>	<i>Add-value</i>	<i>ms flag</i>	<i>ticks/ms</i>

Description:

Adds the add-value n-times to the current pitch. After this, the action starts again at the original pitch. To stop the effect, set add to zero. The last two parameters are zero by default. If they are a non-zero value, they will be used as in WAIT, to suspend the execution of the current macro for a given time interval. There are actually two commands of this kind (PITCHSWEEP1 & PITCHSWEEP2). They work independently from each other and combine to perform very nice effects, if used simultaneously.

Parameters:

- Times = This defines how many frames the add-value will be applied
- Add-value = The value to be added to the pitch per frame
- ms flag = This flag switches the mode of the time parameter
- Ticks/Millise. = A wait time specified in ticks or milliseconds. If 0 the macro proceeds directly to the next step

SETPITCH

Sets the pitch directly

Type: *Pitch*

\$1F	SETPITCH	Frequency in Hz Hz (24bit)	Fine (<i>fine</i>)		
------	----------	-------------------------------	-------------------------	--	--

Description:

Sets the frequency to be used to playback a sample, directly. The *fine* parameter is not supported by all platforms, since most platforms do not have a fine resolution in selecting the playback frequency (10Hz steps are common).

Parameters:

Hz = The coarse frequency in Hz (0-88200)

fine = A fine resolution parameter (0-65535)

Control Macros

PIANOPAN

Piano stereo panning

Type: *Control*

\$0B	PIANOPAN	Scale scale	Centerkey cen.key	Centerpan cen.pan				
------	----------	----------------	----------------------	----------------------	--	--	--	--

Description:

This macro command is especially designed to give instruments like a piano a naturalistic stereo behavior. The current key will be used to calculate a panning position. First, the key is converted to an offset by subtracting the cen.key. Next, the scale is applied. A scale of 127 will give you the full range, while a scale of 0 will eliminate any offset. Finally, the cen.pan is added. This last value can be viewed as the position of the instrument in the room. Since the final values may exceed the normal panning range, illegal values are clipped. A negative scale will invert the stereo panorama.

Parameters:

scale = This scales the range of the panning

cen.key = This defines the middle key of the panning

cen.pan = An offset of the panning range

PANNING

Sets the panning to be used with the macro

Type: Control

	Pan position	Time ms	Width				
\$0E PANNING	<i>pan.pos</i>	<i>Time</i>	<i>width</i>				

Description:

Sets the panning to be used with the macro. If no panning is specified, the default center panning will be used. The pan.pos parameter specifies the position (0=Left, 64=Center, 127=Right). The time and width parameters enable an automated pan-sweep. The panning will move from the current location to the new one specified by the signed offset within time ms. If these parameters are zero, the new panning will be set immediately.

Parameters:

- pan.pos = like midi-panning (0-127), where 0 is totally left, 64 is center and 127 is totally right
- time = Auto panning slide time
- width = An offset to the pan position to which the auto panning slides to

KEYOFF

Sends a keyoff to the currently used voice

Type: *Control*

\$12	KEYOFF								
------	--------	--	--	--	--	--	--	--	--

Description:

Sends a keyoff to the currently used voice, but does not change the ADSR. This command should be used instead of STOPSAMPLE, to finish sample playback if any ADSR has been set.

Special Macros

ADDAGECOUNT

Add a value to the age-counter

Type: Special

\$30	ADDAGECOUNT		add					
------	-------------	--	-----	--	--	--	--	--

Description:

Adds a signed number to the age-counter of the current voice. This allows customized priority handling.

Parameters:

add = Signed value to be added to the age-counter. In order to make a voice "older" a negative value has to be added (-32768 to 32767)

SETAGECOUNT

Set age-counter

Type: Special

\$31	SETAGECOUNT	Counter <i>counter</i>						
------	-------------	---------------------------	--	--	--	--	--	--

Description:

Directly sets the age-counter of the current voice. This allows customized priority handling.

Parameters:

counter = This sets the age value (0-65535). The lower the value, the older the voice

SENDFLAG

Sends a flag to the game program

Type: Special

		Flag-ID	Value													
\$32	SENDFLAG	<i>num</i>	<i>value</i>													

Description:

Sends a flag to the game program. This feature is used mainly to signal certain events to the game program. After startup, the values are all zero. There are 16 values.

Parameters:

num = The flag ID (0-15)

value = The number to be set (0-255)

REV_LEVEL

Set reverb level

Type: Special

\$34	REV_LEVEL	Scale	Add								
		scale	add								

Description:

Sets or scales the reverb level for the current voice. If the reverb effect engine is enabled, all voices are passed through this engine with a send level defined by midi controller 91 (Effect1Depth) or as predefined by the reverb setting in the midi-setup of each MusyX-song. The scale and add parameter can be used to change or override those settings. This is especially useful for drum kits, where each drum may need a different reverb level. If both values are 0, the reverb engine is bypassed for the voice currently used so that, e.g. a base drum can be kept "dry".

Parameters:

scale = Scales the reverb level of the current midi channel

add = Adds a fixed value to the scaled reverb level

SETPRIORITY

Directly sets the priority

Type: Special

\$36		SETPRIORITY	Priority <i>prio</i>								
------	--	-------------	-------------------------	--	--	--	--	--	--	--	--

Description:

Directly sets the priority. This allows customized priority handling.

Parameters:

prio = Sets the priority of the current macro

ADDPRIORITY

Adds a value to priority

Type: *Special*

\$37	ADDPRIORITY	Add	add						
------	-------------	-----	-----	--	--	--	--	--	--

Description:

Adds a signed number to the priority. This allows customized priority handling.

Parameters:

add = A signed number to be added to the current priority

AGECNTSPEED

Sets age-counter speed

Type: Special

					Time until Zero
\$38	AGECNTSPEED				Time

Description:

Changes the speed by which the age-counter is decremented. By default the counter will reach zero after about 18 minutes. The *time* value sets the time in milliseconds the counter will need to decrement to zero from its **current** value. If *Time* is set to zero, the priority counter will not be changed over time at all.

Parameters:

time = The time (0-1080000) in milliseconds to reach 0

AGECNTVEL

Sets the age-counter using velocity

Type: Special

\$39	AGECNTVEL	Age Base <i>AGE Base</i>	Age Scae <i>AGE Scale</i>				
------	-----------	-----------------------------	------------------------------	--	--	--	--

Description:

Sets the age-counter of the current voice, calculated from a base number and the midi velocity. This allows customized priority handling.

Parameters:

AGE Base = The age base value (0-65535)

AGE Scale = The scaling factor for the velocity (0-65535) to be added

ADD_VARS

Add variables

Type: Special

\$60	ADD_VARS	Ctrl Var/Ctrl	A= A	Ctrl Var/Ctrl	B+ B	Ctrl Var/Ctrl	C C	
------	----------	------------------	---------	------------------	---------	------------------	--------	--

Description:

The current values of variable B and C are added together and stored in variable A. The Var/Ctrl switches select whether a variable or a controller should be accessed. If the switch is set to “On”, the corresponding value index is used as an extended controller number.

Parameters:

- | | | |
|----------|---|--|
| Var/Ctrl | = | Controller A switch (OFF = Variable, ON = extended controller) |
| A | = | Variable / Controller A |
| Var/Ctrl | = | Controller B switch (OFF = Variable, ON = extended controller) |
| B | = | Variable / Controller B |
| Var/Ctrl | = | Controller C switch (OFF = Variable, ON = extended controller) |
| C | = | Variable / Controller C |

About variables:

All variable-handling commands can work with extended MIDI controllers and variables. Variables are referenced by specifying a number that identifies the variable to be used. There are local and global variables. While local variables are just accessible from the current macro and are initialized to zero each time the macro is started, global variables can be accessed by all macros and even the application program. They are just initialized to zero when the system is initialized.

Local variables are identified by indices 0 to 15, while indices 16 to 31 specify global variables.

All types of variables – including controllers – are handled as 16-bit signed values. All operations are saturated, meaning that results of mathematical operations are clipped against the maximum and minimum values.

When writing to or reading from controllers, one must be aware that all controllers are handled as 14-bit values, even if they are just 8-bits in size. For example, a MIDI volume of 127 would be represented as 16256 (127×128). This is done so that generalized routines can be written without having to watch out for all the different MIDI controller sizes. The 14-bit value is always used as an unsigned quantity. E.g. a neutral pitchbend position would be 8192, not zero.

All extended MIDI controllers with the exception of the two LFOs may be written to, external MIDI data may overwrite these values at any time, though.

SUB_VARS

Subtract variables

Type: Special

\$61	SUB_VARS	Ctrl Var/Ctrl	A= A	Ctrl Var/Ctrl	B- B	Ctrl Var/Ctrl	C C	
------	----------	------------------	---------	------------------	---------	------------------	--------	--

Description:

The current value of variable C is subtracted from variable B and the result is stored in variable A. The Var/Ctrl switches select whether a variable or a controller should be accessed. If the switch is set to "On" the corresponding value index is used as an extended controller number.

For details about variables see ADD_VARS.

Parameters:

- Var/Ctrl = Controller A switch (OFF = Variable, ON = extended controller)
A = Variable / Controller A
Var/Ctrl = Controller B switch (OFF = Variable, ON = extended controller)
B = Variable / Controller B
Var/Ctrl = Controller C switch (OFF = Variable, ON = extended controller)
C = Variable / Controller C

MUL_VARS

Multiply variables

Type: Special

	Ctrl	A=	Ctrl	B	Ctrl	C	
\$62	VAR_MUL	Var/Ctrl	A	Var/Ctrl	B	Var/Ctrl	C

Description:

The current values of variable B and C are multiplied and the result is stored in variable A. The Var/Ctrl switches select whether a variable or a controller should be accessed. If the switch is set to “On”, the corresponding value index is used as an extended controller number.

For details about variables see ADD_VARS.

Parameters:

- Var/Ctrl = Controller A switch (OFF = Variable, ON = extended controller)
- A = Variable / Controller A
- Var/Ctrl = Controller B switch (OFF = Variable, ON = extended controller)
- B = Variable / Controller B
- Var/Ctrl = Controller C switch (OFF = Variable, ON = extended controller)
- C = Variable / Controller C

DIV_VARS

Divide variables

Type: Special

\$63	DIV_VARS	Ctrl Var/Ctrl	A= A	Ctrl Var/Ctrl	B/ B	Ctrl Var/Ctrl	C C	
------	----------	------------------	---------	------------------	---------	------------------	--------	--

Description:

The current value of variable B is divided by the value of variable C and the result is stored in variable A. The Var/Ctrl switches select whether a variable or a controller should be accessed. If the switch is set to "On", the corresponding value index is used as an extended controller number.

For details about variables see ADD_VARS.

Parameters:

- Var/Ctrl = Controller A switch (OFF = Variable, ON = extended controller)
A = Variable / Controller A
Var/Ctrl = Controller B switch (OFF = Variable, ON = extended controller)
B = Variable / Controller B
Var/Ctrl = Controller C switch (OFF = Variable, ON = extended controller)
C = Variable / Controller C

ADDI_VARS

Add immediate value

Type: Special

\$64	ADDI_VARS	Ctrl Var/Ctrl	A= A	Ctrl Var/Ctrl	B+ B	Imm Immediate	
------	-----------	------------------	---------	------------------	---------	------------------	--

Description:

The immediate value is added to the value of variable B and the result is stored in variable A. The Var/Ctrl switches select whether a variable or a controller should be accessed. If the switch is set to "On", the corresponding value index is used as an extended controller number.

For details about variables see ADD_VARS.

Parameters:

- Var/Ctrl = Controller A switch (OFF = Variable, ON = extended controller)
A = Variable / Controller A
Var/Ctrl = Controller B switch (OFF = Variable, ON = extended controller)
B = Variable / Controller B
Immediate = Immediate value to add to B

SET_VAR

Set variable to immediate value

Type: Special

\$65	SET_VAR	Ctrl	A=	Imm			
		Var/Ctrl	A		Immediate		

Description:

The current value of variable A is replaced with the specified immediate value. The Var/Ctrl switch selects whether a variable or a controller should be accessed. If the switch is set to "On", the value index is used as an extended controller number.

For details about variables see ADD_VARS.

Parameters:

- Var/Ctrl = Controller A switch (OFF = Variable, ON = extended controller)
A = Variable / Controller A
Immediate = 16 bit value (0-65535) to be stored in A

Setup Macros

POR TAMENTO

Sets portamento mode and time

Type: *Setup*

	Port. State	Port. Type			ms switch	Ticks/Millisec.
\$1B PORTAMENT.	<i>state</i>	<i>type</i>			<i>ms flag</i>	<i>ticks/ms</i>

Description:

Setup the mode for portamento operation. If state equals zero, portamento will be disabled, while 1 enables it. A value of 2 keeps the state unchanged. By default, portamento is controlled by the MIDI portamento controller (65). The time parameter controls the time needed to reach the current key. By default, this is 500ms. Type defines the type of portamento function desired. Zero performs portamento only if the last key is still depressed, while 1 performs it all the time. A legato function can be achieved by setting the time parameter to zero.

Parameters:

- mode state = 0=off, 1=on, 2=midi controller 65 controls off/on
- type = 0=enable portamento when the last key is held, and while the new key is pressed. 1=portamento is always active.
- ms flag = This switches the following parameter from ticks to milliseconds
- ticks/ms = Specifies portamento time in either ticks or milliseconds

PITCHWHEELR

Sets the number of keysteps used to calculate the pitchwheel range

Type: Setup

	Range up	Range down							
\$33	PITCHWHEELR	<i>rng up</i>	<i>rng dwn</i>						

Description:

Sets the number of keysteps used to calculate the pitchwheel range. The size of the lower and the upper range can be selected separately. The default value is two keysteps for both ranges.

Parameters:

rng up = The positive range in key steps

rng dwn = The negative range in key steps

VOL_SELECT

Adds a volume calculation component

Type: Setup

	MIDI Contr.	Scale Percentage	Combine uCode				
\$40	VOL_SELECT	<i>ctrl</i>	<i>scale</i>	<i>comb</i>			

Description:

Selects an additional component for the volume calculation. The first use of this command resets the default values to empty and adds the new component. All further calls add additional components. A *comb* value of 0 sets the value, 1 adds the value and 2 multiplies the old and the new values.

Parameters:

- | | | |
|--------------|---|--|
| <i>ctrl</i> | = | The MIDI controller number |
| <i>scale</i> | = | A signed scaling factor (-10000 to 10000) |
| <i>ctrl</i> | = | 0 = set value from defined controller, 1 = add value to current controller assignment, 2 = multiply the old and the new controller |

You may use all standard MIDI controller numbers. In addition, the following extensions have been defined:

ExCTRL	Function
128	<i>Pitchbend (since there is no controller in the MIDI standard)</i>
129	<i>Aftertouch (since there is no controller in the MIDI standard)</i>
130	<i>LFO1 (see SETUP_LFO macro command)</i>
131	<i>LFO2 (see SETUP_LFO macro command)</i>
132	<i>Surround Panning (0= front, 127=surround)</i>

PAN_SELECT

Adds a panning calculation component

Type: Setup

	MIDI Contr.	Scale Percentage	Combine uCode				
\$41 PAN_SELECT	<i>ctrl</i>	<i>Scale</i>	<i>comb</i>				

Description:

Selects an additional component for the panning calculation. The first use of this command resets the default values to empty and adds the new component. All further calls add additional components. A *comb* value of 0 sets the value, 1 adds the value and 2 multiplies the old and the new values.

Parameters:

- | | | |
|--------------|---|--|
| <i>ctrl</i> | = | The MIDI controller number |
| <i>scale</i> | = | A signed scaling factor (-10000 to 10000) |
| <i>ctrl</i> | = | 0 = set value from defined controller, 1 = add value to current controller assignment, 2 = multiply the old and the new controller |

You may use all standard MIDI controller numbers. In addition, the following extensions have been defined:

ExCTRL	Function
128	<i>Pitchbend (since there is no controller in the MIDI standard)</i>
129	<i>Aftertouch (since there is no controller in the MIDI standard)</i>
130	<i>LFO1 (see SETUP_LFO macro command)</i>
131	<i>LFO2 (see SETUP_LFO macro command)</i>
132	<i>Surround Panning (0= front, 127=surround)</i>

PitchW_SELECT

Adds a pitchwheel calculation component

Type: Setup

	MIDI Contr.	Scale Percentage	Combine uCode				
\$42	PitchW_SELECT	ctrl	Scale	comb			

Description:

Selects an additional component for the pitchwheel calculation. The first use of this command resets the default values to empty and adds the new component. All further calls add additional components. A *comb* value of 0 sets the value, 1 adds the value and 2 multiplies the old and the new values.

Parameters:

- | | | |
|-------|---|--|
| ctrl | = | The MIDI controller number |
| scale | = | A signed scaling factor (-10000 to 10000) |
| ctrl | = | 0 = set value from defined controller, 1 = add value to current controller assignment, 2 = multiply the old and the new controller |

You may use all standard MIDI controller numbers. In addition, the following extensions have been defined:

ExCTRL	Function
128	<i>Pitchbend (since there is no controller in the MIDI standard)</i>
129	<i>Aftertouch (since there is no controller in the MIDI standard)</i>
130	<i>LFO1 (see SETUP_LFO macro command)</i>
131	<i>LFO2 (see SETUP_LFO macro command)</i>
132	<i>Surround Panning (0= front, 127=surround)</i>

ModW_SELECT

Adds a modulation wheel calculation component

Type: Setup

		MIDI Contr.	Scale Percentage	Combine uCode			
\$43	ModW_SELECT	ctrl	Scale	comb			

Description:

Selects an additional component for the modulation wheel calculation. The first use of this command resets the default values to empty and adds the new component. All further calls add additional components. A *comb* value of 0 sets the value, 1 adds the value and 2 multiplies the old and the new values.

Parameters:

- | | | |
|-------|---|--|
| ctrl | = | The MIDI controller number |
| scale | = | A signed scaling factor (-10000 to 10000) |
| ctrl | = | 0 = set value from defined controller, 1 = add value to current controller assignment, 2 = multiply the old and the new controller |

You may use all standard MIDI controller numbers. In addition, the following extensions have been defined:

ExCTRL	Function
128	<i>Pitchbend (since there is no controller in the MIDI standard)</i>
129	<i>Aftertouch (since there is no controller in the MIDI standard)</i>
130	<i>LFO1 (see SETUP_LFO macro command)</i>
131	<i>LFO2 (see SETUP_LFO macro command)</i>
132	<i>Surround Panning (0= front, 127=surround)</i>

PEDAL_SELECT

Adds a pedal calculation component

Type: Setup

		MIDI Contr.	Scale Percentage	Combine uCode			
\$44	PEDAL_SELECT	ctrl	Scale	comb			

Description:

Selects an additional component for the pedal calculation. The first use of this command resets the default values to empty and adds the new component. All further calls add additional components. A *comb* value of 0 sets the value, 1 adds the value and 2 multiplies the old and the new values.

Parameters:

- | | | |
|-------|---|--|
| ctrl | = | The MIDI controller number |
| scale | = | A signed scaling factor (-10000 to 10000) |
| ctrl | = | 0 = set value from defined controller, 1 = add value to current controller assignment, 2 = multiply the old and the new controller |

You may use all standard MIDI controller numbers. In addition, the following extensions have been defined:

ExCTRL	Function
128	<i>Pitchbend (since there is no controller in the MIDI standard)</i>
129	<i>Aftertouch (since there is no controller in the MIDI standard)</i>
130	<i>LFO1 (see SETUP_LFO macro command)</i>
131	<i>LFO2 (see SETUP_LFO macro command)</i>
132	<i>Surround Panning (0= front, 127=surround)</i>

PORTA_SELECT

Adds a portamento calculation component

Type: Setup

	MIDI Contr.	Scale Percentage	Combine uCode			
\$45 PORTA_SELECT	<i>ctrl</i>	<i>Scale</i>	<i>comb</i>			

Description:

Selects an additional component for the portamento calculation. The first use of this command resets the default values to empty and adds the new component. All further calls add additional components. A *comb* value of 0 sets the value, 1 adds the value and 2 multiplies the old and the new values.

Parameters:

- | | | |
|--------------|---|--|
| <i>ctrl</i> | = | The MIDI controller number |
| <i>scale</i> | = | A signed scaling factor (-10000 to 10000) |
| <i>ctrl</i> | = | 0 = set value from defined controller, 1 = add value to current controller assignment, 2 = multiply the old and the new controller |

You may use all standard MIDI controller numbers. In addition, the following extensions have been defined:

ExCTRL	Function
128	<i>Pitchbend (since there is no controller in the MIDI standard)</i>
129	<i>Aftertouch (since there is no controller in the MIDI standard)</i>
130	<i>LFO1 (see SETUP_LFO macro command)</i>
131	<i>LFO2 (see SETUP_LFO macro command)</i>
132	<i>Surround Panning (0= front, 127=surround)</i>

REVERB_SELECT

Adds a reverb calculation component

Type: Setup

	MIDI Contr.	Scale Percentage	Combine uCode				
\$46	REVERB_SELECT	<i>ctrl</i>	<i>Scale</i>	<i>comb</i>			

Description:

Selects an additional component for the reverb calculation. The first use of this command resets the default values to empty and adds the new component. All further calls add additional components. A *comb* value of 0 sets the value, 1 adds the value and 2 multiplies the old and the new values.

Parameters:

- ctrl* = The MIDI controller number
- scale* = A signed scaling factor (-10000 to 10000)
- ctrl* = 0 = set value from defined controller, 1 = add value to current controller assignment, 2 = multiply the old and the new controller

You may use all standard MIDI controller numbers. In addition, the following extensions have been defined:

ExCTRL	Function
128	<i>Pitchbend (since there is no controller in the MIDI standard)</i>
129	<i>Aftertouch (since there is no controller in the MIDI standard)</i>
130	<i>LFO1 (see SETUP_LFO macro command)</i>
131	<i>LFO2 (see SETUP_LFO macro command)</i>
132	<i>Surround Panning (0= front, 127=surround)</i>

SPAN_SEL

Adds a surround panning calculation component

Type: Setup

		MIDI Contr.	Scale Percentage	Combine Mode			
\$47	SPAN_SEL	<i>ctrl</i>	<i>Scale</i>	<i>comb</i>			

Description:

Selects an additional component for the surround panning calculation. The first use of this command resets the default values to empty and adds the new component. All further calls add additional components. A *comb* value of 0 sets the value, 1 adds the value and 2 multiplies the old and the new values.

Parameters:

- | | | |
|-------|---|--|
| ctrl | = | The MIDI controller number |
| scale | = | A signed scaling factor (-10000 to 10000) |
| ctrl | = | 0 = set value from defined controller, 1 = add value to current controller assignment, 2 = multiply the old and the new controller |

You may use all standard MIDI controller numbers. In addition, the following extensions have been defined:

ExCTRL	Function
128	<i>Pitchbend (since there is no controller in the MIDI standard)</i>
129	<i>Aftertouch (since there is no controller in the MIDI standard)</i>
130	<i>LFO1 (see SETUP_LFO macro command)</i>
131	<i>LFO2 (see SETUP_LFO macro command)</i>
132	<i>Surround Panning (0= front, 127=surround)</i>

DOPPLER_SEL

Adds a reverb calculation component

Type: Setup

	MIDI Contr.	Scale Percentage	Combine Mode			
\$48	DOPPLER_SEL	ctrl	Scale	comb		

Description:

Selects an additional component for the doppler calculation. The first use of this command resets the default values to empty, and adds the new component. All further calls add additional components. A *comb* value of 0 sets the value, 1 adds the value and 2 multiplies the old and the new values.

Parameters:

- ctrl = The MIDI controller number
- scale = A signed scaling factor (-10000 to 10000)
- ctrl = 0 = set value from defined controller, 1 = add value to current controller assignment, 2 = multiply the old and the new controller

You may use all standard MIDI controller numbers. In addition, the following extensions have been defined:

ExCTRL	Function
128	Pitchbend (since there is no controller in the MIDI standard)
129	Aftertouch (since there is no controller in the MIDI standard)
130	LFO1 (see SETUP_LFO macro command)
131	LFO2 (see SETUP_LFO macro command)
132	Surround Panning (0= front, 127=surround)

SETUP_LFO

Sets up LFO characteristics

Type: Setup

	LFO Nr.	Period in ms					
\$50	SETUP_LFO	<i>num</i>	<i>period</i>				

Description:

Enables or disables the specified LFO. A value of zero as *period* will disable the LFO. From that point on it will continue to produce the last value as a constant. The LFO always starts at center level, unless it already was active. *Num* specifies which LFO should be setup. Currently two LFOs (0,1) are supported.

The LFOs always swing through the full amplitude. Use the scale parameter when combining them, using the **x_SELECT** macro commands to control the amplitude.

Parameters:

num = The LFO number to be set (0 or 1)

period = The period time of one LFO cycle in milliseconds

The following table shows how the two LFOs can be accessed in combination with any of the **x_SELECT** macro commands

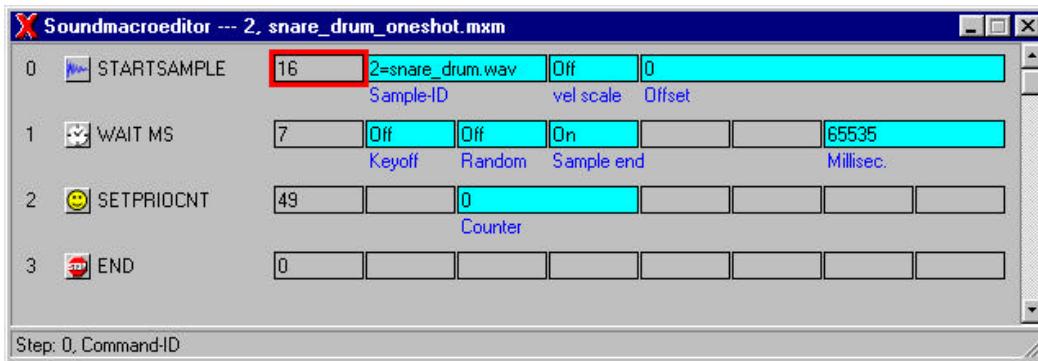
ExCTRL	Function
130	<i>LFO1</i>
131	<i>LFO2</i>

Appendix 1.2 - N64 Macro Templates

This part contains descriptions of the Macro templates used by **MusyX** for the Nintendo64. The descriptions for the templates are based upon the content of the first template. So, to understand the various aspects of programming in SMaL, you should read the descriptions in order.

?_ONESHOT

Template for starting a simple oneshot sample



Description:

This short SoundMacro is an example of how to get a simple oneshot sample working. It may also work even with just 2 commands: STARTSAMPLE and END. But one has to understand the underlying priority and voice allocation scheme in order to build a correct SoundMacro.

The dynamic-voice-allocation of **MusyX** is a very important part of the system and understanding how it works help to produce better results. A voice in **MusyX** uses a priority from 0-255 and the so-called age counter (0-65535) to determine their importance. Please refer to “Dynamic Voice Allocation”, on page 14 of the **MusyX** Manual, for a detailed description.

Step by Step:

Assume that this SoundMacro is properly set to the desired midichannel and ready to play. If the slave now receives a midi-on command (a note is played from a connected midikeyboard or a midi-sequencer), the slave searches for a (free) voice and execution of the SoundMacro starts at step 0.

However, before the first command is executed, there are some basic settings made by the init-routine. The frequency of the voice will be set to the note received by midi. The volume of the voice will be calculated using the midi-velocity and the channel volume. The priority will be set as defined in the Soundlist and the age-counter will be set to 60000 (decimal).

Step 0:

The first command to be executed is STARTSAMPLE, which is of course used to start the sample. The first parameter of this command is the Sample-ID referencing our desired sample (in this case a snare drum). The “Vel scale” and “offset” offer a nice option to start the sample with an offset (measured in samples), which can be scaled to the beginning of the sample using the velocity. This can provide instruments of different attack styles depending on the velocity and helps to give some sounds a more natural feel. If the offset is 0, this option is disabled. Since STARTSAMPLE has no wait parameter, the execution continues ‘virtually’ in real-time.

Step 1:

The second command (WAIT) is a part of the priority and voice allocation scheme mentioned earlier. The “Millisec.” parameter regularly defines in milliseconds how long the execution of the macro is to be stopped. If set to the highest possible value 65535 (hex \$FFFF), the sample will play continuously.

However, the execution of the SoundMacro will continue when one of the condition parameters (keyoff, random or sample-end) becomes ‘true’. In this case the command waits until the sample is finished. This is very important, because of the two following commands.

Step 2:

The next command (SETAGECOUNT) is used to set or reset the age-counter of the voice. Since the sample has played to the end and there are no other things to do in this SoundMacro, we can reset the age counter to 0 (oldest). Thus, a new midi-note can use this voice as soon as possible. This command also proceeds directly to the next and last command in our SoundMacro.

Step 3:

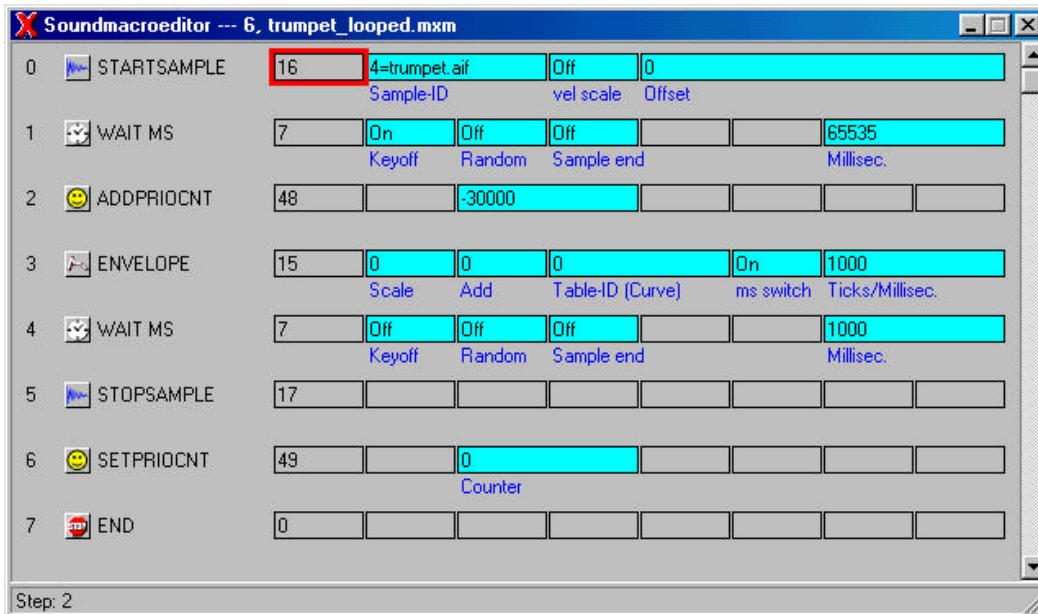
The END command is the last command in all SoundMacros and also resets the priority of the voice to 0. In this case, the voice is now totally free and can be used by any new midi-note, even one with a lower priority.

Tip:

Reset the age counter at the end of every SoundMacro and keep an eye on the voice bars appearing in the slave window.

?_LOOPED

Template for playing a looped sample



Description:

This is an example for a looped instrument. (The following explanation of the steps is based on the description of the first macro-template “?_ONESHOT” which you should have already read.)

Step 0:

The first command is, again, a STARTSAMPLE command. Since we have a looped sample (trumpet.aif) here it would play forever, so we have to take care of what to do after the midi-key is released.

Step 1:

The WAIT_MS command this time is set to wait until the midikey is released. After the keyup is received, we want to stop the sample. But to get a more musical result, we decide to fade out smoothly before we stop the sample.

Step 2:

During the fade out we also want to give the voice more “age”, because it is not so important compared to voices that are still held. To do this we use the ADDAGECOUNT command, but with a negative value (-30000) because the lower the counter the older is the voice.

Step 3:

This command (Envelope) fades the volume of the voice to a new one, calculated from the midi-velocity using the “scale” and “add” parameter. Since both values are 0, the voice will be faded to 0 in the time given by the milliseconds parameter (1000ms or one second).

Step 4:

Here we have, again, a WAIT, but this time no condition is given and the milliseconds parameter is set to the same time like in the envelope command, before. So we can make sure that the next command will be executed just when the envelope fade is completed.

Step 5:

The volume is now 0 and the sample cannot be heard anymore. However the sample is still playing, so we should switch it off if we want to end this soundmacro. It is also recommended to do this because the mixing routine will be unburdened.

Step 6:

The same procedure as in our last macro. We set the age to 0 so that the voice is free.

Step 7:

End our macro.

APPENDIX 2 – Game Boy Musicians Reference

Table of Contents:

An Introduction to Sound on the Nintendo Game Boy	179
Appendix 2.1 – Game Boy Slave	183
What is this Slave?	183
Starting the Slave	184
Working with the Slave	187
Testing your Project with the Slave	191
Exiting the Slave	192
Appendix 2.2 – SlaveROMGenerator	193
Creating a Slave ROM	195
Appendix 2.3 – Game Boy Macro Commands	197
END	198
Structure Macros	199
STOP	199
SPLITKEY	200
SPLITVEL	201
LOOP	202
GOTO	203
WAIT	204
PLAYMACRO	205
KEYOFF	206
SPLITRND	207
TRAP_KEYOFF	208
UNTRAP_KEYOFF	209

Voice/Sample Macros	210
PLAYKEYSAMPLE	210
SETVOICE	211
STARTSAMPLE	212
VOICE_OFF	213
VOICE_ON	214
SETNOISE	215
PWM_START	216
PWM_UPDATE	217
PWM_FIXED	218
PWM_VELOCITY	219
WAVE_ON.....	220
 Volume/Pan Macros	 221
SETADSR.....	221
SETVOLUME.....	222
PANNING	223
ENVELOPE	224
HARDENVELOPE	225
 Pitch Macros	 226
RESET_MOD	226
STOP_MOD.....	227
PORTLAST.....	228
RNDNOTE.....	229
ADDNOTE	230
SETNOTE	231
LASTNOTE	232
PORTAMENTO	233
VIBRATO.....	234
PITCHSWEEP	235
 Special Macros.....	 236
SENDFLAG	236
SAMPLEMAP	237
CURRENTVOL.....	238
ADD_SET_PRIO.....	239
 Appendix 2.4 – Performance Issues	 241
 Appendix 2.5 - Troubleshooting Guide.....	 243

An Introduction to Sound on the Nintendo Game Boy

Since the Game Boy hardware is already ten years old, it can be considered rather old fashioned by today's standards. Nevertheless it is possible to create surprisingly good sounds, if the sound artist is fully aware of the possibilities and the limitations of the Game Boy sound hardware.

This appendix is meant to introduce you to what the Game Boy has to offer (and what not to expect).

First, there is need to mention that the Game Boy and Game Boy Color sound hardware are identical. No improvements have been made to the Game Boy Color with respect to sound.

Having stated this, we will refer to any kind of Game Boy hardware (including the traditional Game Boy, the Game Boy pocket and the Game Boy Color) as Game Boy.

*Note: **MusyX** for Game Boy does not provide support for the extended sound capabilities offered by Super Game Boy hardware.*

Game Boy sound hardware is comprised of 4 individual sound generators of three different kinds. Two of those four generators are identical in function with the exception of one special feature on one of them.

The Game Boy programming manual calls the sound generators Sound1, Sound2, Sound3 and Sound4. To prevent confusion with the general understanding of what a sound is, we will continue to call them Voice1 through Voice4.

The three different kinds of sound generators are as follows:
Voice1 and Voice2 use rectangular wave patterns to create a sound.
Voice3 uses a 32 4-bit sample long wave pattern to create sound.
Voice4 uses a polynomial counter to create random noise.

All four voices have a volume control that can, with the exception of Voice3, be set in 16 steps from mute (0) to maximum (15). Voice3 does not have the full range of volume control. It can be set only in 4 steps from mute (0) to 25%, 50% and 100% (3). There is also an envelope available in hardware for all but Voice3. **MusyX** however offers a more flexible envelope for all 4 voices in software.

All voices, with the exception of Voice4, have a frequency setting that allows for sounds to be produced from 64Hz to 131kHz. Since Voice4 is a noise generator, it does not have such a setting. It can produce different kinds of frequencies by modifying the parameters of the polynomial clock.

For Voice1 and Voice2 you may choose from hardwired pulse widths for the rectangular wave pattern. It offers pulse widths of 12.5%, 25%, 50% and 75%. The latter is the equivalent of the 25% pulse width, but it is phase inverted.

Voice1 also offers a very limited pitchsweep function, which is NOT supported by **MusyX**. Instead we offer 2 sophisticated pitchsweeps for each Voice1 through Voice3 in software.

Finally, each voice can be assigned to the left, right or both sound outputs individually. This selection allows for a wide stereo spectrum only (the sound is played back entirely on the left, right or on both). Smooth panning changes are not possible.

The Game Boy sound hardware has just two flaws that are important to know and that unfortunately cannot be circumvented.

Whenever a voice is started or stopped a clicking sound is produced. The intensity of it varies depending on the signal being output at the moment of the start or stop.

Whenever a change in volume on a playing voice is requested, the voice needs to be restarted. This usually creates a slight clicking sound as well.

MusyX extends on the capabilities of the Game Boy and adds a few new features like extended sample playback and pulse-width modulation.

Extended features fall into the categories of pitch and volume changes. To modify pitch for instance, **MusyX** adds features like vibrato, portamento, pitch sweeps, fixed pitches and random pitches for Voice1, Voice2 and Voice3.

Enhancements for volume controls are ADSR curves, envelopes, fade-ins and fixed volumes for all 4 voices.

New features are:

- Velocity dependent selection of the rectangular wave pattern for Voice1 and Voice2.
- Pulse-width modulation of a rectangular wave pattern for Voice3
- Velocity dependent creation of a rectangular wave pattern for Voice3
- Playback of samples in two quality settings (normal and low) for Voice3 as music instruments or sound effects
- Playback of samples in high quality using full CPU performance for introductory voice or music.

In addition, the SMaL programming language offers control of the sound while it is playing in numerous ways with control commands like: wait, loop, goto, trap, split and more.

To experience the possibilities of *MusyX*, listen to the provided example and see how each sound is achieved by examining its corresponding SMaL macro.

Appendix 2.1 – Game Boy Slave

What is this Slave?

The slave program running on the PC communicates with the [MusyX](#) editor and the Game Boy slave running on Game Boy Color. It also receives MIDI data from a connected MIDI keyboard or the virtual MIDI keyboard in the [MusyX](#) editor.

The slave is responsible for relaying all the data it receives to Game Boy, which will in turn produce sound.

To do this you will need to connect a standard Game Boy Color to the PC running the slave program, via the supplied customized link cable. This cable plugs into the link port of Game Boy Color and any available printer port of the PC.

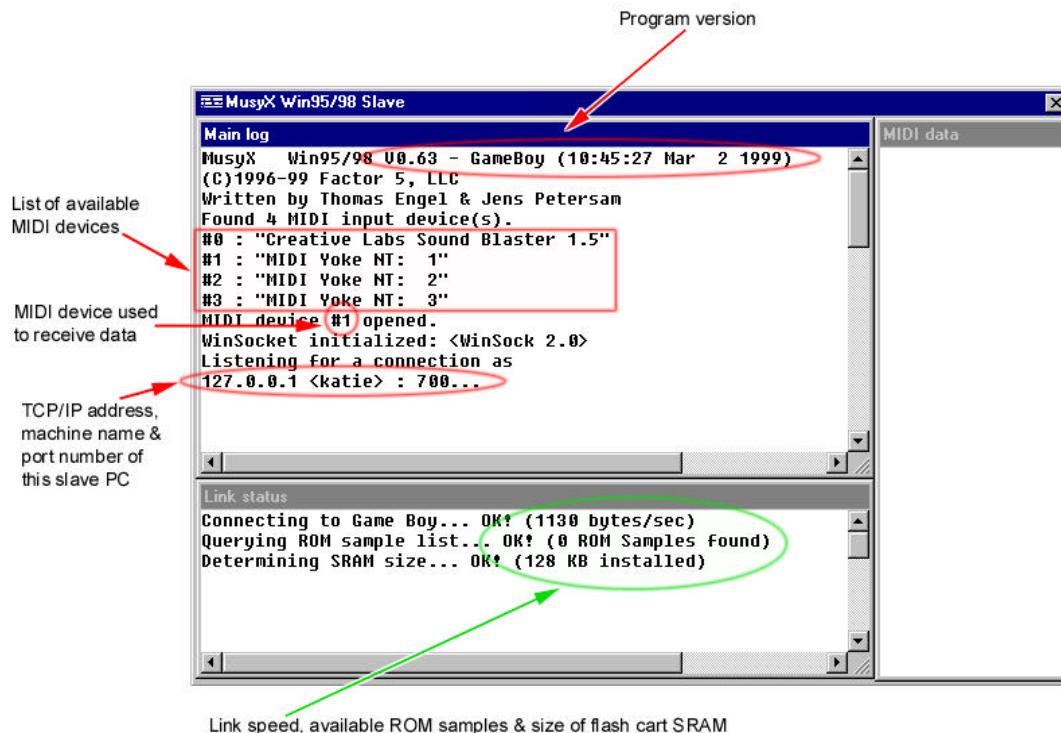
The slave receives data from the [MusyX](#) editor using a network connection known as TCP/IP. This virtually enables any machine in your Local Area Network or even any machine on the globe, to serve as your sound slave. If you are planning to use the same machine to run both the slave and the editor, you still need to have TCP/IP installed and have both the editor and the slave refer to the same IP address.

The slave receives MIDI data through a Windows MIDI device, usually a sound card with a built-in MIDI port. If you plan on using the same machine for the slave and your favorite sequencer program, you will need to connect a loopback plug to your soundcard's midiport, or install a so-called loopback device driver, which serves as a MIDI device to Windows. More on this later.

Starting the Slave

The slave program is a Windows GUI executable, which has been installed on your system along with the **MusyX** editor.

When you start the program, you will see a window similar to this:



Note that the main window is divided into three sections: the main log, the Link status and the MIDI data.

The Main Log

This window holds all the information about the data received from the **MusyX** editor while you are working. Right after the start of the program, it will display system information, as seen above.

Program Version:

This displays the type and version of the slave you are currently running. Your version and build numbers are likely to be different from the example above.

Available MIDI Devices:

These lines enumerate all MIDI devices that can be used for input. The number of devices depends on your Windows installation and the MIDI devices you have installed on your system.

Used MIDI Device:

This line tells you which MIDI device number, from the presented list of available devices, you are now using for input. This should be the setting you made in the slave configuration program. If you need to use a different input device, you must reconfigure the slave with the configuration program.

TCP/IP Network Info

This tells you the local IP and port address the slave program uses to communicate with the *MusyX* editor. It also shows you the name of the machine on which you are running the slave.

The IP and port numbers are the ones that you need to enter in the editors network options to have it communicate with this slave.

The Link Status

This window contains information about the current data link between the slave PC and the connected Game Boy Color. Initially it will display system information, as seen on the previous page.

Link Speed:

If the communication to the Game Boy Color can be established successfully, you can see here how fast the communication is taking place. This speed can drop, if during the course of transmitting data errors cause packets to be retried. Should the speed drop to below 70% of the initial value shown here and the slave is idle for more than 30 seconds, it will try to reconnect to the Game Boy Color at a higher speed.

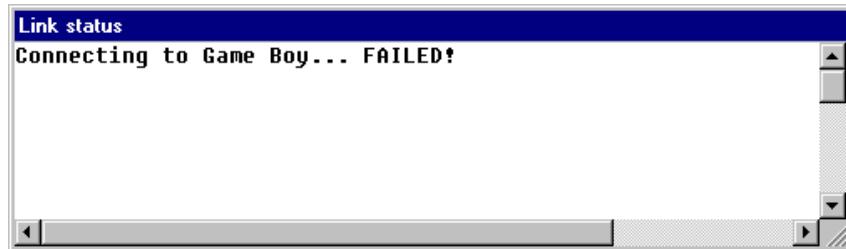
Number of ROM Samples:

Here, the number of samples that are registered in the Game Boy Color slave ROM is displayed. ROM samples become important when the internal RAM of Game Boy Color cannot hold all the samples your project requires. Please refer to "Data Conversion Tools" in the Programmers Reference, for more details about ROM samples.

SRAM Size:

This states how much SRAM is installed in the flash ROM that holds the Game Boy Color slave program. This number can be anything between 0K and 128K in 16K increments.

If you see the following error message instead of a successful connection message...



... then the slave was not able to establish a link with Game Boy Color. In this case, please refer to the troubleshooting section at the end of this appendix.

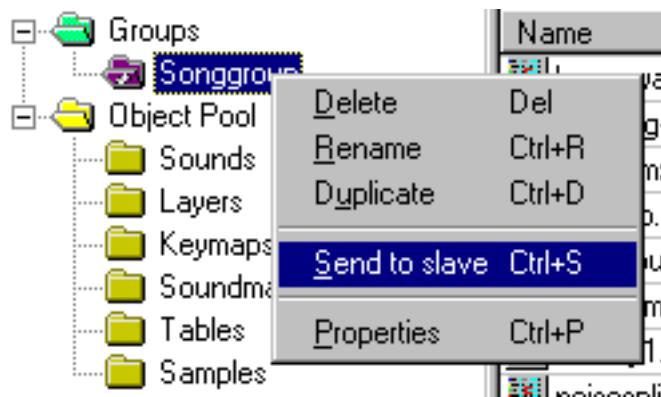
The MIDI Data Window

This window shows all incoming MIDI data from the MIDI device in their form, as 3 byte data packets.

Working with the Slave

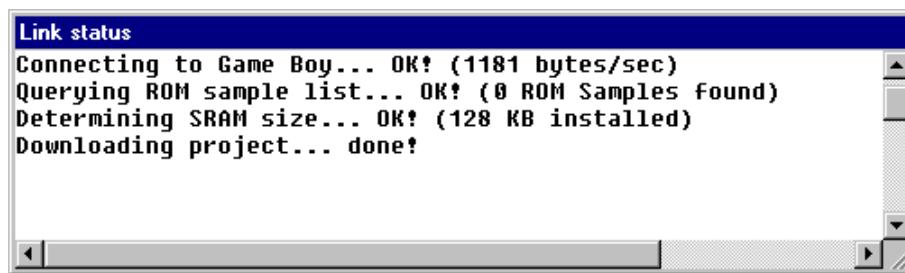
When you create your sounds, the moment will come when you want to test them on Game Boy. Before you can do this, you need to send your sound project to the slave program, which will process it and in turn transfer the data to the Game Boy itself.

You initiate this process by highlighting the sound group (song group or sound effect group) you wish to test and choose "Send to slave" from the context sensitive pop-up menu:



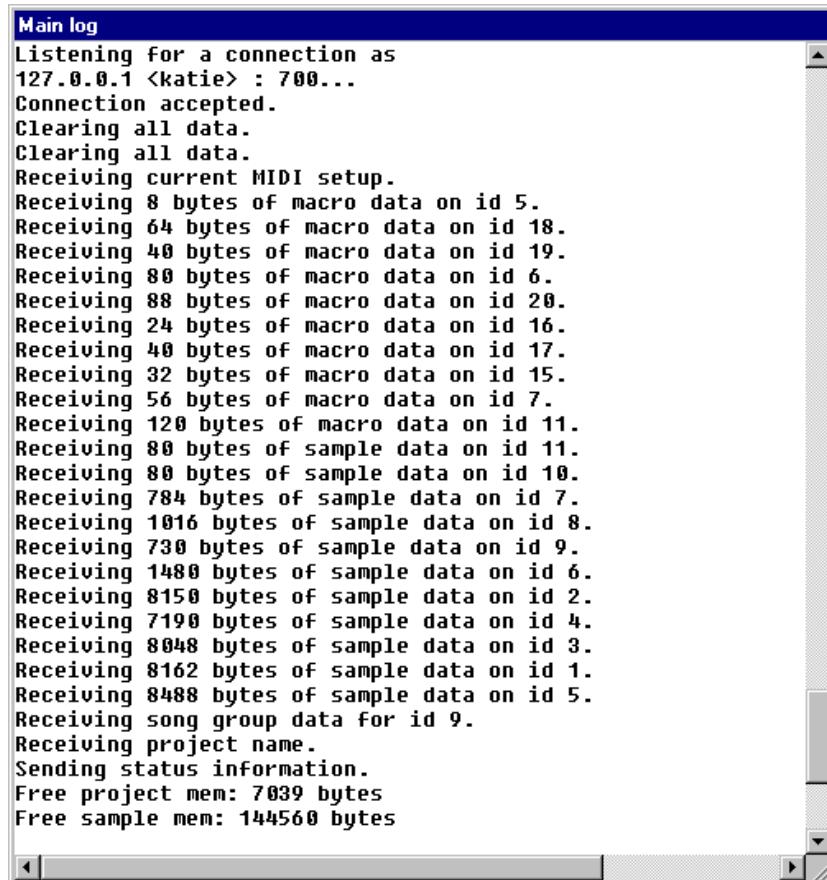
The slave program will now receive all of this group's data and print it out in the main log.

As soon as the slave program has received all data from the group, it will download it to Game Boy Color. This may take some time, depending on the established communication speed and the size of your project.



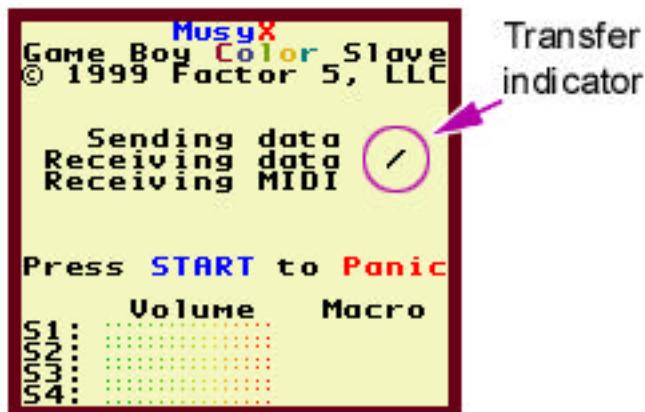
You will see the line "Downloading project..." during the download phase, at the end of which a "done!" is appended if everything worked fine.

Here is an example of what it looks like when you transfer the supplied demo to the slave.



```
Main log
Listening for a connection as
127.0.0.1 <katie> : 700...
Connection accepted.
Clearing all data.
Clearing all data.
Receiving current MIDI setup.
Receiving 8 bytes of macro data on id 5.
Receiving 64 bytes of macro data on id 18.
Receiving 40 bytes of macro data on id 19.
Receiving 80 bytes of macro data on id 6.
Receiving 88 bytes of macro data on id 20.
Receiving 24 bytes of macro data on id 16.
Receiving 40 bytes of macro data on id 17.
Receiving 32 bytes of macro data on id 15.
Receiving 56 bytes of macro data on id 7.
Receiving 120 bytes of macro data on id 11.
Receiving 80 bytes of sample data on id 11.
Receiving 80 bytes of sample data on id 10.
Receiving 784 bytes of sample data on id 7.
Receiving 1016 bytes of sample data on id 8.
Receiving 738 bytes of sample data on id 9.
Receiving 1480 bytes of sample data on id 6.
Receiving 8150 bytes of sample data on id 2.
Receiving 7190 bytes of sample data on id 4.
Receiving 8048 bytes of sample data on id 3.
Receiving 8162 bytes of sample data on id 1.
Receiving 8488 bytes of sample data on id 5.
Receiving song group data for id 9.
Receiving project name.
Sending status information.
Free project mem: 7039 bytes
Free sample mem: 144560 bytes
```

While the slave program is downloading data into Game Boy Color, a rotating transfer indicator on Game Boy signals you that a download is in progress.



The available size of memory will be displayed in two separate units in the main log (see the screenshot on the previous page).

First, we have the available project memory. This is a fixed size chunk of memory, unaffected by any SRAM in the flash ROM, of 7424 bytes. It holds all the sound macros and ADSR tables.

Second, we have an area of variable size, between 24 KB and 152 KB, depending on the amount of SRAM on the flash ROM that holds the Game Boy slave program. This area stores sound samples and is the one most likely to run out first.

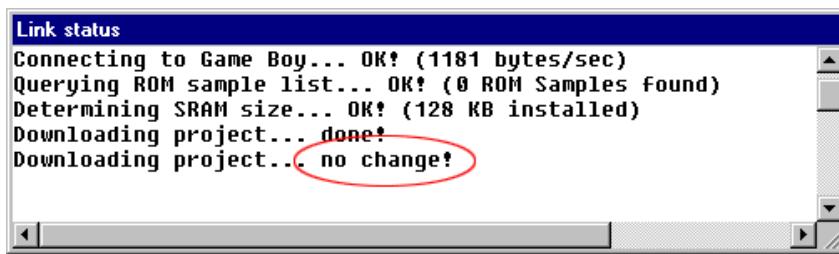
The usage of the sample RAM is also displayed in the *MusyX* editor's Network Master window:



When you are running out of sample RAM, you need to move some or all of your samples into ROM to clear some space. Your project can still use samples that have been moved to ROM. The difference is that they do not consume any RAM space, so you can continue to add more samples here.

Please refer to, "Data Conversion Tools" in the Programmers Reference, for details on how to move samples to ROM.

When you are sending a project to the slave repeatedly, you will see a line displayed in the Link status window like this.



To preserve time, the slave program keeps an internal backup of what it knows is already located in Game Boy. If it determines that the data is accurate it will not download anything to Game Boy.

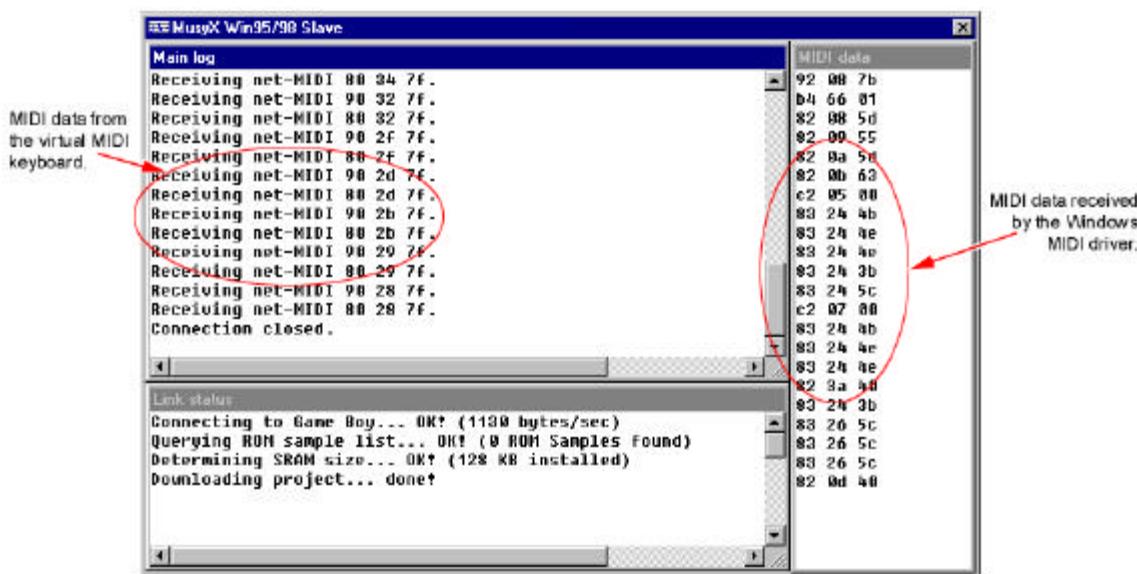
You are most likely to encounter this when you are actively working on a macro while trying out your changes. Some parameters of a macro command have a limited resolution in Game Boy, so sometimes a change you make will actually have no effect.

The WAIT command is a good example. Its resolution is limited to 16.67 milliseconds inside Game Boy, although you can enter any number you like. Because of this, values like 4, 10 and 13 milliseconds will be rounded up to 16.67 and, even though you made a change to the macro, it does not make any difference to Game Boy. Since there is no change to download, you will be prompted with the above line, "no change!".

Testing your Project with the Slave

Once you have successfully downloaded a valid project into Game Boy, you can use your MIDI keyboard, sequencer program or the *MusyX* virtual MIDI keyboard to test your sounds.

Whenever the slave program receives either kind of MIDI data, it displays them in either the MIDI data window or the Main log, depending on the type of MIDI data (data from the virtual keyboard or from the sequencer).



As you feed MIDI data to the slave program it sends it to Game Boy, where it will be processed and a sound is created.

When Game Boy receives MIDI data, the transfer indicator will rotate next to the line "Receiving MIDI" on the Game Boy display.

Since MIDI data is very short compared to project data you download into the Game Boy, the transfer indicator will stay on screen longer for MIDI data than it does for project data, to give you a better visual feedback.

NOTE:

The slave program can only transfer one kind of data at a time. While project data is being downloaded, all incoming MIDI data is discarded. Downloading project data also takes precedence over MIDI data so, whenever you modify your project while your sequencer is sending MIDI to the slave, sound output will stop for the time of the project download. This is normal.

Exiting the Slave

To close the slave program, click on the Windows X symbol in the upper right hand corner of the programs window.

When you still have the **MusyX** editor running while closing the slave program, a window will pop up telling you that the slave program is no longer running.



This can also occur when the link between Game Boy Color and the PC is severed. Possible causes are removing the link cable or switching off Game Boy.

Another occurrence for this is bad communication between the PC and Game Boy. If too many errors occur in a short period of time and they can not be error-corrected, the slave program terminates itself.

Errors usually occur due to line noise caused by radio interference in the link cable or by weak batteries in Game Boy.

We recommend using an AC adapter rather than batteries to operate Game Boy Color, since the batteries are quickly exhausted when flash ROM is used (according to Nintendo, use of the new flash ROM with built-in Rumble Pak can cause batteries to be drained in as little as 1 hour). Using the serial link also increases the required current.

Appendix 2.2 – SlaveROMGenerator

While creating music and sound effects for an application, there is always the possibility that Game Boy, which is used to preview the work, will run out of free memory.

Usually this happens because the musician makes extensive use of samples.

The other possibility is that the entire sound project data exceeds the available project memory, which is only 7.25KB in size.

Should the latter occur, the only solution is for the musician to try and optimize his project, since the project cannot exceed this size restriction.

The former problem can be remedied by "swapping out" the samples into ROM, which can hold many more times the samples than the internal RAM of a Game Boy Color, or by using a flash ROM with added built-in RAM. Flash ROM can provide up to an additional 128 KBytes of memory. The total of 152 Kbytes (128 extension together with the built-in Game Boy Color RAM) is now available for samples.

Although using a flash ROM with RAM seems to be the most convenient way to work, moving samples into ROM offers the benefit of not needing to download them into Game Boy via the serial link. This can save you a lot of time.

Moving samples into ROM is easy, too. We are providing a Windows GUI driven tool that allows the musician to take his project, as it is, and put all samples therein into ROM.

After a couple of mouse clicks a new Game Boy slave ROM is created that contains the samples and only needs to be flashed on the existing flash ROM.

Once installed in the Game Boy, the PC slave program will determine if a sample it received is already stored in Game Boy ROM. If it is, the newly received one will not be transferred. This identification process takes the length of the sample and a 32-bit CRC checksum into consideration. Thus the possibility that any new sample added to the project is accidentally believed to be in ROM (and therefore would not be transferred) is minimized.

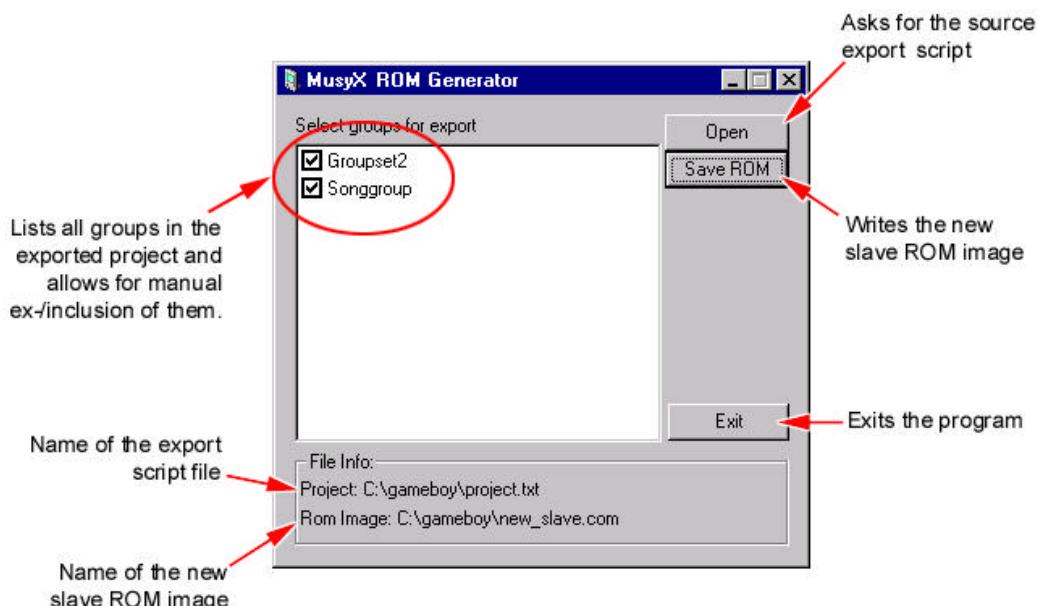
Sometimes a single sample exceeds the above mentioned 152KByte (usually the flash ROM available today have only 32 KByte on them, limiting the size of any one sample to 56 KByte). Locating a sample in ROM is then the only way for the musician to actually listen to the sample

Creating a Slave ROM

To create a new slave ROM with the samples currently in the sound project, you will need to create an export file from within the **MusyX** editor. This is the same procedure you would use if you wanted to convert your project for being built into the application.

From the Project-menu choose "Generate scriptfile for export", enter a file name for the exported script and click the 'save' button.

Now you need to start the SlaveROMGenerator tool (which has been installed in the same location as the **MusyX** editor).



This is what it looks like.

Initially, there will be nothing displayed in the large white area and the "Save ROM" button will be disabled.

What you need to do is click on the "Open" button and navigate, in the file dialog box that appears, to the export script you just created from the **MusyX** editor.

If the script file is correct, in the large group area you will be shown a list of all groups defined in the **MusyX** editor. In the above example, the project contains 2 groups.

For your reference the name of the project script is shown at the bottom in the "File Info" box.

All groups are pre-selected to be included in the new ROM image. If you do not want the samples of a particular group to be included, click on the groups corresponding checkbox to disable it.

For the "Save ROM" button to become enabled, at least one group must be selected.

After you have selected the groups you want to include, click on the "Save ROM" button to bring up a dialog box prompting you for a filename for the resulting ROM image.

Note:

- The file extension for a binary ROM image to be written with the Game Boy development system unfortunately is .COM, which is not to be confused with an MSDOS executable, even though the Windows explorer will call it as such.

If for some reason the ROM file could not be successfully created, a popup dialog box will inform you.

The name of the new ROM image will also be shown in the "File Info" box for your reference.

Now you need to remove the flash ROM from your Game Boy. Flash the new ROM you just created on it and put it back in. You're all set.

Note:

- The next time you start the PC slave program, it will tell you how many ROM samples are installed in your flash ROM.
- If you edit a sample that is located in ROM it will be downloaded into RAM again, since it has changed, and the ROM version will not be used. This ensures that you are always listening to the correct sample.

Appendix 2.3 – Game Boy Macro Commands

On the following pages you will find a description of all Macro Commands used by *MusyX* for Game Boy and Game Boy Color.

END

End of the Macro

Type: *Structure*

\$00	END								
------	-----	--	--	--	--	--	--	--	--

Description:

This is always the last macro command. It can not be deleted from the macro. It terminates the macro permanently.

Structure Macros

STOP

Similar to end

Type: *Structure*

\$01	STOP								
------	------	--	--	--	--	--	--	--	--

Description:

This macro command serves the same function as END, but in contrast to END it can be placed anywhere in the macro.

SPLITKEY

Splits the macro flow depending on the midikey

Type: *Structure*

Key Nr.	SoundMacro ID	SoundMacro step		
\$02	SPLITKEY	key	macro	step

Description:

This command is used to conditionally change the flow of execution in the current macro. The macro program will jump to the given macrostep inside the specified macro, if the current key is higher or the same as the key specified in the parameter.

Parameters:

- key = This parameter specifies a key number to compare against. If the key you play is higher or the same as this key, the macro will branch, otherwise it resumes.
- macro = The ID of the macro to branch to
- step = The step number inside the macro to branch to

SPLITVEL

Splits the macro flow depending on the velocity

Type: *Structure*

	Velocity	SoundMacro ID	SoundMacro step		
\$03	SPLITVEL	<i>velocity</i>	<i>macro</i>	<i>step</i>	

Description:

This command is used to conditionally change the flow of execution in the current macro. The macro program will jump to the given macrostep inside the specified macro, if the current velocity is higher or the same as the velocity parameter.

Parameters:

- velocity = Specifies the velocity to compare the current velocity against.
If the current velocity is higher or the same, the macro will branch, otherwise it will resume.
- macro = The ID of the macro to branch to
- step = The step number inside the macro to branch to

LOOP

Loops back to a macrostep

Type: *Structure*

\$05	LOOP				SoundMacro step <i>step</i>	Times <i>times</i>
------	------	--	--	--	--------------------------------	-----------------------

Description:

Loops to the specified location within the current macro n-times.

Parameters:

- step = The step number inside the current macro to loop back to
times = Number of times to loop back (0=infinite)

GOTO

Jumps to another macro

Type: *Structure*

\$06	GOTO		SoundMacro ID <i>macro</i>	SoundMacro step <i>step</i>		
------	------	--	-------------------------------	--------------------------------	--	--

Description:

Performs an unconditional jump to the specified location.

Parameters:

macro = The ID of the macro to jump into

step = The step number inside the target macro to jump to

WAIT

Suspends macro execution for some time

Type: *Structure*

		Keyoff <i>key release</i>	Random Time <i>random</i>				Milliseconds <i>Ms</i>
\$07	WAIT						

Description:

The execution of the current macro will be suspended until the specified time has elapsed or a keyoff occurs.

Parameters:

- key release = If this flag is set to ON, the macro will resume when it receives a keyoff regardless of the specified wait time
- random = If this flag is set, the macro will resume after a random time has elapsed. In this case the ticks/millisec. parameter defines the maximum wait time
- ms = Specifies, in milliseconds, the time to delay macro execution
A value of 65535 will cause the wait to be endless

PLAYMACRO

Start a macro on another voice

Type: *Structure*

Voice Nr.	SoundMacro ID	Don't reset
\$08	PLAYMACRO	<i>Voice</i> <i>macro</i> <i>Rst.flag</i>

Description:

Starts another macro on the specified voice. Can be used to start 2 or more macros at the same time from a single note.

Parameters:

Voice Nr. = Identifies the voice to be associated with the macro to be started.
(0=Voice1,..., 3=Voice4)

Macro ID = The ID of the macro to start

Don't reset = If this flag is set to ON the voice specified will not be reset
This is useful to take over the voice as it is at this point.

KEYOFF

Sends keyoff to voice

Type: *Control*

		Voice Nr.									
\$12		KEYOFF	Voice								

Description:

Sends a keyoff to the specified voice. Specify 255 to send a keyoff to the current voice.

Parameters:

Voice = Specifies the voice to send a keyoff to (0-3). Enter 255 to send a keyoff to this voice.

SPLITRND

Splits the macro flow depending on a random number

Type: *Structure*

RND	SoundMacro ID	SoundMacro step		
\$13 SPLITRND	<i>rnd</i>	<i>macro</i>	<i>step</i>	

Description:

This command is used to conditionally change the flow of execution in the current macro. The macro program will jump to the given macrostep inside the specified macro, if the generated random value is higher or the same as the *rnd* parameter.

Parameters:

- rnd = Value to compare the random number against.
macro = The ID of the macro to branch to
step = The step number inside the specified macro to branch to

TRAP_KEYOFF

Sets a trap on reception of a keyoff

Type: *Structure*

		SoundMacro ID	SoundMacro step		
\$28	TRAP_KEYOFF		<i>macroID</i>	<i>step</i>	

Description:

This command sets a so-called trap for a keyoff. This means that as soon as the macro receives a keyoff by either the MIDI sequencer or the KEYOFF command, the macro will jump to the macrostep where the trap was set. As long as no keyoff is received the macro proceeds its execution in normal fashion.

This command is used to escape an infinite loop or wait.

Parameters:

macro = ID of the macro to jump to as soon as a keyoff is received

step = Step number in the macro to jump into as soon as a keyoff is received

UNTRAP_KEYOFF

Removes a trap set for keyoff

Type: *Structure*

\$29	UNTRAP_KEYOFF							
------	---------------	--	--	--	--	--	--	--

Description:

Remove a previously set TRAP_KEYOFF.

Voice/Sample Macros

PLAYKEYSAMPLE

Starts a sample on key index

Type: Voice/Sample

\$09	PLAYKEYSAMPLE								
------	---------------	--	--	--	--	--	--	--	--

Description:

Starts a sample on voice 3, using the midikey as an index, into an index table called “sample-map”. This index table must be defined with one single macro using the SAMPLEMAP command. This “sample-map” macro must be placed in the Songgroup on the first Drumlist entry. There can be only one index table per project.

See also SAMPLEMAP.

SETVOICE

Selects a voice for this macro

Type: Voice/Sample

\$0B SETVOICE		Voice Nr.	Don't reset	S 1/2 toggle
Voice	Rst.Flag	Toggle flag		

Description:

Selects a new voice channel for the current macro. This command overrides any selections by the MIDI channel.

This command *must* be the first one in the macro.

Parameters:

- Voice = Specifies the voice number to use (0-3). A value of 255 keeps the voice chosen by the MIDI sequence
- Rst.Flag = If this flag is set to ON the voice selected will not be reset. This is useful to take control over the voice in its present state.
- Toggle Flag = If this flag is set to ON and the voice selected by the MIDI sequencer is voice 1 or 2, the voice really used to play the sound will toggle between voice 1 and 2 on every key played. This is very effective to create echos.

STARTSAMPLE

Plays a sample

Type: Voice/Sample

		Sample-ID <i>Sample ID</i>							
\$10	STARTSAMPLE								

Description:

Plays back the specified sample.

Parameters:

Sample ID = The ID of the sample to play back.

VOICE_OFF

Stops sound

Type: Voice/Sample

\$11	VOICE_OFF								
------	-----------	--	--	--	--	--	--	--	--

Description:

Stops the sound on the current voice.

VOICE_ON

Starts sound

Type: Voice/Sample

\$14		VOICE_ON	Duty cycle Duty Cycle						
------	--	----------	--------------------------	--	--	--	--	--	--

Description:

This command starts the sound after all initial setups (if the initialization phase was not overridden by a SET_VOICE command). The *DutyCycle* parameter is used for the rectangular wave oscillator (voice 1/2), with values from 0-3 representing 12.5, 25, 50 and 75 percent pulse-width. A duty-cycle of 255 can be used to have the velocity influence the pulse-width.

For voice 3 use the WAVE_ON command.

For voice 4 the parameter is unimportant.

Parameters:

DutyCycle = Specify the duty cycle to use (see above). Enter 255 to have the velocity modify the duty cycle (0-31=12.5%, 32-63=25%, 64-95=50%, 96-127=75%)

SETNOISE

Sets parameters for the noise generator

Type: Voice/Sample

	Poly.clock	Poly.Step	Freq.ratio					
\$15	SETNOISE	Clock	Step	Freq.				

Description:

Sets up the polynomial clock counter for the white noise generator (voice 4).

Parameters:

Clock = Values from 0-13 select the ratio of frequencies

Step = If 0 selects 15 steps for the counter, 1 selects 7 steps

Freq. = Values from 0-7 select the frequency ratio

PWM_START

Starts the Pulse-Width-Modulation effect on voice 3

Type: Voice/Sample

		Low limit	High limit	Speed				
\$1F	PWM_START	LimitLo	LimitHi	ms				

Description:

Starts a software generated pulse-width effect on voice 3. The width of the pulse will change over time (in a ping-pong-like fashion) between the specified low and high limits.

Parameters:

LimitLo = Low limit of the pulse width (0-15)

LimitHi = High limit of the pulse width (0-15)

ms = Time in milliseconds it takes to complete one pulse cycle

PWM_UPDATE

Updates the Pulse-Width-Modulation effect on voice 3

Type: Voice/Sample

		Low limit	High limit	Speed				
\$20	PWM_UPDATE	<i>LimitLo</i>	<i>LimitHi</i>	<i>ms</i>				

Description:

This is basically the same as the PWM_START command. It modifies the pulse width limits and the pulse cycle time, without restarting the effect.

Parameters:

LimitLo = Low limit of the pulse width (0-15)

LimitHi = High limit of the pulse width (0-15)

ms = Time in milliseconds it takes to complete one pulse cycle

PWM_FIXED

Starts a generated pulse wave of fixed width

Type: Voice/Sample

\$21	PWM_FIXED	Duty									
------	-----------	------	--	--	--	--	--	--	--	--	--

Description:

Starts a fixed generated rectangular pulse wave on voice 3. The duty parameter specifies the pulse-width from 0-15, which is equivalent to a 0-50% duty cycle.

Parameters:

Duty = Width of the rectangular pulse (0-15)

PWM_VELOCITY

Starts a generated pulse wave of velocity-dependent width

Type: Voice/Sample

\$22	PWM_VELOCITY								
------	--------------	--	--	--	--	--	--	--	--

Description:

Starts a fixed generated pulse wave on voice 3. The pulse-width is set according to the current key velocity.

WAVE_ON

Loads a looping wave into voice 3 and starts it

Type: Voice/Sample

\$26		WAVE_ON	1 Block_SMPID	Sample ID							
------	--	---------	---------------	-----------	--	--	--	--	--	--	--

Description:

This command loads a short, 32 sample long looping waveform into the WaveRAM and starts voice 3 after all initial setups (if the initialization phase was not overridden by a SET_VOICE command).

Parameters:

SampleID = ID of the sample to load into Wave RAM. If the ID is 0 no new sample will be loaded and the contents of the Wave RAM remain unchanged.

Volume/Pan Macros

SETADSR

Uses a software ADSR envelope on the current voice

Type: Volume/Panning

		Table-ID (ADSR)						
\$0C	SETADSR	Table						

Description:

The data from the specified ADSR table will be used to perform an ADSR envelope on the current voice.

Parameters:

Table = The ID of the ADSR table to use

SETVOLUME

Sets an absolute volume

Type: Volume/Panning

\$0D SETVOLUME		Volume	Volume								
----------------	--	--------	--------	--	--	--	--	--	--	--	--

Description:

Sets a fixed volume on the current channel.

Parameters:

volume = Specifies an absolute volume for the current channel (0-127).

PANNING

Sets the panning to be used with the macro

Type: *Volume/Panning*

\$0E PANNING		Pan position <i>pan.pos</i>							
--------------	--	--------------------------------	--	--	--	--	--	--	--

Description:

Sets the position for the current voice channel. Game Boy hardware only allows for absolute left, absolute right and center positions.

Parameters:

pan.pos = 0-41 designates left output, 42-83 center and 84-127 right

ENVELOPE

Starts a software envelope

Type: Volume/Panning

		Envelope/Fade-in	Milliseconds			
\$0F	ENVELOPE	Flag				ms

Description:

Starts a software envelope. The volume will be faded out/in to mute level or full volume in the time specified. Due to Game Boy hardware restrictions, this may be of lower quality than the hardware envelope.

Parameters:

Flag = If OFF, fades out to mute level, if ON fades in to maximum level

ms = Time to fade out to zero in milliseconds

HARDENVELOPE

Starts hardware envelope

Type: Volume/Panning

		Envelope/Fade-in						Milliseconds
\$1E	HARDENVELOPE	Flag						ms

Description:

Starts a hardware envelope. The volume will be faded out/in in the specified time, which cannot be longer than 1640ms and is dependent on the current volume. By employing the hardware feature for the envelope, the sound might be slightly less distorted in comparison with the software envelope. The software envelope, however, can span longer fading times and also works on voice 3.

Parameters:

Flag = If OFF, fades the voice down to mute level. If ON, fades the voice in from the current volume to maximum.

ms = Time in milliseconds (highly approximate!) for the fade to complete. Due to hardware restrictions this value cannot be larger than 1640 ms for a fade from maximum volume.

Pitch Macros

RESET_MOD

Reset all pitch modulations

Type: *Pitch*

\$04	RESET_MOD								
------	-----------	--	--	--	--	--	--	--	--

Description:

This command will stop and reset any active pitch modulation on the current voice channel.

STOP_MOD

Stops any pitch modulation

Type: *Pitch*

\$0A	STOP_MOD							
------	----------	--	--	--	--	--	--	--

Description:

Stops any pitch modulation on the current voice channel, but does not reset the current values.

PORLAST

Portamento from the last note

Type: *Pitch*

		Transpose Keys	Detune Cents				Milliseconds <i>Ms</i>
\$16	PORLAST						

Description:

The pitch will slide from the last known value to the pitch of the current key, plus the keys and cents parameters, in the specified time.

Parameters:

- | | | |
|-------|---|--|
| Keys | = | Transposes the current key by this value (-127 ~ 127) |
| Cents | = | Transposes the current key by a fraction of one key (-99 ~ 99)
Can be used together with the whole key transposed |
| ms | = | A time in milliseconds for the portamento to be finished |

RNDNOTE

Creates a random pitch

Type: *Pitch*

	Note Lo <i>note-lo</i>	Detune <i>detune</i>	Note Hi <i>note-hi</i>	Fixed/Free <i>fix/free</i>	Abs/Rel <i>Rel/abs</i>		
\$17	RNDNOTE						

Description:

Sets random pitch. Note lo is the lower end of the range, note hi the upper end. The detune value will be added after the random pitch is calculated and is specified in positive cents. If the *free* flag is set, the pitch will be generated freely inside the range. Otherwise, a random key from this range will be generated. If the *abs* flag is set, the specified range is absolute. Otherwise it is relative to the current key.

Parameters:

- note-lo = Specifies the low key for the random range
- detune = Specifies a fraction of a key (0-99) to be added to the random result in the end
- note-hi = Specifies the top key for the random range
- fix/free = If OFF, a random key will be generated inside the specified range. If ON a random pitch inside the permissible range is generated without respect to any keys.
- rel/abs = If OFF, the range is relative to the current key. If ON the range is fixed by the specified keys

ADDNOTE

Modifies the current key by offset values

Type: *Pitch*

\$18	ADDNOTE	Add <i>add</i>	Detune <i>detune</i>	Org Key <i>org.key</i>				
------	---------	-------------------	-------------------------	---------------------------	--	--	--	--

Description:

Transposes the current key by a number of keys and a cent fraction.

The result is temporary when the *org.key* flag is set, so that further ADDNOTE commands will again take the MIDI key as base. If the flag is OFF, the result of this command will be considered to be the new base.

Parameters:

add = Value (-127 – 127) to transpose the current key by

detune = Key fraction (-99 – 99) to transpose the current key by

org.key = If set to OFF the result will form a new base key. If set to ON the result is temporary until the next ADDNOTE command.

SETNOTE

Sets pitch to a fixed note

Type: *Pitch*

	Key <i>Note</i>	Detune <i>Detune</i>							
\$19	SETNOTE								

Description:

Sets the pitch to a fixed note and detune in cents.

Parameters:

key = The key number to set the pitch to (0-127)

detune = The fraction of a key (-99 – 99) to add to the pitch

LASTNOTE

Retrieves the last note of the current voice

Type: *Pitch*

\$1A	LASTNOTE	Add	Detune						
		Add	detune						

Description:

Recalculates the current key by transposing the last key played on this voice.

Parameters:

add = Number of keys to transpose the last key by (-127 – 127)

detune = The fraction of a key to transpose the last key by (-99 – 99)

PORRAMENTO

Starts a portamento

Type: *Pitch*

	Range Note	Range Detune	Abs/Rel	Milliseconds	
\$1B	PORTAMENT.	Keys	Cents	Flag	ms

Description:

Slides the pitch from the current pitch to a target pitch specified in the given time.

Parameters:

- Keys = Transpose value for the current key to yield the target key of the portamento (-127 ~ 127)
- Cents = Transpose value in fractions of a key to yield the target key of the portamento (-99 ~ 99)
- Flag = If OFF, the key and cents specified form the target of the portamento. If ON, the key and cents are added to the current key (relative mode) to yield the target of the portamento.
- ms = A time in milliseconds for the portamento to be finished

VIBRATO

Starts a vibrato effect

Type: *Pitch*

	Level note	Level fine				Milliseconds
\$1C	VIBRATO	Keys	Cents			ms

Description:

Adds a vibrato effect to the current voice. The intensity of the vibrato is specified by a displacement of the current key by a number of keys and cents. When the *keys* parameter is negative, the effect will start to decrease in frequency. Otherwise, the frequency will first increase.

Parameters:

- Keys = Intensity of the vibrato in keys relative to the current key
Cents = Intensity of the vibrato in fractions of a key relative to the current key. This value is added to the *Keys* parameter.
ms = Time in milliseconds for a full frequency cycle to complete.

PITCHSWEEP

Adds a sweep effect to the pitch

Type: *Pitch*

	Note Limit	Cent Limit		Sweep 0/1	Milliseconds
\$1D PITCHSWEEP	<i>Limit</i>	<i>Limit fine</i>		Select	Ms

Description:

Adds a sliding effect to the current pitch. After reaching the limit, the pitch wraps back and the slide starts again. There are 2 independent modulators that can be selected by the *Select* parameter. If the *Limit* is negative the sweep goes downwards, otherwise upwards.

Parameters:

- Limit = Specifies the number of keys to slide up or down relative to the current key.
- Limit fine = Specifies the fraction of a key in addition to the full keys to slide up or down to.
- Select = If 0 selects sweep effect 1, if 1 selects sweep effect 2. Two independent sweep effects which may work against each other can be started.
- ms = Time for one sweep cycle to complete in milliseconds

Special Macros

SENDFLAG

Raises a flag the application can evaluate

Type: Special

		Flag Bit							
\$23	SENDFLAG	Num							

Description:

Raises one of 8 user flags the game application can evaluate.

This feature is mainly used to signal certain events to the game program.

A raised flag remains raised until the application has read its status, at which point the flag will be cleared again.

Parameters:

num = Number of the flag to raise (0~7)

SAMPLEMAP

Builds the sample map macro

Type: *Voice/Sample*

		SMP ID							
\$24	SAMPLEMAP	Sample ID							

Description:

This command can be used multiple times inside a single macro, for the entire project to define the order of the samples that can be used with the PLAYKEYSAMPLE command. Each such macrostep references a sample that will be assigned to the keynumber that is equivalent to the macro step number.

Parameters:

Sample ID = ID of the sample to assign to the midikey equivalent to the macro step number

CURRENTVOL

Fakes the internal volume

Type: Special

\$25		CURRENTVOL	Volume								
------	--	------------	--------	--	--	--	--	--	--	--	--

Description:

This command is used to change the internal volume in the sound system, only. The real voice volume remains unaffected. This is necessary after the HARDENVELOPE command has been used which, due to hardware limitations, leaves the sound system uninformed about the real hardware volume.

Use this command in conjunction with carefully timed macros, to tell the soundsystem your idea of the current hardware volume (usually after a full fade-in or fade-out).

Parameters:

volume = MIDI volume (0~127) to set as a fake value in the sound system

ADD_SET_PRIO

Changes the priority of a sound effect

Type: Special

	Prio	Add/Set							
\$27	ADD_SET_PRIO	Value	Flag						

Description:

Modifies the current priority of a sound effect. Depending on the state of *Flag*, the *Value* is either set directly or added to the current priority.

Parameters:

- Flag = If OFF, adds the *Value* parameter to the current priority. If ON, sets the *Value* parameter immediately as the new priority.
- Value = Ranges from -128 to 127. If the Flag parameter is OFF, it will be added to the current priority. If the Flag parameter is ON, this value will be set as an absolute priority (using the 2's complement of the value. So -1 becomes 255 and -128 becomes 128).

Appendix 2.4 – Performance Issues

Because of the age and nature of the Game Boy architecture, some macro commands require more CPU performance than others.

This means that the musician has, to some extent, direct influence on the overall performance of the game application.

If the game logic is simple enough, as for instance in a puzzle game like Tetris, performance might not be much of an issue. More complex games, like action shooters on the other hand, could suffer if the music is using a lot of special features. This would require more performance to be spent on the music.

The table on the following page puts all commands in categories, from 1 through 5, with 1 being the category that requires the **most** performance.

All commands that do something just once might sometimes require more time than other commands, just because *what* they are doing is more complex.

Category	Command(s)
1	PORTAMENTO PORTLAST
2	VIBRATO PITCHSWEEP PWM_START, PWM_UPDATE PWM_FIXED, PWM_VELOCITY PLAYKEYSAMPLE, START_SAMPLE SETADSR ENVELOPE
3	RNDNOTE PLAYMACRO SPLITRND VOICE_ON WAVE_ON
4	VOICE_OFF ADDNOTE, SETNOTE, LASTNOTE SETVOLUME SPLITKEY, SPLITVELOCITY WAIT LOOP KEYOFF SETVOICE
5	RESETMOD STOPMOD PANNING SETNOISE GOTO HARDENVELOPE SENDFLAG ADDSETPRIO TRAPKEYOFF, UNTRAPKEYOFF CURRENTVOL ENDMACRO, STOP SAMPLEMAP

Appendix 2.5 - Troubleshooting Guide

This section is intended to provide answers to potential problems you might encounter.

I start the slave program but it fails to connect to Game Boy

There are a number of possibilities. Please read through every one of them to find out which one applies to your situation.

- Is the link cable securely connected to your PC's parallel port on the one end and securely to the Game Boy Color on the other end?
- Is Game Boy Color switched on?
- Does the Game Boy Color have a fresh set of batteries, if you are not using an AC adapter?
- Try using a fresh set of batteries instead of an AC adapter. Some adapters may cause too much interference for the serial link.
- Is the slave cartridge properly inserted into the Game Pak slot? When you turn on the Game Boy, you should see a screen similar to this.



- Did you use an MBC-5 Flash ROM when you flashed the slave program for Game Boy Color?

Windows 95/98:

- Have you specified to correct parallel port in the configuration program of the slave?

Windows NT 4.0

- Have you installed the device driver?
 - Have you specified the correct parallel port address in the registry?
 - Have you disabled the parport device driver?
-
- If you are using an on-board parallel port of your PC's mainboard, is it set from the BIOS to be a standard parallel port (no ECP/EPP mode with DMA)?
 - If you are using an on-board parallel port of your PC's mainboard, is it accidentally disabled from the BIOS?

I get the Game Boy to recognize a connection but it seems to get stuck immediately

Once the Game Boy Color acknowledges a connection to the slave program, the display of the Game Boy will look like this.



If you do not get out of the negotiating phase, check for the following.

- Does the Game Boy Color have a fresh set of batteries, if you are not using an AC adapter?
- Try using a fresh set of batteries instead of an AC adapter. Some adapters may cause too much interference for the serial link.
- Did you use an MBC5 Flash ROM when you flashed the slave program for Game Boy Color?
- If you are using an on-board parallel port of your PCs mainboard, is it set from the BIOS to be a standard parallel port (no ECP/EPP mode with DMA)?

Once you are out of the negotiation phase the Game Boy screen should look like this.



If you are getting errors in the slave program Link status window like this,



OR



check the following.

- Does the Game Boy Color have a fresh set of batteries, if you are not using an AC adapter?
- Try using a fresh set of batteries instead of an AC adapter. Some adapters may cause too much interference for the serial link.
- Did you use an MBC5 Flash ROM when you flashed the slave program for Game Boy Color?
- If you are using an on-board parallel port of your PCs mainboard, is it set from the BIOS to be a standard parallel port (no ECP/EPP mode with DMA)?

I send a project to the slave but it does not download any data

This is most likely caused by an error in your project.

All references in all sound macros, for instance, need to be resolved before a project that is suitable for the Game Boy can be created and downloaded by the slave program.

Typical project errors are:

- Another macro referenced in a sound macro that does not yet exist, or no longer exists. The editor will have assigned the reserved ID 0 (zero) for it.
- An ADSR curve referenced in a sound macro that does not yet exist, or no longer exists. The editor will have assigned the reserved ID 0 (zero) for it.
- A sample referenced from a sound macro that does not exist. Here the reserved ID 0 (zero) will also have been assigned.
- SETVOICE command inside a macro is not step 0.
- A soundeffect macro contains no SETVOICE instruction.
- More than 256 macros used.
- More than 256 ADSR curves used.
- More than 256 samples used.
- Out of project memory.
- Out of sample memory.
- More than one (1) keymap macro in the project.
- A loop command in a macro that does not loop back above the command itself.

I made changes to my project but the slave tells me there are none

This is caused by accuracy problems in the Game Boy target format.

Sometimes Game Boy can not exactly represent values you specify. For instance, a wait time can be specified in milliseconds, but the millisecond resolution on Game Boy is a raster of 16.67ms. Hence the values 5, 10 and 16 would all be treated like 16.67ms by the Game Boy hardware and effectively cause no change in the project data.

While I'm working the sound slave disconnects itself

The slave program verifies the connection to the Game Boy hardware every second. During this phase, it also determines if the highest possible transfer rate can still be obtained.

If the slave program detects that the link to Game Boy has been severed or too many data transmission errors occur, which can not be corrected, it will terminate the connection and close itself. In these cases this popup window will be displayed.



The most common causes for too many transmission errors are:

- The Game Boy is running on batteries rather than an AC adapter and the batteries are too weak to supply the power for the serial port.
- The link cable is intertwined with other cables or is routed alongside a monitor or other electrical devices, which causes interference in the link cable.
- A 'noisy' AC-adapter is used to supply power to the Game Boy. Use only a stabilized adapter that does not introduce any artifacts like humming sound or noise in the Game Boy (can usually be quickly verified by connecting a headset to the Game Boy and listening closely for any disturbances).

I send MIDI data but I do not hear anything

First, you need to make sure that the project data was all valid. If you send the project to the slave program but nothing is downloaded to Game Boy, please refer to the previous section in this trouble-shooting guide titled, "I send a project to the slave but it does not download any data".

Step #1:

Assuming that your project has been downloaded to Game Boy, we must first verify that the slave program receives MIDI data. To do this, keep an eye on the 'MIDI data' window in the slave program, while pressing a key on your MIDI keyboard. Is the received MIDI data printed out in the slave's MIDI data window?

If yes, please skip right to step #2.

Otherwise:

- Verify that your MIDI cabling between your keyboard (or whatever MIDI device you are sending the MIDI data from) and the PC running the slave program is correct.
- Verify that the slave program is using the correct MIDI device for its data. A list of all accessible devices appears in the main log when you start the slave. Right after this list, it tells you which device number corresponding to the list was opened. If this is the wrong device, please reconfigure the slave using the configuration program.
- If you are using a single machine setup with multiple MIDI input and output devices *and* you are sending MIDI data from your PC to an output device (which you have also opened for input in the slave), verify that a hardware loopback device is installed to send the MIDI data back into the PC. This can be circumvented by using a MIDI loopback device (software based) like "MIDI Yoke" or "HuBi" or a two machine setup with a dedicated MIDI input device.
- Verify that the hardware port used for MIDI data input is operational. The best way to do so is to monitor the MIDI input signal indicators on your sequencer program (like CuBase).

When you have found the problem please fix it and try again.

Step #2:

Now that the slave program is receiving MIDI data, does the Game Boy itself receive it?

Examine the line "Receiving MIDI" on the Game Boy screen. Does a rotating indicator appear while you are sending MIDI data?

If yes skip ahead to Step #3.

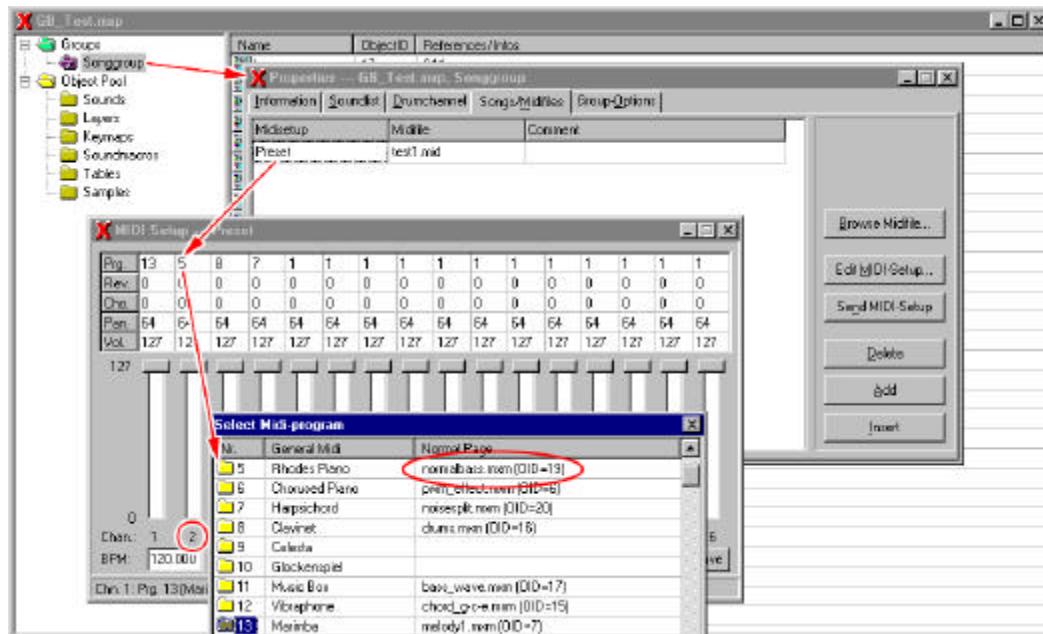
Otherwise:

- Are you currently downloading anything into Game Boy? While a project is being downloaded to Game Boy all incoming MIDI data is discarded. This is normal.
- Are you sending data on a MIDI channel >4? Only the first four MIDI channels can be used.

Step #3:

The reception of MIDI data by Game Boy suggests that there is a fault in either the MIDI setup or the sound macro itself.

- Verify that the correct MIDI program for the MIDI channel being used is set in the MIDI setup for your sound group in the *MusyX* editor.
- Verify that the correct sound macro has been assigned to the program number that you have assigned to the MIDI channel on which you send data. Refer to the screen shot below.



(follow the path of the arrows above, in determining these settings)

- Verify that the macro contains either a VOICE_ON (for voices 1, 2 and 4) or WAVE_ON, STARTSAMPLE, PLAYKEYSAMPLE, PWM_START, PWM_FIXED or PWM_VELOCITY (for voice 3) instruction.
- Verify that the macro does not contain a SETVOLUME 0 instruction.
- Verify that the velocity of the key you are sending is not 0.

In addition, you need to make sure that you did not assign a macro, created for instance for voice 3, to another voice. Although this will not be harmful to the system, mixing macros and their implicitly associated voices will rarely produce desirable results. The only occasion where this does not make a difference is the usage on either voice 1 and 2, since these are of identical design.

You may always override any implicit voice selection (made by MIDI channels 1-4 which map to voices 1-4) with the macro instruction SETVOICE. This must appear as the very first instruction inside a macro.

In doing so, you can reroute any macro to any voice you want. This requires careful design of your sound sequence. Using the SETVOICE command in a song macro is usually discouraged, unless used to select the Voice1/2 toggle option.

The SETVOICE command, however, is required for every sound effect macro, since they have no default voice assignment.

When I play a sample I hear a humming sound

The Game Boy hardware was originally not intended to play back music or sound which contains more than 32 samples. To reproduce samples which are longer, Game Boy needs to be fed the samples in parts containing 32 samples each. While feeding it these parts, sound output needs to be stopped temporarily (due to hardware restrictions) and then restarted. This stopping/restarting causes an audible distortion.

Depending on the quality of the sample (either 1920Hz or 8192Hz), this distortion changes frequency as well, due to the fact that the voice needs to be fed those 32 sample chunks at a different rate.

At a sample rate of 1920Hz, the distortion has a frequency of 60Hz and 256Hz at a sample rate of 8192Hz.

Also, this distortion is always of a fixed volume, regardless of the volume setting of voice 3. Therefore, it becomes much more audible when the sample is not played back at full volume. We recommend playing back samples at the highest possible volume to minimize notable distortions.

This problem does not arise when you use looping music or sound of only 32 samples in length, since no further reloads are necessary.

There seem to be timing problems with regard to the start and length of notes

The shortest note Game Boy can handle is 1/16th. This is due to internal timing resolutions of the Game Boy hardware vs. acceptable CPU performance. Because of this, the length of a note should always be a multiple of 1/16th.

We suggest that you quantize your arrangement to 1/16th to minimize any timing artifacts.

You may still experience timing problems with a 1/16th note limit when you play your song from your MIDI sequencer. These problems will vanish once the song is integrated in the game application and played back solely by Game Boy.

The reason for this lies in the fact that the PC and Game Boy, in the master/slave setup, are not synchronized while creating a song. Whenever a note is to be played back *during* this Game Boy internal 1/16th raster, the note will be delayed in its keyon until the next scheduled point in this raster.

To slightly compensate for this, a keyoff is sent to a note immediately. This does not necessarily cause anything to happen right away. But should the keyoff be received while servicing the voice or before, while already being in the raster, the voice will react to the keyoff. If the time frame for the corresponding voice has elapsed, the keyoff will be delayed until the next scheduled time within the raster.

Once implemented in the application this no longer applies, since the Game Boy synthesizer is of course in sync with its own sequencer.

Further timing problems while working on the song which are beyond any control are:

- Windows MIDI device delays.
- Windows preempting a task.
- Network delays in multiple machine setups.
- Recurring communication errors with Game Boy.

APPENDIX 3 – N64 Programmers Reference

Table of Contents:

<i>MusyX</i> Basic Architecture	257
<i>MusyX</i> and MORT Voice Compression.....	260
Performance Impact on the Game Application.....	261
Requirements for Services Provided to <i>MusyX</i>	264
Reverb Effect Engine - REE.....	266
Volume Control.....	266
IDs.....	267
3D API.....	267
Function Section: SOUND	269
SND_INIT	269
SND_QUIT.....	271
SND_SHUTDOWN	272
SND_GET_PLAYBACKINFO.....	273
SND_REINIT	274
SND_VOLUME	276
SND_MASTER_VOLUME	277
SND_MONO	278
SND_PLAY	279
SND_STOP	282
SND_PAUSE	283
SND_SILENCE	284
SND_IS_IDLE.....	285
SND_CROSSFADE	286
SND_CROSSFADE_DONE	289
SND_CONTINUE.....	290
SND_MUTE	291
SND_SPEED	292
SND_SEQLOOP	293
SND_GET_SEQLOOPCNT	294
SND_GET_SEQVALID	295

Function Section: SOUND (Continued)

SND_SEQ_VOLUME.....	296
SND_GET_SEQVOLGROUP	297
SND_ASSIGN_VGROUP2TRACK	298
SND_FXSTART.....	299
SND_FXKEYOFF	300
SND_FXCHECK	301
SND_FXPANNING	302
SND_FXSURROUNDPANNING	303
SND_FXVOLUME.....	304
SND_FXPITCHBEND	305
SND_FXMODULATION	306
SND_FXPEDAL.....	307
SND_FXDOPPLER.....	308
SND_FXREVERB.....	309
SND_PUSHGROUP	310
SND_POPGROUP.....	311
SND_READFLAG	312
SND_WRITEFLAG	313
SND_ALLOC_STREAM.....	314
SND_STREAM_ALLOCLength	316
SND_STREAM_MIXPARAMETER	317
SND_FREE_STREAM.....	318
SND_ACTIVATE_REVERB	319
SND_DEACTIVATE_REVERB	321
SND_ADD_LISTENER	322
SND_UPDATE_LISTENER	324
SND_REMOVE_LISTENER	325
SND_ADD_EMITTEREX	326
SND_ADD_EMITTER	329
SND_UPDATE_EMITTER	330
SND_REMOVE_EMITTER	331
SND_CHECK_EMITTER	332
SND_EMITTER_FID.....	333

Function Section: VoiceLib MORT Interface **334**

VOICE_INIT	334
VOICE_EXIT	335
VOICE_SET_DIRECTORY	336
VOICE_START	337
VOICE_STOP	339
VOICE_CHECKACTIVE	340
VOICE_PARAMETERS	341
VOICE_GET_TIME	342
VOICE_SYNC_IDLE	343

MusyX Basic Architecture

Every sound system has one central element, the instrument or sound effect, depending on whether you are speaking about music or effects. Sometimes they are handled separately from each other, sometimes the only difference is the way they are started. **MusyX** follows the latter approach. According to the unified way these two entities are handled, we will use just one name, Sound.

A sound is the central element in **MusyX**. Although the name suggests that the sound really is nothing more than a sample and a few parameters, things are quite different within **MusyX**.

In **MusyX** a sound actually represents a little program that is executed in a tokenized form at run time. This allows the music / sound designer more control over the produced sound.

From a programmer's point of view, these details are quite well hidden. The only time a programmer gets in contact with the macro program is when data is exchanged between the sound and application program.

As mentioned before, there are two basic types of any kind of sound reproduction within **MusyX**. Instruments are used in the context of a piece of music, called a song. Each song has its very own ID, to identify which song is to be started once the application decides to do so.

Songs always use a MIDI-like, but much more powerful, proprietary representation. Sequenced music reproduction was chosen over streamed audio, since it offers much more flexibility.

Streamed audio is supported, but the application programmer will have to take care of the actual streaming process. **MusyX** just offers buffers for streaming data.

Sound effects are directly accessed using an automatically generated, unique sound effect ID. The sound designer has the possibility of accessing multiple sounds using just one ID, but this detail is totally hidden from the programmer.

Both sound effects and songs are grouped together to form units called **Groups**. Groups are the basic data element that the programmer has to deal with. Groups are used to build small units of data that can be more easily managed than the whole project. On CD based systems or other systems that cannot access their mass storage device in real time, saving ROM space is always an issue. This way of structuring the data is meant to help with that.

For example, the musician and the programmer could agree on setting up a group containing all basic sound effects, and yet another one that contains all basic jingles needed during normal game play.

These groups probably would be present all the time, while groups containing special boss songs and sound effects could be loaded as needed. **MusyX** takes care of all data that is defined multiple times, so that every item is stored just as often as needed.

To manage the groups, **MusyX** uses a stack structure. We will make references to it as "soundstack". Groups are pushed onto the stack and can be removed by just popping them off the top of the stack. This is done to prevent memory fragmentation in internal structures of the sound system and to simplify the data management used to prevent multiple data storage.

Data is not actually copied when it's pushed onto the stack. The programmer has to keep most of the data around as long as the references remain on the stack. There are several types of data that the programmer has to deal with. The data is saved in separate files to allow for more flexible data management.

Project Data

Project data is one of two data types, which are **not** pushed onto the soundstack. It is actually used to represent the logical structure of the project the musician hands over to the programmer, and to enable the programmer (and the system) to access all the other data easily.

Pool Data

This set of data contains macros and all data that is referenced by these, with the exception of samples. The musician uses macros to describe the kind of sounds the synthesizer is to produce. Since this data is necessary to reproduce the sound, it has to be present in RAM all the time. The size of this data block is often quite small.

Sample Data

This data contains all the samples to be used. It therefore, can be quite large. This type of data sometimes can be discarded after it has been pushed onto the soundstack. This will be the case if the system features a separate sound RAM.

Some systems even allow specifying a ROM location, so that the samples do not have to be present in RAM. So does the N64.

Some platforms, like Game Boy, don't need sample data at all or just in a very limited way (Game Boy).

Sequencer Data

All the data about the song(s) is stored here. The size very much depends on the complexity and length of the song(s) contained within. Just the currently played or paused songs need to be loaded. The sequencer data itself is not pushed onto the stack, but is just specified as a reference while pushing a group containing song data onto the stack.

The pool, sample, and sequencer data sections are always used together with the Project Data. There may be multiple sets of these data sets in one project file. This is because the data may be split up into multiple sections, to limit the amount of memory used at one time.

MusyX and MORT Voice Compression

MORT is currently not included with **MusyX**. Nevertheless, to integrate MORT as easily as possible into **MusyX**, the “voicelib” library has been implemented. It uses **MusyX**’s standard streaming interface to pass the decompressed MORT data into **MusyX**’s system. The functions are simply a wrapper, to hide the details of the implementation from the game application and thereby make things easier for the game programmers.

The voice library introduces a new way to store MORT compressed samples. To make the data easier to handle, all MORT compressed sample files are joined in one large “MORT directory file”. The tool generating this file will also write a header file containing defines, that make it possible to reference the MORT samples contained in the directory file by their name.

Each running MORT data stream will need about 5% of the total CPU power. The overhead produced by passing the data through **MusyX** is minor. The RSP workload for a MORT voice will be less than for a standard ADPCM sample, since the decompression takes place on the host CPU.

Performance Impact on the Game Application

MusyX has been designed to offer a maximum of flexibility at a minimum of CPU and RSP performance impact. Nevertheless, there are a couple of things that one has to keep in mind to yield better results.

On the average, **MusyX** will use about 0.7% to 0.9% of the CPU's and about 0.7% to 0.8% of the RSP's calculation time per active voice in an ideal setup. The impact the system has on ROM transfer or RDP performance is hard to profile. There are just too many variables in the equation. Nevertheless, there are a couple of basic thoughts that may help evaluate the situation.

MusyX uses a cache system to optimize ROM transfers of samples. This cache is very efficient even at small sizes. A sample set of 3Mb may work very well with a cache area of just about 90Kb. The size of the cache rarely needs to be larger, but as always this largely depends on what is going on in the system. A larger cache may be helpful if you use a lot of different and long samples, or your application needs to transfer extreme amounts of data over the PI bus.

The impact on the RDP from the sound generation on the RSP is also hard to evaluate. The microcode designed for **MusyX** has been optimized to use the RSP's time as little as possible. If your game is limited by RDP fill rate, the RSP performance hit for MusyX should not affect the RDP too much, since the RSP will be waiting for the RDP to finish drawing anyway. The worst situation for the RSP is reached when it has a lot of very small triangles to draw. The RSP is very much involved in the triangle setup during drawing and will almost never wait for the RDP, if the FIFO is large enough. This will cause any RSP performance hit to effect the RDP in a very pronounced way. Yet again, it's obvious that one has to evaluate the specific needs of the application.

Generally, one can say that reducing the amount of voices processed tends to have a larger impact on the performance than reducing the mixing rate. **MusyX**'s priority system offers a lot of ways to control the use of a very small number of voices. Tests have shown that about 20 voices at 22KHz are a good compromise, and still allow for big orchestral scores if needed.

Reverb is another element that can cost quite a bit of RSP time. The CPU is only indirectly impacted by the increase of DMA traffic to the RSP. This impact will usually be very small.

An average reverb setup while filtering the output will cost about as much as a voice on the RSP. Generally, one can say that the reverb calculations take more time when more DMA is needed. The worst case would obviously be if all 8 reflections are used, and are spread out over a very wide range in the reverb buffer.

All issues up to now have been primarily of importance to the programmer, but the musicians have influence on the performance, too. Here are some things to think about.

The SMaL language has been designed with performance in mind. When you design macro programs, you'll notice that these programs are most often waiting for something. This "idle" state is obviously the least performance eating state. But one can write macros that handle a lot of commands in a row, which will lead to a larger performance hit. Common sense presents a good rule of thumb. Do what you have to do, but only use the features you really need for a specific task.

The MIDI controller-mapping feature offers a lot of flexibility. Nevertheless, one has to keep in mind that the controller data is needed every frame, whether new macro commands are handled or not. So, the more complex the combinations which are used simultaneously, the more calculation time is needed.

Pitching up samples can cost calculation time. If samples are pitched up so far that their required playback frequency is above the mixing rate, the system has to provide more data to the resampler than will be output. Normally this will have little impact, but in extreme situations it may be necessary to consider this. (Keep in mind that, due to memory limitations in the RSP, the system cannot handle playback frequencies above twice the mixing frequency.)

Memory is another resource of which one never has enough. It may be useful to know that pitchbend and modulation wheel controllers are specially optimized to use up very little memory in song files. All other controllers are stored in a simpler, more RAM consuming way. So if you are going to use a lot of controller data over time, you should consider mapping the input that you want to control to these controllers.

Each running MORT data stream will need about 5% of the total CPU power. The overhead produced by passing the data through *MusyX* is minor. The RSP workload for a MORT voice will be less than the RSP workload for a standard sample, since the decompression takes places on the host CPU.

Requirements for Services Provided to **MusyX**

MusyX needs a couple of services provided by the game application. Here's a list of these services and the requirements that **MusyX** imposes on them.

VBL Hooks

MusyX requires two functions from the application to install and remove a VBL handler. **MusyX** will only use these functions during system initialization and shutdown. The VBL handler needs to be called at 50 to 60Hz. All internal structures and lists are optimized for a minimum calling frequency of about 50Hz. A lower frequency would make the system's timing far too slow to guarantee proper playback of music, and would concentrate the performance hit in a very unsymmetrical fashion.

Memory Allocation

MusyX requires basic memory allocation services. These services will only be used during system initialization and shutdown, to avoid unnecessary memory fragmentation. In addition to the required size of the memory block, **MusyX** will tell the service routine if the memory block will be needed for a long time or if it will be freed shortly using a flag field. All memory blocks allocated must be aligned to 16 byte boundaries (cache boundaries).

RSP Yielding

The system needs a function to start an audio task on the RSP, and one to wait for the task to be finished. The function starting the audio task should return immediately and leave the "potential waiting for the RSP's yield" to a separate thread. The function should then start the task as soon as possible. The wait function will be called by **MusyX** one frame after the audio task start has been issued to the RSP scheduler to be implemented by the application. This is done to support the rare instance when the RSP can not finish audio processing within one frame. The game application should try to ensure that this is a rare exception. Nevertheless, the application has total freedom to delay the RSP's audio task start as it sees fit, as long as the audio task is finished within one frame.

DMA Services

Once a frame, **MusyX** will issue a list of ROM data blocks to be transferred to the game application. This is done during the VBL handler. **MusyX** expects these data packages to be transferred to the specified RAM addresses within one frame. How the application takes care of this task, is totally up to the game programmer. As with yielding the RSP, **MusyX** only expects that the application handles the request in a separate thread. The function called by **MusyX** should be returned as fast as possible.

One possible way to handle DMA transfers would be to subdivide any DMA transfer by the game applications into smaller blocks, so that the sound system's requests can be handled in time, even if a very large data block is downloaded by the application. **MusyX** uses a cache system to minimize the amount of data to be transferred.

M.O.R.T. Services

The voice library uses the services provided to **MusyX** by the game application, to allocate buffers and initiate DMA transfers (see initialization routines for details). In some modes, the voice library will expect to be able to allocate and free memory at run-time. The library uses no additional VBL hooks, but installs hooks internally within **MusyX**.

Reverb Effect Engine - REE

Like most hardware platforms, the N64 System supports a reverb engine to add some hall, echo or delay to the output signal. The architectures and possibilities vary greatly from platform to platform. Please refer to Appendix 5 for details on the implementation.

Volume Control

MusyX uses an elaborate scheme to control the volume of both music and sound effects. Besides all local volume control, either through MIDI velocities or direct specification of a volume to be used for sound effect playback, there are a large number of so called "volume groups" which take care of the master volume control.

During the development of *MusyX*, a limited number of master volumes (e.g. just separated for music and sound effects) proved to be far too limited in today's complex game environments. By default just one volume group is defined. It is used to control the master volume for all sound effects. Each time a song is started, a new volume group that controls the master volume for that specific song is created.

In addition to this default behavior, the application programmer may define new volume groups for single sound effects or a set of sound effects, as well as defining new volume groups to control single tracks within a song separately from the rest of the song. Just think of it as a large mixer that can be completely software controlled.

Each volume group actually contains two faders. This is done so that crossfades can be performed, while the secondary fader is still available for overall volume control. (Crossfades are directly controlled by the main fader of each volume group. So, if this fader is manipulated during a crossfade, the crossfade may fail.)

Two master faders, scaling all sound effect volume groups and music volume groups respectively, are also implemented.

IDs

MusyX uses IDs to reference sound effects, songs and groups. These IDs are automatically generated during the data conversion process. To make them easily accessible from the programmer's side, a header file is generated automatically (either in C or assembler syntax) that contains symbolic constants for these IDs. Their names are generated using the names for these entities, given to them by the musician.

In this way, we ensure that both the musician and the programmer have a common basis to reference IDs – their names – and that changes in the project do not change anything on the programmer's side of things.

3D API

MusyX features a complete 3D API. It handles all parameters of SFXs that change over time in a 3D environment. Volume, panning – including surround panning – and Doppler effects are calculated in real time. The API does not feature any kind of culling mechanism. While SFXs that are not audible will not use any voices, they will still be calculated – in case they become audible again. The application has to take care of any form of scene culling to limit the amount of handled SFXs via the 3D API. It specifies SFX as "emitters". These emitters are structures that the application has to allocate space for, and that are made known to the system using `snd_add_emitter()`. The position and orientation of the "listeners" is specified using another structure made known to the system using `snd_add_listener()`. There may be more than one listener. Using more than one listener makes it impossible for the system to generate a Doppler effect, though. For a detailed description of the functions and structures see the following Function Section.

MusyX uses a standard right-handed coordinate system.

Function Section: SOUND

SND_INIT

```
int snd_init(ULONG playfrq, UBYTE voices,  
UBYTE music, UBYTE sfx, UWORLD maxdelay, ULONG flags);
```

Purpose:

This function initializes the sound system. It must be called once before any other routines of the system are used. This includes the interrupt handlers, if they are installed by the application program. Whether or not that is the case, depends upon the platform on which you are running. See Musician's Reference for details.

The system will come up initially, with no voices active and the main volume for music and FX down.

Input:

ULONG mixfrq

Specifies the frequency in Hz used to mix all voices. This frequency will be the actual playback frequency used with the sound hardware. Invalid values will be clipped. See each platform appendix for supported values with the various platforms. Be aware that some platforms impose limits on how far samples may be pitched up or down. Some playback frequencies may be incompatible with the music and or sound effects that you try to play back. See Musician's Reference for details.

UBYTE voices

Specifies the maximum number of voices to be mixed at a time. The maximum number of voices allowed varies from platform to platform. See Musician's Reference for details.

UBYTE music

Specifies the maximum number of voices, of the total amount of voices, that can be used by the synthesizer to playback instruments.

UBYTE sfx

Specifies the maximum number of voices, of the total amount of voices, that can be used by the synthesizer to playback sound effects.

UWORD maxdelay

Maximum delay size of the reverb engine. On most platforms, the delay buffer will be allocated before the actual game is started up, to avoid memory fragmentation. The reverb processing is not activated, even if values different from zero are specified.

ULONG flags

These flags are used to trigger specific behaviors of the sound system. Most flags are platform specific and can be found in the Musician's Reference. A few, however, are platform independent.

SND_FLAGS_DEFAULT

Default settings are used.

SND_FLAGS_STEREOONLY

Any surround processing or multi channel output is disabled. This may save calculation time on some platforms. Some platforms may ignore this flag.

SND_FLAGS_NOINTERPOLATION

Any interpolation will be switched off. This may save calculation time on some platforms. Some platforms may ignore this flag.

Output:

The function returns a value of 0, if successful.

SND_QUIT

```
void snd_quit(void)
```

Purpose:

This function exits the sound system and frees all allocated resources. IRQ handlers should not be called after this function has been executed, if any were manually installed.

Input:

None.

Output:

None.

SND_SHUTDOWN

```
void snd_shutdown(void)
```

Purpose:

This function exits the sound system in preparation for a reinitialization of the system. IRQ handlers should not be called after this function has been executed.

Input:

None.

Output:

None.

SND_GET_PLAYBACKINFO

```
SND_PLAYBACKINFO( * SNDCALL) snd_get_playbackinfo(void)
```

Purpose:

This function returns information about the version of the sound system.

Input:

None.

Output:

A pointer to the following initialized structure is returned.

```
typedef struct _snd_playbackinfo {
    ULONG frq;           // frequency used to
                          // output audio
    BYTE stereo;         // TRUE if output is
                          // stereo
    UBYTE bits;          // Number of bits per
                          // sample
    char devname[256];   // ASCII name of device
    char version_text[256]; // ASCII driver name &
                           // version
} SND_PLAYBACKINFO;
```

SND_REINIT

```
reinit SNDCALL snd_reinit(ULONG playfrq,  
UBYTE voices, UBYTE music, UBYTE sfx,  
UWORD maxdelay, ULONG flags);
```

Purpose:

This function reinitializes the sound system. It must be called once before any other routines of the system are used after a temporary shutdown. This includes the interrupt handlers, if they are installed by the application program. Whether that is the case, depends on the platform you are running on. See Musician's Reference for details.

The system will initialize with no voices active and the main volume for music and FX down.

Input:

ULONG mixfrq

Specifies the frequency in Hz used to mix all voices. This frequency will be the actual playback frequency used with the sound hardware. Invalid values will be clipped. See each platform appendix for supported values with the various platforms. Be aware that some platforms impose limits on how far samples may be pitched up or down. Some playback frequencies may be incompatible with the music and or sound effects that you try to play back. See Musician's Reference for details.

UBYTE voices

Specifies the maximum number of voices to be mixed at a time. The maximum number of voices allowed varies from platform to platform. See Musician's Reference for details.

UBYTE music

Specifies the maximum number of voices, of the total amount of voices, that can be used by the synthesizer to playback instruments.

UBYTE sfx

Specifies the maximum number of voices, of the total amount of voices, that can be used by the synthesizer to playback sound effects.

UWORD maxdelay

Maximum delay size of the reverb engine. On most platforms, the delay buffer will be allocated before the actual game is started up, to avoid memory fragmentation. The reverb processing is not activated, even if values different from zero are specified.

ULONG flags

These flags are used to trigger specific behaviors of the sound system. Most flags are platform specific and can be found in the Musician's Reference. A few, however, are platform independent.

SND_FLAGS_DEFAULT

Default settings are used.

SND_FLAGS_STEREOONLY

Any surround processing or multi channel output is disabled. This may save calculation time on some platforms. Some platforms may ignore this flag.

SND_FLAGS_NOINTERPOLATION

Any interpolation will be switched off. This may save calculation time on some platforms. Some platforms may ignore this flag.

Output:

None.

SND_VOLUME

```
void snd_volume(UBYTE volume, UWORLD time,  
UBYTE volgroup)
```

Purpose:

This function sets the current volume for any volume groups. The volume may be set at once or may be faded to the new setting. A fade does not need to be finished before a new one can be started.

Input:

UBYTE volume

Specifies the volume to be used for set / fade. 0=Silence, 127=100%.

UWORD time

Specifies the time in ms to fade to the new volume. If zero is specified, the volume will be set immediately. This may result in clicking sounds.

UBYTE volgroup

Specifies which volume group is to be changed. You may also specify one of the constants below to modify a set of volume groups at one time, while only using only one call.

SND_USERMUSIC_VOLGROUPS

SND_USERFX_VOLGROUPS

SND_USERALL_VOLGROUPS

Set a new volume for all music, sfx or both volume groups, defined by the user.

SND_MUSIC_VOLGROUPSS

SND_FX_VOLGROUPS

SND_ALL_VOLGROUPS

Set a new volume for all music, sfx or both volume groups, predefined or defined by the user.

Output:

None.

SND_MASTER_VOLUME

```
void snd_master_volume(UBYTE volume, UWORLD time,  
UBYTE music, UBYTE sfx)
```

Purpose:

This function sets the current volume for the master faders of music or sfx. The volume may be set at once or may be faded to the new setting. A fade does not need to be finished before a new one can be started.

Input:

UBYTE volume

Specifies the volume to be used for set / fade. 0=Silence, 127=100%.

UWORD time

Specifies the time in ms to fade to the new volume. If zero is specified, the volume will be set immediately. This may result in clicking sounds.

UBYTE music

Set to TRUE if music master fader should be affected.

UBYTE sfx

Set to TRUE if sfx master fader should be affected.

Output:

None.

SND_MONO

```
void snd_mono(UBYTE mono)
```

Purpose:

Set the system to mono or stereo mode.

Input:

UBYTE mono

If TRUE, all output will be mono. FALSE selects stereo mode. The default mode is FALSE.

Output:

None.

SND_PLAY

```
ULONG snd_play(UWORD sgid, UWORD sid,  
void *arrfile, SND_PLAYPARA *para)
```

Purpose:

This function starts the playback of a song contained within a song group.

Input:

UWORD sgid

Specifies the ID of the song group to be used as a source.

UWORD sid

ID that specifies which sequencer file in the specified song group is to be used.

*ULONG *arrfile*

Pointer to memory containing sequencer data. Certain alignment requirements may apply for certain platforms.

*SND_PLAYPARA *para*

Pointer to a structure containing various parameters for starting the song. This pointer may be set to NULL. In this case, the song will be started immediately, and no volume will be set (by default the volume is down).

```

typedef struct _snd_playpara {
    ULONG      flags;           // Enable features by
                                // using these flags
    ULONG      trackmute[2];   // Initial mute settings
    UWORLD     speed;          // Initial speed factor
                                // (0x100 = 1:1)
    struct {
        // Start volume
        // information
        UWORLD      time;
        UBYTE       target;
    } volume;
    UBYTE      num_seqvoldef; // Number of non-
                                // standard volume group
                                // tracks
    SND_SEQVOLDEF *seqvoldef; // List of tracks and
                                // volume groups
    UBYTE      num_faded;     // Number of entries to
                                // the fade list
    UBYTE      *faded;
} SND_PLAYPARA;

```

The *flags* field defines which subset of parameters is active at any given time. The following values are defined.

SND_PLAYPARA_DEFAULT

No parameters are valid.

SND_PLAYPARA_TRACKMUTE

The *trackmute* fields are active. Each cleared bit in these two ULONGs defines a track to be muted.

SND_PLAYPARA_SPEED

The *speed* field is active. A value of 0x100 selects normal speed. Lower values slow the song down. Higher levels speed it up.

SND_PLAYPARA_VOLUME

This flag turns on the basic volume control. The *time* and *target* values become active. *Target* specifies the volume level at the end of the fade, while *time* specifies the time in milliseconds that fade should take. A value of zero will simply set the new volume immediately.

SND_PLAYPARA_SEQVOLDEF

Each track in an arrangement may be controlled by its very own volume group. This flag enables one to control this feature in detail. If the flag is set, the following fields have to be initialized.

Num_seqvoldef has to be initialized with the number of tracks to which the non default settings should be applied. The seqvoldef field is a pointer to an array of SND_SEQVOLDEF structures that define all parameters for the specific tracks.

```
typedef struct _snd_seqvoldef {  
    UBYTE track;      // Target track (0-63)  
    UBYTE volgroup;   // Volume group to  
                      assign / use  
} SND_SEQVOLDEF;
```

Num_faded contains the number of tracks that should be faded. One may enable or disable fading for each track, to enable the programmer to fade in and out separate tracks without having to setup all volume groups manually. The *faded* field contains a pointer to a list of track numbers.

SND_PLAYPARA_PAUSE

The song will be "started" in pause mode. No audio from the song will be audible until you restart it.

Output:

The function returns a 32-bit sequencer ID if successful, if not it returns SND_SEQ_ERROR_ID .

SND_STOP

```
void snd_stop(ULONG seqid)
```

Purpose:

Stops the song currently playing with the specified ID. If the song is not playing anymore, nothing happens.

Input:

ULONG seqid

ID that specifies which sequencer is to be stopped.

Output:

None.

SND_PAUSE

```
void snd_pause(ULONG seqid)
```

Purpose:

Pauses the song currently playing with the specified ID. If the song is not playing anymore, nothing happens.

Input:

ULONG seqid

ID that specifies which sequencer is to be paused.

Output:

None.

SND_SILENCE

```
void snd_silence(void)
```

Purpose:

All voices (Music & FX) will be stopped immediately.

Input:

None.

Output:

None.

SND_IS_IDLE

```
UBYTE snd_is_idle (void)
```

Purpose:

Checks to see if all activity in the sound system has ended.

Input:

None.

Output:

The function returns TRUE if there is no activity left in the sound system and it's ready to be shut down. It returns FALSE otherwise.

SND_CROSSFADE

```
void snd_crossfade(SND_CROSSFADE *ci,  
ULONG *new_seq_id);
```

Purpose:

Initiates a crossfade between two pieces of music.

Input:

*SND_CROSSFADE *ci*

Pointer to a structure containing all necessary information to perform the crossfade.

```
typedef struct _snd_crossfade {  
    ULONG      seq_id1;  
    WORD       time1;  
  
    ULONG      seq_id2;  
    WORD       time2;  
    void      *arr2;  
    WORD       gid2;  
    WORD       sid2;  
    BYTE       vol2;  
  
    ULONG      trackmute2[2]; // Mute bits (see  
                           // snd_mute())  
    WORD      speed2;        // Initial speed  
                           // (of new song)  
  
    BYTE      flags;  
} SND_CROSSFADE;
```

The song specified by the ID passed in *seq_id1* will be faded down in the time specified by *time1*. *Flags* defines what is to be done to it at this moment.

SND_CROSSFADE_STOP

Stop song after fade down is finished.

SND_CROSSFADE_PAUSE

Pause song after fade down is finished.

SND_CROSSFADE_MUTE

Mute song after fade down (no voices will be allocated), but continue handling (playing) it.

A couple of other flags define how the new song is to be faded up.

SND_CROSSFADE_CONTINUE

The new song is in paused state. Continue playback from that position. The ID of that song must be specified in *seq_id2*, which is otherwise unused.

SND_CROSSFADE_PAUSENEW

The new song will be started, but immediately put into paused state.

SND_CROSSFADE_TRACKMUTE

The *trackmute* fields will be used to setup the track muting of the new song.

SND_CROSSFADE_SPEED

The new song will be started with a playback speed specified in the *speed2* field.

SND_CROSSFADE_MUTENEW

Start new song, but mute it completely, immediately after its start.

In all cases where a completely new song is started the *arr2* field must be initialized with a pointer to the arrangement data. *Gid* must contain the ID of the group which contains the settings for the new song. *Sid* contains the ID of the song itself.

The volume at end of fade in the time *time2* is always to be specified in *vol2*.

Crossfades can be synchronized to certain points in a song. The musician can place special controllers into the MIDI data representing the song. If the flag,

SND_CROSSFADE_SYNC

is set, the crossfade will be delayed until the next controller of that type is detected.

Crossfades are controlled directly by the faders. If you reuse a fader while a crossfade is in progress, the crossfade may not succeed. This should rarely be necessary, though. On the one hand the system's master faders can still be used and, on the other hand, each volume group has two faders. A standard fader and a so-called "pause" fader. The later can be freely used, even if a crossfade is in progress.

*ULONG *new_seq_id*

Pointer to a ULONG that will be assigned the new sequencer ID as soon as it is available. A temporary ID will be assigned immediately, while the real ID will be assigned at some point during the crossfade. This variable must be present during the crossfade for that reason. The temporary ID can be used for most control purposes, as long as no real ID is generated.

Output:

None.

SND_CROSSFADE_DONE

UBYTE snd_crossfade_done (ULONG seqid)

Purpose:

Check if a crossfade has finished.

Input:

ULONG seqid

ID that specifies which sequencer is to be checked. The ID specified is the one of the new song, not the old one.

Output:

The function returns TRUE if the crossfade is complete, otherwise FALSE is returned.

SND_CONTINUE

```
void snd_continue(ULONG seqid)
```

Purpose:

Resumes playback of a paused song.

Input:

ULONG seq_id

ID of the song / sequencer to be continued.

Output:

None.

SND_MUTE

```
void snd_mute(ULONG seqid, ULONG mask1, ULONG mask2)
```

Purpose:

This function mutes / turns on volume for single tracks from the sequencer file. The 64 tracks are represented by 32 Bits in 2 ULONGs.

Input:

ULONG seqid

ID of sequencer to be affected.

ULONG mask1

Mute mask for lower 32 tracks. A 0 mutes the corresponding channel.

ULONG mask2

Mute mask for upper 32 tracks. A 0 mutes the corresponding channel.

Output:

None.

SND_SPEED

```
void snd_mute(ULONG seqid, UWORLD speed)
```

Purpose:

This function changes the playback speed of the specified sequencer.

Input:

ULONG seqid

ID of sequencer to be affected.

UWORD speed

Speed to be set. A value of 0x100 will reset the speed to normal. Lower values will slow down the playback, higher values will speed it up.

Output:

None.

SND_SEQLOOP

```
void snd_seqloop(ULONG seqid, UBYTE on)
```

Purpose:

This function allows the looping of songs to be disabled. By default looping is enabled, if the musician defined loop locators. Disabling looping may be useful if e.g. a song is meant to loop a couple of times before finally coming to an end.

Input:

ULONG seqid

ID of sequencer to be affected.

UBYTE on

A value of TRUE will enable looping (default), FALSE will disable it.

Output:

The function will return TRUE on success, FALSE otherwise.

SND_GET_SEQLOOPCNT

```
UWORD snd_get_seqloopcnt (ULONG seqid)
```

Purpose:

This function retrieves how often the playing song has looped.

Input:

ULONG seqid

ID of sequencer to report.

Output:

Returns the number of loops taken by the song.

SND_GET_SEQVALID

```
UBYTE snd_get_seqvalid (ULONG seqid)
```

Purpose:

This function is used to determine if the song is actually playing or still in a temporary state. Songs may be in a temporary state immediately after or during crossfades

Input:

ULONG seqid
ID of song / sequencer to report.

Output:

Returns TRUE if the sequencer actually is playing the song specified by the ID. FALSE is returned if the ID is temporary.

SND_SEQ_VOLUME

```
void snd_seq_volume(UBYTE volume, UWORLD time, ULONG  
seq_id, UBYTE mode);
```

Purpose:

This function is used to set / fade the volume of the specified sequencer / song.

Input:

UBYTE volume

Specifies the volume to be used for set / fade. 0=Silence, 127=100%.

UWORD time

Specifies the time in ms to fade to the new volume. If zero is specified, the volume will be set immediately. This may result in clicking sounds.

ULONG seqid

ID of sequencer to be affected.

UBYTE mode

Mode of fade operation. The following modes are defined:

SND_SEQVOL_CONTINUE

Continue playback of the current song when fade is finished.

SND_SEQVOL_STOP

Stop current song when fade is finished.

SND_SEQVOL_PAUSE

Pause song when fade is finished.

SND_SEQVOL_MUTE

Mute all tracks of the current song when fade ends. The song will continue playing although muted.

Output:

None.

SND_GET_SEQVOLGROUP

UBYTE snd_get_seqvolgroup (ULONG seqid)

Purpose:

This function is used to determine the volume group used to control the referenced song / sequencer. If non default groups have been assigned by the user, the default one will be returned.

Input:

ULONG seqid
ID of song / sequencer to report.

Output:

The volume group used to control the default song / sequencer volume.

SND_ASSIGN_VGROUP2TRACK

```
void snd_assign_vgroup2track (ULONG seq_id,  
    UBYTE track, UBYTE vgroup);
```

Purpose:

Assign a non standard volume group to the specified track of the given song / sequencer.

Input:

ULONG seqid

ID of song / sequencer to affect.

UBYTE track

Track to be affected.

UBYTE vgroup

Volume group to assign to track.

Output:

None.

SND_FXSTART

```
ULONG snd_fxstart(UWORD fid, UBYTE vol, UBYTE pan)
```

Purpose:

Starts a sound effect.

Input:

UWORD fid

Specifies the ID of the sound effect to be used.

UBYTE vol

Specifies the default volume to be used with this sound effect. This value may or may not be used by the sound effect. (0=Silence, 127=100%, 0xFF=Use default volume)

UBYTE pan

Specifies the default panning to be used with this sound effect. This value may or may not be used by the sound effect. (0=Left, 64=Center, 127=Right, 128=Surround, 0xFF=Use default panning)

Output:

Returns SND_ID_ERROR if failed, a 32-bit voice ID if successful.

SND_FXKEYOFF

```
int snd_fxkeyoff(ULONG vid)
```

Purpose:

Send a "KeyOff" to the sound effect with the specified voice ID. A "Key off" is used to signal the sound effect to go into its final phase. In most cases it will be used to stop the sound effect.

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

Output:

Returns 0 if successful.

SND_FXCHECK

```
UBYTE snd_fxcheck(ULONG vid)
```

Purpose:

Test if the given sound effect is currently processed by the sound system.

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

Output:

Returns the ID passed to it if sound effect is currently active, `SND_ID_ERROR` otherwise.

SND_FXPANNING

```
int snd_fxpanning(ULONG vid, UBYTE pan)
```

Purpose:

Set new panning offset for the specified sound effect. The default panning offset after start is 64. (0=Left, 64=Center, 127=Right, 128=Surround) A value of 128 should no longer be used. It's still supported for now to maintain compatibility with older versions.

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

UBYTE pan

The new panning value to be applied to the effect.

Output:

Returns 0 if successful.

SND_FXSURROUNDPANNING

```
int snd_fxsurroundpanning(ULONG vid, UBYTE pan)
```

Purpose:

Set new surround panning offset for the specified sound effect. The default panning offset after start is 0. (0=Front, 64=Center, 127=Back)

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

UBYTE pan

The new surround panning value to be applied to the effect.

Output:

Returns 0 if successful.

SND_FXVOLUME

```
int snd_fxvolume(ULONG vid, UBYTE vol)
```

Purpose:

Set new volume for the specified soundeffect. The default volume set after start is 127. (0=Silence, 127=100%)

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

UBYTE vol

The new volume value to be applied to the effect.

Output:

Returns 0 if successful.

SND_FXPITCHBEND

```
int snd_fxpitchbend(ULONG vid, UWORLD pb)
```

Purpose:

Set new pitchbend value for the specified sound effect. The default pitchbend value set after start is \$2000. (\$2000=No Pitch offset, lower values pitch down, higher up).

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

UWORD pb

The new pitchbend value to be applied to the effect.

Output:

Returns 0 if successful.

SND_FXMODULATION

```
int snd_fxmodulation(ULONG vid, UWORLD mod)
```

Purpose:

Set new modulation value for the specified sound effect. The default modulation value set after start is 0. (0=Lowest, 16383=Highest)

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

UWORD mod

The new modulation value to be applied to the effect.

Output:

Returns 0 if successful.

SND_FXPEDAL

```
int snd_fxp pedal(ULONG vid, UBYTE pedal)
```

Purpose:

Set new pedal state for the specified sound effect.

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

UBYTE pedal

The new pedal state. If the pedal controller is still mapped to the pedal input of the synthesizer (see Musicians Reference), a value lower than 0x3F will clear the pedal state, while a larger value will set it.

Output:

Returns 0 if successful.

SND_FXDOPPLER

```
int snd_fxpitchbend(ULONG vid, UWORLD doppler)
```

Purpose:

Set new Doppler value for the specified sound effect. The default Doppler value set after start is \$2000. (\$2000=No Pitch offset, lower values pitch down, higher up). In contrast to the pitchbend, the Doppler effect is applied by scaling the frequency, rather than offsetting the pitch by offsetting the current key.

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

UWORD doppler

The new Doppler value to be applied to the effect.

Output:

Returns 0 if successful.

SND_FXREVERB

```
int snd_fxreverb(ULONG vid, UBYTE rvol)
```

Purpose:

Set new reverb volume for the specified sound effect.

Input:

ULONG vid

The voice ID obtained through `snd_fxstart()`.

UBYTE rvol

The new reverb volume to set.

Output:

Returns 0 if successful.

SND_PUSHGROUP

```
UBYTE snd_pushgroup(void *prj_data, UWORLD gid,  
void *samples, void *sampdir, void *pool);
```

Purpose:

Push group data onto soundstack. See general information section for details.

Input:

*void *prjdata*

Pointer to project data.

UWORD gid

ID of the group to be pushed.

*void *samples*

Pointer to sample data / ID to reference data.

*void *sampdir*

Pointer to sample directory containing data to locate samples within the sample data.

*void *pool*

Pointer to pool data.

Output:

Returns TRUE if successful.

SND_POPGROUP

```
UBYTE snd_popgroup(void)
```

Purpose:

Pop group from soundstack. See general information section for details.

Input:

None.

Output:

Returns TRUE if successful.

SND_READFLAG

```
SWORD snd_readflag(UBYTE num)
```

Purpose:

Read flag value from synthesizer. Flags may be used to signal certain conditions of the synthesizer to the main program. There are 16 flags.

Input:

UBYTE num

Number of flag to read from.

Output:

Returns value obtained from flag. Default after startup is zero.

SND_WRITEFLAG

```
void snd_writeflag(UBYTE num, SWORD value)
```

Purpose:

Writes flag value to synthesizer. Flags may be used to signal certain conditions of the synthesizer to the main program. There are 16 flags.

Input:

UBYTE num

Number of flag to write to.

SWORD value

Value to write to specified flag.

Output:

None.

SND_ALLOC_STREAM

```
ULONG snd_alloc_stream(UBYTE prio, void *buffer,
ULONG size, ULONG frq, UBYTE vol, UBYTE pan,
UBYTE span, UBYTE fxvol,
UBYTE (*update_function)(SWORD *buffer1,
ULONG len1, SWORD *buffer2, ULONG len2,
ULONG user), ULONG user)
```

Purpose:

Allocates a voice for stream playback. The allocated voice will no longer be available to the synthesizer for playing instruments or sound effects until it's explicitly freed.

Input:

UBYTE prio

Priority for allocating the voice to be used for playing the stream. To guarantee allocation, the priority should be set as high as possible.

*void *buffer*

Pointer to buffer used to stream the data into the voice. The buffer is used as a simple ring buffer.

ULONG size

Size of the ring buffer in samples.

ULONG frq

Frequency to be used for playback.

UBYTE vol

Volume used for playback (0-127).

UBYTE pan

Panning used for playback (0-127).

UBYTE span

Surround panning used for playback (0-127).

UBYTE fxvol

Reverb volume used for playback (0-127).

*UBYTE (*update_function)(SWORD *buffer1,ULONG len1,SWORD *buffer2,ULONG len2,ULONG user)*

Pointer to a function that will be called by *MusyX* each time that the buffer has to be updated. *Buffer1* and *len1* define the 1st area to be updated, while *buffer2* and *len2* define the 2nd area. Two areas are necessary, since the buffer is used as a ring buffer, and the area to be updated may therefore wrap around. *Len2* will be zero if no second area is needed. *User* is a value specified by the application, see below).

ULONG user

Value to be passed to the update function. E.g. it can be used to identify different instances of streams to the update function.

Output:

The function returns a handle to the stream if successful and *SND_ID_ERROR* if not.

SND_STREAM_ALLOCLLENGTH

ULONG snd_stream_alloclength(ULONG num)

Purpose:

Return the number of bytes to be allocated for a stream buffer, containing the specified number of samples. This may or may not be identical to the number of samples multiplied by the number of bytes per sample. The reason for this is that some platforms need a couple of extra samples that are managed automatically to ensure proper looping.

Input:

ULONG num

Number of samples in the buffer.

Output:

Size of the buffer that has to be allocated in bytes.

SND_STREAM_MIXPARAMETER

```
void snd_stream_mixparameter(ULONG stid,  
    UBYTE vol, UBYTE pan, UBYTE span, UBYTE fxvol)
```

Purpose:

Change stream playback parameters during playback.

Input:

ULONG stid

Handle of stream to be manipulated.

UBYTE vol

New volume to be set.

UBYTE pan

New panning to be set.

UBYTE span

New surround panning to be set.

UBYTE fxvol

New reverb volume to be set.

Output:

None.

SND_FREE_STREAM

```
void snd_free_stream(ULONG stid)
```

Purpose:

Free an allocated stream. The voice will be made accessible to the synthesizer, again.

Input:

ULONG stid

Handle of stream to be freed.

Output:

None.

SND_ACTIVATE_REVERB

```
void snd_activate_reverb(SND_REVERB *rev);
```

Purpose:

Activate reverb engine with the given parameters. Only signals on the reverb or "auxiliary" bus are put through the engine.

Input:

SND_REVERB *rev

Pointer to a structure describing the desired reverb effect.

```
typedef struct _snd_reverb {
    UWORLD buf_size;
    UWORLD ref_off[SND_MAX_REFLECTIONS];
    UBYTE ref_vol[SND_MAX_REFLECTIONS];
    UBYTE ref_num;
    UBYTE fb_vol;
    UBYTE filter;
    SWORD filter_coeff[4];
} SND_REVERB;
```

Buf_size specifies the total size of the reverb delay buffer in milliseconds. This size must be smaller or equal to the maximum size specified at system initialization.

Ref_num must be set to the number of reflections to be used. The maximum number of reflections in current implementations is 8.

Fb_vol specifies the volume of the feedback into the buffer. The buffer output is continuously fed back into the buffer together with new input. If this volume is specified too high, there is the danger of a so-called "resonant catastrophe".

A FIR filter may be used to filter the output signal. *Filter* can be set to *SND_FILTER_ON* to enable the filter or *SND_FILTER_OFF* to disable it. If it is enabled, one must specify the filter coefficients to be used. These values are passed in the *filter_coeff*[] array. The values are specified as integers in S15 format. A value of 0xFFFF equals about 1.0.

Reflections are specified using two arrays. Each pair of array elements specifies one reflection. The *ref_off*[] array contains the amount of delay for each reflection in milliseconds. The *ref_vol*[] array contains the volume of each reflection. A value of 127 specifies full volume. A volume of zero is possible, but should be avoided for performance reasons.

Generally one can say, that the more reflections used and the more these reflections are spread out over the buffer, the bigger the performance hit will be.

Output:

None.

SND_DEACTIVATE_REVERB

```
void snd_deactivate_reverb(void);
```

Purpose:

Stop reverb processing.

Input:

None.

Output:

None.

SND_ADD_LISTENER

```
void snd_add_listener(SND_LISTENER *li,
                      SND_FVECTOR *pos, SND_FVECTOR *dir,
                      SND_FVECTOR *heading, SND_FVECTOR *up,
                      float front_sur, float back_sur,
                      float soundspeed, ULONG flags, UBYTE vol);
```

Purpose:

Add a listener structure to the list of listeners.

Input:

*SND_LISTENER *li*

Pointer to a structure defining the parameters of the listener to be added.

```
typedef struct _snd_listener {
    struct _snd_listener    *next;
    struct _snd_listener    *prev;

    ULONG                  flags;
    SND_FVECTOR             pos;
    SND_FVECTOR             dir;
    SND_FVECTOR             heading;
    SND_FVECTOR             right;
    SND_FVECTOR             down;
    SND_FMATRIX              mat;
    float                  surround_dis_front;
    float                  surround_dis_back;
    float                  soundspeed;
    float                  vol;
} SND_LISTENER;
```

All values are initialized by the called function. The structure is not copied by the function. It has to be kept around as long as the listener is active.

*SND_FVECTOR *pos*

Pointer to a vector containing the initial position of the listener.

*SND_FVECTOR *dir*

Pointer to a vector containing the initial movement direction of the listener.

*SND_FVECTOR *heading*

Pointer to a vector containing the initial normalized viewing direction of the listener.

*SND_FVECTOR *up*

Pointer to a vector containing the initial normalized up vector of the listener.

Float front_sur

Distance at which an emitter will only be audible on the front channels.

Float back_sur

Distance at which an emitter will only be audible on the surround channel.

ULONG flags

Flags contains one or more of the following flags:

SND_LISTENER_DEFAULT

No Doppler effect is calculated. The speed of sound does not need to be set, anymore. This mode should be chosen if more than one listener is specified.

SND_LISTENER_DOPPLERFX

Doppler effects are calculated.

Float soundspeed

Defines the speed of sound for use in Doppler effects. It's specified in "units" per second, and will depend on the game's world coordinate system. Float soundspeed must be set to "less than" the speed of sound (in game world coordinates).

UBYTE vol

Overall volume value to be applied to all emitters that are audible to this listener.

Output:

None.

SND_UPDATE_LISTENER

```
void snd_update_listener(SND_LISTENER *li,
SND_FVECTOR *pos, SND_FVECTOR *dir,
SND_FVECTOR *heading, SND_FVECTOR *up, UBYTE vol);
```

Purpose:

Update a listener structure.

Input:

*SND_LISTENER *li*

Pointer to a structure containing the current settings of the listener.

*SND_FVECTOR *pos*

Pointer to a vector containing the new position of the listener.

*SND_FVECTOR *dir*

Pointer to a vector containing the new movement direction of the listener.

*SND_FVECTOR *heading*

Pointer to a vector containing the new normalized viewing direction of the listener.

*SND_FVECTOR *up*

Pointer to a vector containing the new normalized up vector of the listener.

UBYTE vol

Overall volume value to be applied to all emitters that are audible to this listener.

Output:

None.

SND_REMOVE_LISTENER

```
void snd_remove_listener(SND_LISTENER *li);
```

Purpose:

Remove a listener structure from the sound systems list. The structure may be discarded after calling this function.

Input:

*SND_LISTENER *li*

Pointer to a structure containing the current settings of the listener.

Output:

None.

SND_ADD_EMITTEREX

```
ULONG add_emitterex(SND_EMITTER *em,
SND_FVECTOR *pos, SND_FVECTOR *dir,
float max_dis, float comp, ULONG flags,
UWORD fxid, UWORD groupid, UBYTE maxvol,
UBYTE minvol);
```

Purpose:

Add an emitter structure to the list of emitters.

Input:

*SND_EMITTER *em*

Pointer to a structure defining the parameters of the emitter to be added.

```
typedef struct _snd_emitter {
    struct _snd_emitter    *next;
    struct _snd_emitter    *prev;

    ULONG                  flags;
    SND_FVECTOR            pos;
    SND_FVECTOR            dir;
    float                  max_dis;
    float                  maxvol;
    float                  minvol;
    float                  volpush;
    ULONG                  vid;
    ULONG                  group;
    UWORD                  fxid;

    UWORD                  vollevel_cnt;
    float                  fade;
} SND_EMITTER;
```

All values are initialized by the called function. The structure is not copied by the function. It has to be kept around as long as the emitter is active.

*SND_FVECTOR *pos*

Pointer to a vector containing the initial position of the emitter.

*SND_FVECTOR *dir*

Pointer to a vector containing the initial movement direction of the emitter.

float max_dis

Maximum distance at which the emitter will still be audible.

float comp

Normally the volume will be lowered linearly over distance. At times it may be desirable to keep the volume up for some time and later fade it down faster. **MusyX** offers both methods. This value allows fading between both extremes. A value of zero will use only the linear method. Higher values will switch more and more toward the “compressed” form. A value of 1 uses only the “compressed” method.

ULONG flags

The following flags are defined to influence the behavior of the emitter.

SND_EMITTER_DEFAULT

No special features are activated.

SND_EMITTER_CONTINOUS

Update all parameters continuously. If this is not set, the volume, panning and Doppler values will just be calculated at the start of the emitter. This may be used to save calculation time with short time SFX.

SND_EMITTER_RESTARTABLE

The SFX handled by the emitter may be restarted after being stopped for any reason. This can be used to reactivate e.g. an engine hum after an explosion has interrupted it for a short time.

SND_EMITTER_PAUSABLE

If the emitter is no longer in the audible range it may be stopped. If this flag is not set, the SFX will remain active although its volume is all the way down. This costs valuable voices, but may be necessary to run long term sound effects.

SND_EMITTER_DOPPLERFX

If this flag is set, the emitter will be included into the Doppler effect calculations of all listeners having Doppler calculations enabled.

SND_EMITTER_HARDSTART

Continuos emitters are by default faded in over a short period of time. This is done to avoid any popping artifact and similar effects. This flag disables the default effect. The feature is currently not implemented.

UWORD fxid

ID of the SFX to be used by the emitter.

UWORD groupid

A value used to group emitters together. Emitters that share the same group are all fighting for the available voices on the basis of their volume and time being audible. The group ID specified here has nothing to do with the groups used to store data. It is simply an integer between 0 and 65535 that is chosen by the game application to identify such an emitter group.

Emitters of different groups just use the normal priority system.

UBYTE maxvol

Volume to be used at maximum audible range.

UBYTE minvol

Volume to be used at the position of the listener.

Output:

The function returns a handle to the SFX started by the emitter. This value may be SND_ID_ERROR without any failure. Continuous emitters are not immediately started sometimes.

SND_ADD_EMITTER

```
ULONG snd_add_emitter(SND_EMITTER *em,
                      SND_FVECTOR *pos, SND_FVECTOR *dir,
                      float max_dis, float comp, ULONG flags,
                      WORD fxid, UBYTE maxvol, UBYTE minvol);
```

Purpose:

Add an emitter structure to the list of emitters.

Input:

All parameters are the same as the ones listed for SND_ADD_EMITTEREX. The *groupid* parameter of this function is internally set, so that each SFX forms its own emitter group.

Output:

The function returns a handle to the SFX started by the emitter. This value may be SND_ID_ERROR without any failure. Sometimes continuous emitters are not started immediately.

SND_UPDATE_EMITTER

```
UBYTE snd_update_emitter(SND_EMITTER *em,  
SND_FVECTOR *pos, SND_FVECTOR *dir, UBYTE maxvol);
```

Purpose:

Update an emitter structure.

Input:

*SND_EMITTER *em*

Pointer to a structure containing the current settings of the emitter.

*SND_FVECTOR *pos*

Pointer to a vector containing the new position of the emitter.

*SND_FVECTOR *dir*

Pointer to a vector containing the new movement direction of the emitter.

UBYTE maxvol

Volume to be used at position of listener for this emitter.

Output:

Returns TRUE if successful, FALSE otherwise.

SND_REMOVE_EMITTER

```
UBYTE snd_remove_emitter(SND_EMITTER *em);
```

Purpose:

Remove an emitter structure from the sound systems list. The structure may be discarded after calling this function.

Input:

*SND_EMITTER *em*

Pointer to a structure containing the current settings of the emitter.

Output:

Returns TRUE if successful, FALSE otherwise.

SND_CHECK_EMITTER

```
UBYTE snd_check_emitter(SND_EMITTER *em);
```

Purpose:

Check if the specified emitter is currently active or not.

Input:

*SND_EMITTER *em*

Pointer to a structure containing the current settings of the emitter.

Output:

Returns TRUE if emitter is active, FALSE otherwise.

SND_EMITTER_FXID

```
ULONG snd_emitter_fxid( SND_EMITTER *em );
```

Purpose:

Get the handle of the SFX handled by the emitter specified.

Input:

*SND_EMITTER *em*

Pointer to a structure containing the current settings of the emitter.

Output:

If the SFX is active, the function will return its handle. If not SND_ID_ERROR will be returned. Even the latter value may be passed to all SFX functions without any negative side effects.

Function Section: VoiceLib MORT Interface

VOICE_INIT

```
void voice_init(ULONG flags);
```

Purpose:

This function initializes the voice library and MORT. It must be called once, before any other routines of the library are used. **MusyX** must be fully initialized before this function is called.

Input:

ULONG flags

These flags are used to trigger specific behaviors of the voice library.

VOICE_FLAGS_DEFAULT

Default settings are used. Buffers for streaming MORT data will be allocated at run-time (during the call to voice_start()) and will be freed at run-time. This, depending on the implementation of the memory allocation system, may lead to memory fragmentation.

VOICE_FLAGS_PREALLOCATE_BUFFERS

All buffers needed to stream MORT data are allocated during initialization and stay allocated until the library is shutdown using voice_exit().

Output:

None.

VOICE_EXIT

```
void voice_exit(void)
```

Purpose:

This function exits the voice library and frees all allocated resources.
Voices still active will be stopped.

Input:

None.

Output:

None.

VOICE_SET_DIRECTORY

```
UBYTE voice_set_directory(void *vdir, void *vdata)
```

Purpose:

This function tells the voice library where the MORT directory data file can be found.

Both *vdir* and *vdata* can be RAM addresses or ROM offsets. If *vdir* is a ROM offset, the function assumes that all data of the MORT directory is still in ROM. It will use the **MusyX** DMA services to download the MORT directory data table, without the actual compressed sample data section, into an allocated RAM area. (This may take as much as 2 frames, due to the nature of the **MusyX** DMA services.) In this configuration the second parameter is ignored and should be left at NULL.

If *vdir* specifies a RAM address, it's assumed that the MORT directory table is already downloaded into RAM. (It's always located at the very beginning of the data and its size is defined using the MORTDIR_DIRECTORY_LENGTH constant generated by MORTDIR.EXE.) Whether or not the actual MORT sample data is still in ROM is determined by examining the second parameter. If *vdata* specifies the same RAM location as *vdir*, it's assumed that all data is in RAM. If *vdata* specifies the "original" ROM offset of the whole MORT directory file, it's assumed that the table part has been downloaded to RAM but the samples are still in ROM.

Both addresses / offsets are passed through the **MusyX** address translation hook to enable the user to abstract data location via handles, if desired.

Input:

*void *vdir*

Pointer to the logical table part of the MORT directory file.

*void *vdata*

Pointer to the MORT directory file. (Used only if *vdir* specifies a RAM address. Keep it printing to NULL otherwise.)

Output:

The function returns TRUE if successful or FALSE otherwise.

VOICE_START

```
ULONG voice_start(ULONG dir_index, ULONG frq,  
UBYTE prio, UBYTE vol, UBYTE pan, UBYTE span,  
UBYTE fxvol)
```

Purpose:

This function starts a MORT compressed sample on a *MusyX* voice. The voice will be registered as an SFX voice. The current implementation allows two voices to be used simultaneously.

Input:

ULONG dir_index

Index of the MORT sample within the MORT directory file to be played. A header file containing these indices, in symbolic form, is generated by MORTDIR.EXE.

ULONG frq

Specifies the frequency to be used to playback the MORT sample. If you specify VOICE_DEFAULT_FRQ, the frequency stored within the MORT directory file will be used (only 8 or 16 kHz are supported). Sample rates above 22KHz may cause clicking artifacts, due to the limited size of the stream buffers.

UBYTE prio

Priority to be used to allocate the voice into which the data will be streamed. To “guarantee” a successful allocation, you should specify a priority of 255. Lower values mean a lower priority.

UBYTE vol

Volume to be used to play the MORT sample. (0-127, 127 = 100%)

UBYTE pan

Panning to be used to play the MORT sample. (0-127, left->right)

UBYTE span

Surround panning to be used to play the MORT sample.
(0-127, front->back)

UBYTE fxvol

FX (or AUX) Volume to be used to play the MORT sample (0-127, 127 = 100%). In the current implementation, this is the “reverb volume”.

Output:

The function returns a 32-bit handle if successful or VOICE_ERROR if not.

VOICE_STOP

```
UBYTE voice_stop(ULONG handle)
```

Purpose:

Stop a MORT sample which is playing.

Input:

ULONG handle

Handle of the MORT sample voice to be stopped.

Output:

Returns TRUE if successful, FALSE otherwise.

VOICE_CHECKACTIVE

```
UBYTE voice_checkactive(ULONG handle)
```

Purpose:

This function checks if the specified handle still refers to an active voice. The voice will be active a couple of frames before and after audio is audible.

Input:

ULONG handle

Specifies the voice to be checked.

Output:

The function returns TRUE if the voice is active and FALSE otherwise .

VOICE_PARAMETERS

```
UBYTE voice_parameters(ULONG handle, UBYTE vol,  
UBYTE pan, UBYTE span, UBYTE fxvol)
```

Purpose:

This function changes the mixing parameters for the specified voice.

Input:

ULONG handle

Handle that specifies which voice is to be influenced.

UBYTE vol

New volume to be used.

UBYTE pan

New panning to be used.

UBYTE span

New surround panning to be used.

UBYTE fxvol

New FX (AUX) volume to be used.

Output:

Returns TRUE if the parameters were set successfully, FALSE otherwise.

VOICE_GET_TIME

```
float voice_get_time(ULONG handle, float *total_time)
```

Purpose:

Returns information on both, total playing time and current time into playback of the MORT sample playing on the specified voice. This information can be used to synchronize other events to the MORT sample's playback (e.g. subtitles).

Input:

ULONG handle

Handle that specifies which voice's info structure is to be accessed.

*float *total_time*

Pointer to a float that will receive the total playing time of the MORT sample active on this voice, in seconds.

Output:

The function returns the current duration of the sample playback on the specified voice in seconds or zero if not successful.

VOICE_SYNC_IDLE

```
void voice_sync_idle(void)
```

Purpose:

Stalls until all activity in the voice library and MORT has ended. No voices will be stopped explicitly.

Input:

None.

Output:

None.

APPENDIX 4 – Game Boy Programmers Reference

Table of Contents:

<i>MusyX's Basic Architecture</i>	347
<i>Performance Impact on the Game Application</i>	348
<i>Memory Impact on the Game Application</i>	349
<i>Requirements for the Game Application</i>	350
<i>Linking MusyX Object Files to Your Application</i>	351
APPENDIX 4.1 – Game Boy Data Conversion Tools	353
What is Data Conversion?	353
MUConv.EXE.....	354
GM2Song.EXE	359
APPENDIX 4.2 - Function Section	360
snd_Init.....	361
snd_Exit.....	362
snd_Silence	363
snd_Handle	364
snd_StartSong	365
snd_StopSong	366
snd_PauseSong	367
snd_ResumeSong	368
snd_SongActive.....	369
snd_ChangeSongSpeed.....	370
snd_SetSongVolume	371
snd_GetStateSize	372
snd_SaveState	373
snd_RestoreState	374
snd_StartSFX	375
snd_StopSFX.....	376
snd_SetSFXVolume.....	377
snd_StartSample	378
snd_StopSample	379
snd_DoSample	380
snd_PlaySample.....	381
snd_CheckFlag.....	382
snd_SetMasterVolume.....	383

<i>Appendix 4.3 - Mini-MORT Samples.....</i>	384
--	-----

MusyX's Basic Architecture

In every sound system there is but one central element, the instrument or sound effect (depending upon whether you are talking music or effects). **MusyX** does not really differentiate between the two, so we will use the general term Sound throughout this manual.

In **MusyX** a sound actually represents a small, tokenized program that is executed at run time. This allows the sound designer more control over the produced sound.

As mentioned before there are two basic types of any kind of sound reproduction within **MusyX**. Instruments are used in the context of a piece of music, called a song. Each song has its own unique ID associated with it, which identifies the song when the programmer decides to start it.

Sound effects are also accessed using an automatically generated unique sound effect ID.

All IDs are hidden behind symbolic names so no changes to the program are required if an ID has changed.

All data, songs and sounds are collected into one larger project file, which needs to be included in the application.

Performance Impact on the Game Application

MusyX has been designed to offer a maximum of flexibility, at an acceptable CPU performance impact.

This impact varies depending on the features the musician utilizes in his or her small sound programs, since some features are slightly more complex than others.

It is also a question of whether or not it is possible to use **MusyX**'s built-in capabilities for sample playback in music. Samples for use in music come in two flavors, low and normal quality. Their difference is sampling rate. Low quality samples play back at 1920Hz and normal quality at 8192Hz. To make use of the normal quality, the game application must not make use of the timer interrupt. This interrupt will be used solely by **MusyX**, should you allow for samples of normal quality.

If you cannot spare the timer interrupt, the musician will need to resort to the low quality samples. These are still good enough for most drums.

You will need to coordinate this issue with the musician beforehand.

Finally, performance is naturally better when you write your game in dedicated Game Boy Color format, since it offers twice the speed of the conventional Game Boy.

Memory Impact on the Game Application

MusyX requires exactly **256 bytes** of memory for operation.

There is no requirement concerning placement or alignment, but for simplicity we have decided to let it take the last 256 bytes ranging from \$df00-\$dfff.

This will allow you to get the most out of the internal RAM on Game Boy Color, since it is permissible to have the RAM area in the bank switching RAM area of Game Boy Color. Before making any calls to *MusyX* however, you will need to make sure that the correct RAM bank has been selected.

For regular Game Boy applications, the top of RAM is also a good choice. It normally requires little change in your program, since most programmers keep their stack there, and this is easily moved from \$dff to \$deff.

Requirements for the Game Application

MusyX assumes that its core routines have been linked into their own bank at an offset of \$4000 (start of the switchable bank).

MusyX assumes that it will be called once from within every vertical blank interrupt, to process all its handling and updating tasks.

MusyX assumes that all calls to its API are made from within the vertical blank interrupt, or with interrupts disabled.

To guarantee the best possible timing, you should call the service function right after any updates to the Video RAM and before any game logic.

If normal quality samples are used in any song, the game will have to provide a timer interrupt handler that calls a second service function of **MusyX** on every timer interrupt. Also, the game needs to allow for interrupt nesting to service the timer interrupts immediately, when they occur. Failure to do so will impact the quality of the samples.

After a timer interrupt, the game cannot assume that the ROM bank is configured as it was prior to the interrupt. It is the responsibility of the application to restore its own bank configuration. (This is because **MusyX** cannot determine the current bank configurations.)

If normal quality samples are used in any song, the game is not allowed to switch between single and double speed mode on Game Boy Color. It will need to select one speed to generally run in and then not switch back and forth as long as a song or effect is playing. This will render the HALT instruction inoperable when running in double speed mode, but there is no alternative. (This is because the timer interrupt is curiously affected by the CPU speed as well.) Failure to do so will impact the quality of the samples.

Before calling any **MusyX** function, the game needs to setup the correct ROM and RAM banks.

Linking *MusyX* Object Files to Your Application

Before you can make use of *MusyX* in your application, you need to link the supplied object files to it. The object files are in the ISAS object format and can be linked only with ISLK (which are used by the official Nintendo development system for each product).

Since both ISAS and ISLK are a bit inflexible when it comes to assigning addresses and the link order of modules, we have provided the object file containing code for Bank 0 in two different versions.

After installing the example from the *MusyX* package, there will be three object files.

musyx.o

This contains the main code of the *MusyX* synthesizer and sequencer. It needs to be linked into an otherwise empty code bank. To define the bank and the offset (preferably \$4000) for this module, you enter the following to your linker call:

```
-G MUSYX=$BBXXXX musyx.o
```

where B specifies the bank number and XXXX the offset (i.e., \$104000)

musyxb0.o

This file contains service code for *MusyX* which needs to be located in the common ROM Bank 0. The group name for this module is MUSYXB0 and you need to manually assign an ORG address for it on the command line (see musyx.o). Because this is not an easy task for Bank 0 we have provided an alternate service code file.

musyxbank00.o

This is essentially the same as musyxb0.o, but the group name for this module is BANK00, which we figure is the same name as what you will be calling your bank 0. Because of this, you do not need to supply an absolute ORG address for this module on the linker command line. It will be located somewhere in bank 0 at a yet unused address.

(Please note that all three object files also exist in a subdirectory named 'CapsOff'. If you are using case-sensitive symbols in your application, you might want to use the object files in the 'CapsOff' directory.)

The current version of the service module, which you need to link to bank 0 requires \$550 (or 1360 bytes) of free space. If you do not have enough room in bank 0 to accommodate the service module, we suggest moving some of your code from bank 0 to another bank.

You will also need to specify a target address for the MUSYX_DATA group which will contain the actual converted project data. The MUSYX_DATA group **must** be assigned the next bank after the MUSYX group with an offset of \$4000. Otherwise, it is very likely that any call to **MusyX** will cause a crash.

Here's an example of how to do this.

(We assume that the source code created during the data conversion process, which includes the binary project data, was called sounddata.s (and therefore assembled to sounddata.o)).

```
islk -G MUSYX=$44000 musyx.o -G MUSYX_DATA=$54000  
sounddata.o -G MUSYXB0=$03ab0 musyxb0.o .....
```

Note that the command line in the above example uses the musyxb0.o module and assigns the address \$3ab0 to it, which is the highest possible address in bank 0 which will hold the \$550 byte long service module. If you will be using musyxb0.o instead of musyxbank00.o, we recommend using this technique, as it will relieve you of keeping track of 'the next best' bank 0 address.

Please refer to the supplied sample code for further information. If you are using OPUS make and MKMF, you can actually use the supplied make file after you have made the appropriate changes for the path names at the top of it.

We have also supplied the GNU make utility and a suitable GNU makefile (invoked by the make.bat batch file).

If you're not using makefiles at all, please refer to the makeme.bat batch file.

APPENDIX 4.1 – Game Boy Data Conversion Tools

What is Data Conversion?

Data conversion, is the process in which the sound project created with the *MusyX* editor is processed into a suitable format for the target machine. This process involves converting the actual *MusyX* project itself, all related samples and the MIDI sequences.

There are two different processes of converting the project data, depending upon whether data is needed for the game application itself, or as sample data for the sound slave.

Data suitable for implementation into the game application is usually converted from the *MusyX* project into the final target format by the application programmer, since he needs to import this data into the application.

Sample data to be used as add-on ROM samples for the sound slave, during creation of the sound, can be created by the musician himself using a simple GUI driven tool. The data this tool creates needs to be programmed onto a flash ROM. If you have no access to a Flash ROM Gang Writer, this will be the only time where the application programmer might need to be involved, to kindly flash the ROM using his development system.

To convert a *MusyX* project into either kind of data, a number of command line based tools are employed. This also applies to Windows GUI applications, which will invoke those tools to get the job done. Naturally, this requires that those tools are installed on your machine by the installation program (which is the default).

The following pages describe the purpose and use of all tools needed to produce all necessary data files.

MUCONV.EXE

MUConv is a command line driven tool which will convert a **MusyX** project into a final target platform data file. During its conversion process, MUConv will, in turn, call GM2SONG a number of times to convert the MIDI sequences into the target format. You need to make sure that MUConv can locate GM2SONG, otherwise the conversion process will fail.

The output created by MUConv consists of a number of files, three of which are to be included in the game application. More on this later.

Command Line Parameters

MUCONV.EXE expects the following command line.

```
MUCONV [Options] <Export script> <Description>
```

Export Script:

This is the filename of the export script the musician created from within the **MusyX** editor (Menu: Project->Generate scriptfile for export). This file describes the **MusyX** project in a manner MUConv can understand. Actually, you may specify more than one export script here, separated by spaces. These will be merged together. For Game Boy development however, we do not suspect that you will ever need to do this.

Description:

This is a file the application programmer has to provide. The contents of this file list which groups contained in the **MusyX** project need to be included in the final data file. It also provides information concerning where the final data files should be created. See below for details.

Options:

-a

This option tells MUConv to create an include file suitable for assembly language, rather than C. When Game Boy is specified as the target platform, this will be the default.

-b

This option will create a data file suitable for systems using big endian byte order, rather than little endian (the default). For the Game Boy platform this option has no effect.

-p <path>

If the tool is not invoked from the project directory itself, you may use this option to specify a search path for referencing all files from the project. This may be useful if you specify more than one project to be jointly converted. This option may be used multiple times, to add a search path to the search path list.

-t <system>

This option will select the target platform for which to convert the project. Possible systems are:

N64
Game Boy

-s

This option disables the processing of samples, to speed up the conversion process, when the samples have not changed.

This option has no effect for Game Boy.

-d

This option will force all samples to be converted to the default format for the target platform, overriding the musician's specifications.

This option has no effect for Game Boy.

-v

This option enables the verbose mode.

Description File

The description file is a text file containing several sections that tell MUConv what portions of the **MusyX** project to include in the target data file. It also tells MUConv the location and base name of the data files to create, and what to call the automatically created include file for all assigned IDs.

A general description file has the following layout.

```
[Pool]
...
[Samples]
...
[Project]
...
[OutputDirectory]
...
[Name]
...
[Include]
...
```

Before you can successfully convert a project, you need to fill in the blanks.

[Pool]

This section lists all group names (case sensitive!), which contribute their macros to the final project data. List one group per line.

[Samples]

This section lists all group names (case sensitive!), from which referenced samples will be taken into the final project data. List one group per line.

[Project]

This section lists all group names (case sensitive!), from which information about MIDI sequences and MIDI setups are taken into the final project data. List one group per line here, as well.

[OutputDirectory]

This section contains only one entry, which is the name of the directory into which all final data files will be written. The directory must exist in advance, otherwise MUConv will fail.

[Name]

This section contains only one entry, which is the base name (without extension) for all final data files in the output directory. All created output files share the same base name, but different extensions will be appended, according to the type of data written.

[Include]

This final section specifies the file name (with extension) of the include file to be written, which will contain all IDs for the songs and sound effects. For assembly type output, the extension for this name is usually .i (for C-type output .h).

! For Game Boy projects, you should list the same groups in every section (see example below).

Description File Example

```
[Pool]
Songgroup
SFX
[Samples]
Songgroup
SFX
[Project]
Songgroup
SFX
[OutputDirectory]
output
[Name]
GameSound
[Include]
SoundIDs.i
```

Output Files

After a successful conversion process, MUConv will have created a number of files in the specified output directory.

Three files will be named according to the base name you specified in the description file with added suffixes. The include file containing all IDs, will have been created under the full name you gave in the description file.

Taking the above example, MUConv will have created the following 4 files.

```
output\GameSound.pool  
output\GameSound.proj  
output\GameSound.s  
output\SoundIDs.i
```

.pool files

The data in these files contain all converted samples and MIDI sequences. It needs to be included in your application, starting in the ROM bank right after the **MusyX** sound routine.

.proj files

These files contain converted project data, such as sound macros and ADSR curves. This file needs to be included in your application, in the same bank as the **MusyX** sound routine.

.s files

This file is source code that you can add to your application, to include both the .pool and .proj files more easily.

include files

This file is source code that contains symbolic names for all sound effects and songs in the **MusyX** project. Since the numeric representation of these are likely to change when changes to the sound project are made, you should always reference an object by its symbolic name. This requires that you set up a dependency to this file in all your application source codes that deal with sound.

Please refer to the supplied example application, for a detailed demonstration which illustrates how to include the data files into your own application.

GM2SONG.EXE

GM2SONG.EXE is a command line based tool, which converts a MIDI-1 sequencer file into a proprietary file format, used by the runtime library of *MusyX*.

Since this tool is invoked by MUCONV.EXE, we will not explain it's stand-alone usage here.

You need to make sure, however, that GM2SONG.EXE can be found by MUCONV.EXE. You can ensure this by either placing it in your search path, or by having it in the same directory that you are in when you start MUCONV.EXE.

APPENDIX 4.2 - Function Section

The **MusyX** API consists of several functions that the application can call to do things like starting/pausing/resuming a song, starting sound effects and similar services.

The following pages contain a description for every API function, including what they do, what parameters they expect and what they return.

snd_Init

Purpose:

This function needs to be called once, during the initialization phase of your application. It will setup all internal structures and variables of *MusyX*. You must call this function to make sound.

Inputs:

- A Bit 0: Set when running on Game Boy Color. Clear if not.
- Bit 7: Set when the flash ROM is for Game Boy Color

Output:

None.

Remarks:

There are two distinctions you need to make before calling this routine.

- Are you currently running on Game Boy Color?
- Is the flash ROM enabling Game Boy Color?

For instance, you could be writing a program that does not require the Game Boy Color features, at all. In this case you would not have the CGB compatibility flag set in the ROM registration area, and Game Boy Color would behave like a conventional Game Boy. So all Game Boy Color features, such as double speed mode, do not function. But still this flash ROM can be plugged into Game Boy Color. This needs to be communicated to *MusyX*.

So:

- A = \$81 Running a Game Boy Color game on Game Boy Color
- A = \$80 Running a Game Boy Color game on Game Boy
(not permissible)
- A = \$01 Running a conventional game on Game Boy Color
- A = \$00 Running a conventional game on Game Boy

snd_Exit

Purpose:

This function immediately stops all sounds and disables **MusyX**.

Inputs:

None.

Output:

None.

Remarks:

After calling `snd_Exit`, you will need to call `snd_Init` again, prior to any other API call.

See also:

`snd_Silence`

snd_Silence

Purpose:

This function will immediately stop all sounds and any active song, but will leave *MusyX* in an active state.

Inputs:

None.

Output:

None.

Remarks:

See also:

[snd_Exit](#)

snd_Handle

Purpose:

This function needs to be called once every vertical blank interrupt.

Inputs:

None.

Output:

None.

Remarks:

To ensure the best possible timing for samples and songs, call this function as soon as possible in vertical blank. This assumes that all tasks you do before calling this function will take approximately the same time, every interrupt.

You will also need to allow interrupt nesting (see Game Boy Development Manual, Revision G, "CPU Control Register") right before calling this function, if you are utilizing normal quality samples.

snd_StartSong

Purpose:

This function will start song playback.

Inputs:

A ID of the song to start

Output:

None.

Remarks:

The ID you need to pass to this function was created by MUCONV.exe when you converted the project. Please use only the symbolic names assigned by MUCONV, as the numeric values behind them are likely to change when you convert the project again.

See also:

[snd_StopSong](#), [snd_PauseSong](#), [snd_ResumeSong](#),
[snd_ChangeSongSpeed](#), [snd_SetSongVolume](#)

snd_StopSong

Purpose:

This function will stop any song that is currently playing.

Inputs:

None.

Output:

None.

Remarks:

The song is stopped immediately, and cannot be resumed. It will need to be started again by calling `snd_StartSong`.

See also:

`snd_StartSong`, `snd_PauseSong`, `snd_ResumeSong`,
`snd_ChangeSongSpeed`, `snd_SetSongVolume`

snd_PauseSong

Purpose:

This function will pause a song that is currently playing. It can be resumed at a later time, by calling `snd_ResumeSong`. You can also save the state of a paused song into a user supplied buffer and play another song. Then, restore the buffered state and resume the first song (to play a jingle for instance), by calling the state functions `snd_GetStateSize`, `snd_SaveState` and `snd_RestoreState`.

Inputs:

None.

Output:

None.

Remarks:**See also:**

`snd_StartSong`, `snd_StopSong`, `snd_ResumeSong`,
`snd_ChangeSongSpeed`, `snd_SetSongVolume`,
`snd_GetStateSize`, `snd_SaveState` and `snd_RestoreState`

snd_ResumeSong

Purpose:

This function will resume a song that was paused. You can also save the state of a paused song into a user supplied buffer and play another song. Then, restore the buffered state and resume the first song (to play a jingle for instance), by calling the state functions `snd_GetStateSize`, `snd_SaveState` and `snd_RestoreState`.

Inputs:

None.

Output:

None.

Remarks:

See also:

`snd_StartSong`, `snd_StopSong`, `snd_PauseSong`,
`snd_ChangeSongSpeed`, `snd_SetSongVolume`
`snd_GetStateSize`, `snd_SaveState` and `snd_RestoreState`

snd_SongActive

Purpose:

This function will return whether or not a song is currently playing. The main use of this function, is to determine the end of a 'one-shot' song, like a jingle.

Inputs:

None.

Output:

A 0 = Not playing, 1 = playing
ZF 0 = Playing, 1 = not playing

Remarks:

The song itself, is certain to be at its end, when this function tells you that no song is playing. This does not necessarily mean that no further sound can be heard, since the sound macro has control over the sound. If a note is fading out, the song will have been finished, and you could prematurely end the sound synthesis. The musician should make sure that, in case of a 'one-shot' song, the last note is not a note, but a dummy program change that allows all notes sufficient time to really fade out.

See also:

[snd_StartSong](#), [snd_StopSong](#), [snd_PauseSong](#), [snd_ResumeSong](#)

snd_ChangeSongSpeed

Purpose:

This function will change the play back speed of the currently playing song. This can be used for instance, in a game to indicate that time is running out.

Inputs:

BC Speed scale factor based on \$0100 being 1.0.
If zero is passed, the song's default speed is restored.

Output:

None.

Remarks:

To speed the song up by 50%, set BC to \$0180. To undo the 50% speed increase, set BC to \$00ab (not \$0080).

Example:

- current speed = \$0100
- snd_ChangeSongSpeed with \$0180 yields \$0180 as new speed
- snd_ChangeSongSpeed with \$0080 would yield \$00c0, which is half of \$0180. To get back to \$0100, you need to specify (\$0100/\$0180)<<8, or \$00ab for this example.

See also:

[snd_StartSong](#), [snd_StopSong](#), [snd_PauseSong](#),
[snd_ResumeSong](#), [snd_SetSongVolume](#)

snd_SetSongVolume

Purpose:

This function sets a new master volume for the playback of a song.
Sound effects are not affected.

Inputs:

A New master volume (0-15)

Output:

A Previous master volume

Remarks:

See also:

`snd_StartSong`, `snd_StopSong`, `snd_PauseSong`,
`snd_ResumeSong`, `snd_ChangeSongSpeed`

snd_GetStateSize

Purpose:

This function returns the size of the buffer you need, to provide in calls to the functions `snd_SaveState` and `snd_RestoreState`.

Inputs:

None.

Output:

A Size, in bytes, of the state buffer

Remarks:

Call this function just once, during the course of your game development. Once you have determined the size of the buffer for this particular version of **MusyX**, you do not need to do it again.

See also:

`snd_SaveState`, `snd_RestoreState`

snd_SaveState

Purpose:

This function will backup the current state of the sequencer, for a paused song. After the state is secured, you can play another song (i.e., a jingle), then restore the state and resume the original song.

Inputs:

- C Size of the user state buffer (for verification purposes)
- HL Address of the user state buffer to store the state in

Output:

None.

Remarks:**See also:**

[snd_GetStateSize](#), [snd_RestoreState](#)

snd_RestoreState

Purpose:

This function will restore a previously buffered sequencer song state.

Inputs:

C Size of the user state buffer (for verification purposes)
HL Address of the user state buffer to restore the state from

Output:

None.

Remarks:

See also:

[snd_GetStateSize](#), [snd_SaveState](#)

snd_StartSFX

Purpose:

This function will start a sound effect.

Inputs:

- A ID of the effect to start
- B Volume (0-15, or 255 for default)
- C Position (0=left, 1=center, 2=right)

Output:

- A Active ID, or 0 if effect could not be started

Remarks:

The ID returned by this function, if not zero, is a handle for this sound effect. You need to keep it for a later call to `snd_StopSFX`.

See also:

`snd_StopSFX`, `snd_SetSFXVolume`

snd_StopSFX

Purpose:

This function will stop a sound effect, previously started by snd_StartSFX.

Inputs:

A Active ID returned by snd_StartSFX

Output:

None.

Remarks:

The ID returned by snd_StartSFX, will remain unique, as long as no other sound effect cancels this one due to a higher priority. In this particular case the ID you kept will become invalid, and might cancel another soundeffect that has received this ID assignment, in the meantime.

For one-shot sound effects, this is not a problem, since you need not stop them, explicitly. For permanent effects however, this could be an issue, which is best solved by having the sound designer assign appropriate priorities to the sound effects, to minimize these occurrences.

See also:

 snd_StartSFX, snd_SetSFXVolume

snd_SetSFXVolume

Purpose:

This function sets a new master volume for the sound effects.
The volume of a song remains unaffected.

Inputs:

A New master volume (0-15)

Output:

A Previous master volume

Remarks:

See also:

[snd_StartSFX](#), [snd_StopSFX](#)

snd_StartSample

Purpose:

This function will start the playback of a normal quality sample.

Inputs:

A ROM bank number of the sample to play back
HL ROM address of the sample to play back
BC Length of the sample / 16
DE Bank0 address of a callback to call when sample ends

Output:

None.

Remarks:

The sample cannot be longer than 1 MByte. This function requires the availability of the timer interrupt. The sample address needs to be aligned to 16 byte boundaries.

See also:

[snd_StopSample](#), [snd_PlaySample](#), [snd_DoSample](#)

snd_StopSample

Purpose:

This function will stop a sample being played back.

Inputs:

None.

Output:

None.

Remarks:

See also:

[snd_StartSample](#), [snd_PlaySample](#), [snd_DoSample](#)

snd_DoSample

Purpose:

This function needs to be called, when the timer interrupt occurs.

Inputs:

None.

Output:

None.

Remarks:

To insure good quality of the sample, you need to respond to the occurrence of the timer interrupt the moment it occurs. To do this, you will need to allow for interrupt nesting. Please refer to the Game Boy Development Manual, "CPU Control Register" for details.

See also:

[snd_StartSample](#), [snd_StopSample](#), [snd_PlaySample](#)

snd_PlaySample

Purpose:

This function suspends *MusyX*, stops all sound output, then plays back a high quality sample using all CPU power. When playback is completed, it resumes *MusyX* and your application program.

Inputs:

- A ROM bank number of the sample to play back
- HL ROM address of the sample to play back
- BC Length of the sample / 16
- E Bit mask for the buttons A,B,SELECT and START.

Output:

None.

Remarks:

The bitmask you supply, identifies one or more buttons that can cancel playback of the sample. This is required, because no interrupts occur during playback, and your application is suspended, as well.

The sample cannot be longer than 1 MByte.

The sample start address needs to be aligned on a 16 byte boundary.

See also:

[snd_StartSample](#), [snd_StopSample](#), [snd_DoSample](#)

snd_CheckFlag

Purpose:

Checks if a user flag, triggered by a sound, is set. This enables limited signaling capabilities from **MusyX** to your application program. When a flag is set, it remains set until this function is called. Its state will be returned, and the flag will then be cleared.

Inputs:

A Flag number to test for (0-7)

Output:

A State of the flag
ZF State of the flag (zero or not zero)

Remarks:

snd_SetMasterVolume

Purpose:

This function is used to set a new master volume in the final mixing circuit. It does affect both sound effects and music.

Inputs:

A New master volume (0-7)

Output:

Previous master volume.

Remarks:

This setting works together with the `snd_SetSFXVolume` and `snd_SetSongVolume`. It acts as a final volume scaler.

See also:

`snd_SetSFXVolume`, `snd_SetSongVolume`

Appendix 4.3 - Mini-MORT Samples

The samples you can play back, using the functions 'snd_PlaySample' and 'snd_StartSample', are "Mini-MORT" samples.

MORT is our proprietary compressed sample format, and the 'Mini' derivative of it is used for small platforms, like Game Boy.

You can create Mini-MORT samples with the supplied Mini-MORT editing tools, from 16 bit mono samples in WAV or AIFF format. The only sampling rate supported by Game Boy is 8192 Hz, so your input samples should be sampled at this rate. No resampling is done in the Mini-MORT editor, so converting a sample of a different rate is likely to yield an undesirable result when played back on the actual Game Boy.

A sample written by the Mini-MORT editor cannot be imported into Game Boy directly. You will need to remove the header of this file, manually, when you build your data resources for Game Boy.

This file header, as of version 1 MORT files, is 44 (or \$2e) bytes in size. To verify that the file you are processing is a MORT file, verify the contents of the first 8 bytes of the header and compare to the values listed below.

0x00:	DB	"MORT"
0x04:	DW	0
0x06:	DW	Version (little endian)

To verify that the file is in Mini-MORT format, check the bytes at offset 8 in the header.

0x08:	DW	1 (little endian)
-------	----	-------------------

If this 16 bit value does not contain the value 1, then this file is **not** in Mini-MORT format and therefore cannot be used on Game Boy.

APPENDIX 5 – Slave Reverb Control (N64)

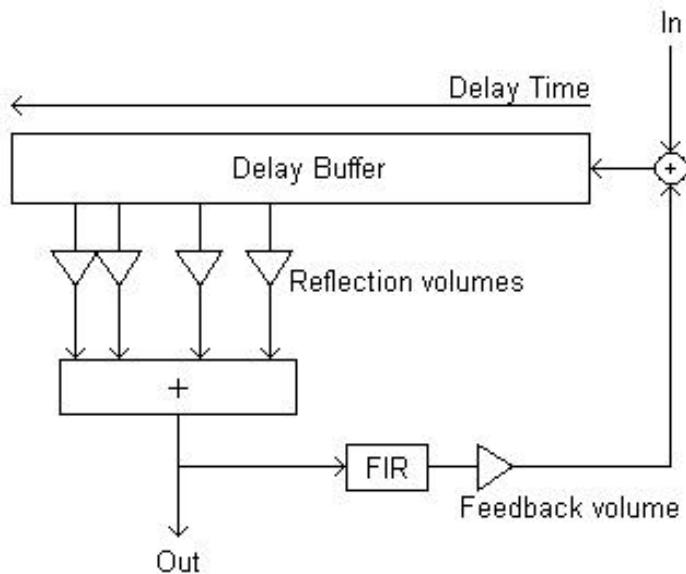
Table of Contents:

The Reverb Effects Engine (REE) on the N64	387
REE Structure.....	387
Performance Issues	388
How to Use the Reverb Panel.....	389

The Reverb Effects Engine (REE) on the N64

REE Structure

The schematic below, is a structural overview of the N64 reverb effects engine.



The incoming data is written into the delay buffer at the current write position. Next, up to eight reflections are processed by accumulating the data from the different offsets in the delay buffer, after scaling the different reflections with their specific volumes.

The accumulated signal is then mixed into the dry part of the output signal. At the same time, it is passed through a 4 point FIR filter, scaled with the feedback volume and added to the input signal, written earlier to the current delay buffer write position.

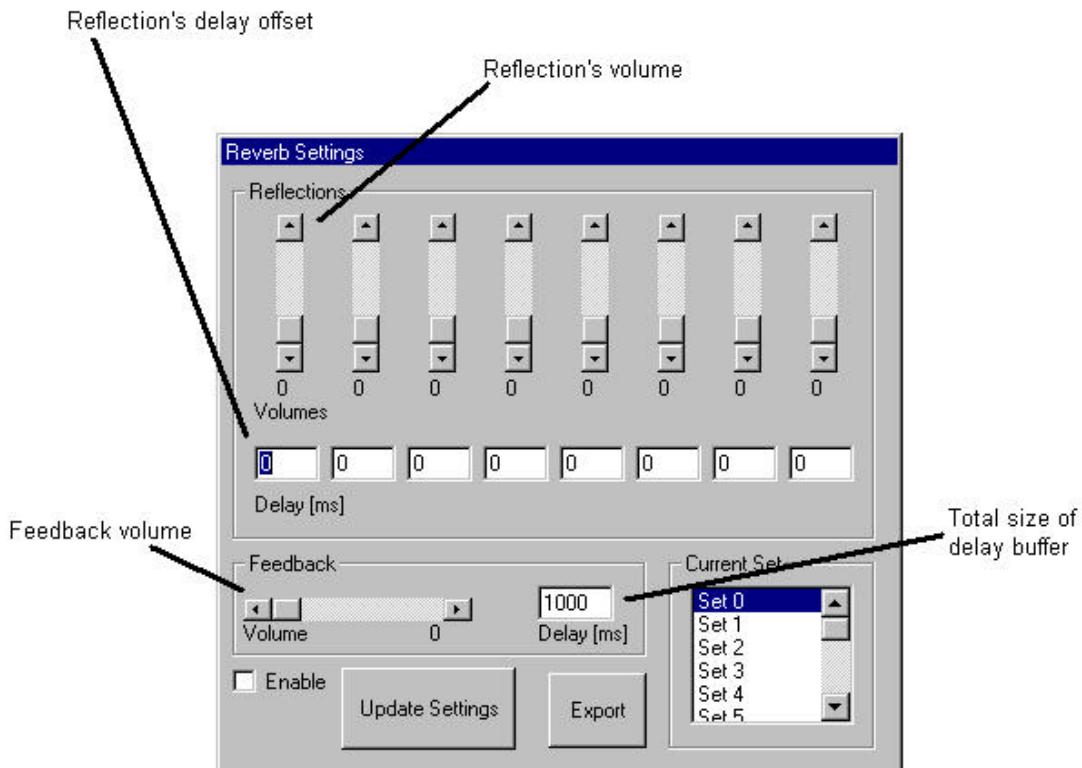
Performance Issues

The N64 implementation of REE supports up to eight reflections. When using the engine, keep in mind that resources on the N64 are limited. While great care has been taken to keep the performance hit as small as possible on the implementation side of things, there are some “rules” that can be used to minimize the impact, by the way the system is used.

- Each reflection produces additional work for the RSP. It's generally a good idea to use the smallest possible number of reflections to achieve a certain effect.
- When reflections are spread out over a wide range of delay times, this will have a larger impact on the overall performance, than the same number of reflections within a smaller range.
- Do not set the feedback volume to large values. The REE engine does not protect you against feedback signal amplitudes that are too high. These can cause the output signal to be distorted.
- The delay buffer will cost main RAM space. At 22.05 KHz 1000ms of buffer space will cost about 43 Kbytes of RAM.
- REE will align the specified delay buffer size to the next higher number of samples that can be divided by 192. So, you may not be able to specify the exact buffer length you want.

How to Use the Reverb Panel

The parameters that can be influenced using the reverb control panel, are directly mapped to the parameters in the reverb schematic from the last section. This is described in the following paragraphs.



To use the parameters, adjust the values as you see fit and press the update button. Keep in mind that you will have to do this each time you change any values, otherwise the old values will still be used. This is true for all settings, including the enable switch and the selection of different setup sets.

You may define up to 16 different setup sets at any one time. This limit is only imposed by the slave program. At runtime, the number of different settings that can be used is not limited at all by the system. These setup sets are only introduced to allow the musician and/or SFX designer to test different REE setups quickly.

The export button will bring up a file selector dialog box, that allows specifying the name of an ASCII text file that will contain all current parameters of the reverb panel, for easy access by the programmer.

Mini-M.O.R.T.

Advanced Sample Optimize Editor

General Information

What is Mini-M.O.R.T.?

Mini-M.O.R.T. is a subset of the M.O.R.T. sample compression system, specifically designed for smaller platforms (like Game Boy) that cannot deal with the required computations of the large M.O.R.T. system. Mini-M.O.R.T. has also stricter requirements for the input format of samples, which is based on the limitations of the target platform. The normal compression ratio is about 4:1. This can be increased to about 7:1 with manual adjustments.

M.O.R.T. File Formats

Source Sample Format

The Mini-M.O.R.T. editor can load samples in WAV format. Basically any sample processing software you use to create samples should be able to write in this format.

However, any sample you wish to encode into a Mini-M.O.R.T. sample needs to be a 16 bit mono audio file. If it is not in this format, the editor will refuse to load it.

Samples should have a sample rate of either 1920 Hz or 8192 Hz. The editor can load samples of any rate, but it bases its mode of operation on a sample rate of 4000Hz. Any sample below this rate will be treated as "low" quality, all others as "normal" and "high" quality.

Regardless of the sample rate you load into the editor, Game Boy can play back only at 1920 Hz and 8192 Hz. It chooses the appropriate frequency based on the "low" or "normal/high" setting. This means that samples will not sound correct if they were not sampled in one of the two supported sampling rates.

MIF (M.O.R.T. Information File)

The MIF File is written by the editor, and contains sample processing information for the sample you edited. This file is useful if you have already applied Mini-M.O.R.T. compression information on a sample, but need to make changes to the original wave file.

Change the Wave File as desired and load it into the Mini-M.O.R.T. editor. Then press the 'Reload MIF'-button. If you saved the MIF File the first time, you will find all your blocks optimized, after you have applied the MIF-data to the voice-file. For this to work as intended it is a requirement that you do not change the overall length of your sample (like changing pitch, cutting sections from it, etc.), or otherwise you will end up with the information stored in the MIF file at blocks where they do not belong to.

Blocks, Curves and Easy Optimize

General

A Mini-M.O.R.T.-file is divided into blocks of 32 samples each.

The actual voice compression scans each block and compresses it. You can increase the compression ratio by marking some blocks as 'Empty' (visualized as a green block).

If at least two consecutive blocks are marked empty, the Mini-M.O.R.T. voice compression will disregard these blocks and insert zero-data.

To mark a block 'empty', it is necessary that the previous block ends on zero and the next block starts on zero, to assure a smooth playback. To achieve this, you can choose from three different curves that are used to fade the previous block out and the next block in. Remember, these curved blocks (yellow, orange and pink depending on the applied fading curve) are not empty and cannot be used to increase the compression ratio.

The Easy Optimize section will help you find empty blocks and mark them empty. You can adjust the threshold (1-32767) of this 'Noise-Gate'-algorithm. When the Optimize-button is used, the tool will mark all those blocks whose amplitudes stay within the threshold range as empty.

To manually mark any block as empty you need to click on it with the left mouse button. If you want to remove this empty mark, select it with the right mouse button and choose "Free" to restore the original data.

The Mini-M.O.R.T. editor will load samples of any rate, but it bases its mode of operation (sample quality) at a rate of 4000Hz. Any sample with a lower rate is considered to be of "low" quality. All other samples will enable the "normal" and "high" quality settings.

General Functions

Open / Open New

Opens a new .WAV sample for editing. Please examine the previous table for details on sampling rates.

Save / Save As

Saves the sample file. Because of the non-destructive nature of the Mini-M.O.R.T. Editor, saving your source sample is not necessary. The exception to this is when you are in the Mini-M.O.R.T preview mode, and want to save the generated Mini-M.O.R.T. file as a demonstration.



This will play the entire voice file from the beginning of the file.



This will play the voice file from the current position.



This will stop playback of the voice file.

Original / MORT

With these functions, you can switch between the original voice file and an encoded Mini-M.O.R.T. file for preview purposes. This shows you the difference in sound between the Mini-M.O.R.T. file and the original wave file.

Save MIF

Saves the MIF file (M.O.R.T. Information File).

Do MIF

Applies the MIF data to the voice file. Use this with the "Free All" command to get an A/B comparison with the original voice file and your optimizations.

Free All

Sets all blocks free, so you can get the original status of the file. Use this with the ‘Do MIF’ command, to get an A/B compare with the original voice file and your optimizations.

Reload MIF

Loads the matching MIF file for the opened voice file. This function is not available if you have not saved a MIF file for this voice file before.

Reload WAV

Reloads the wave file. This is a useful feature that enables you to hear how your changes to the wave file affect the Mini-M.O.R.T. voice compression.

Save MORT

Saves your final Mini-M.O.R.T file. After using “Save MORT”, it is not possible to change anything in the .mort file. **It can not be loaded into the Mini-M.O.R.T. editor again.**

About

Shows the copyright Information.

Help

Not implemented.

Exit

Exits the editor.

Statistics

Encode

This will test-encode your file and give you information about the compressed file size and the compression factor.

The "Noise Filter" checkbox right below the "Encode" button enables a special noise filter for Game Boy samples. It may or may not increase the quality of normal and high quality samples.

You should try A/B comparisons of an encoded sample with this filter turned on and off to see if the current sample sounds better with this filter enabled.

Comp.Size

Is the file size of the compressed Mini-M.O.R.T. file in bytes.

Comp.Factor

Is the compression ratio, as compared to the original file size.

Filelength

Is the length of the original wave file in samples.

Rate

Sample Rate of the wave file.

NOTE: Either 1920 Hz or 8192 Hz are supported on the Game Boy.

Mode

Shows the amount of channels in the original wave file.

NOTE: Mini-M.O.R.T. only supports MONO files with one channel.

Bits

Shows the resolution of the original wave file.

NOTE: Mini-M.O.R.T. only supports 16 bit files.

Position

Shows the start-position of the sample-window in blocks. You change the start position by scrolling the scrollbar, or setting a block as the new start position. Selecting the block with the middle mouse-key does this. This is helpful if you want to check a specific range of a sample with the  Button.

Scalefactor

This is the scaling factor you chose in the view area.

View

1:1 - Full

With these Buttons you select a scalefactor and choose how much you want to zoom into or out of the Sample.

Redraw

Redraws the contents of the sample-window.

Grid

Switches the Grid on or off.

B-Style

Shows the sample-data in a different display style.