ARTISAN
uMIDI CONTROL SYSTEM
MANUAL

For Support Call: (910) 739-5481
Or email: emarka@mac.com

Table of Contents

Planning the Micro MIDI Control System

The first step in implementing a Micro-MIDI system is to plan the system.

Count Inputs and Outputs

The first association involves those items of organ elements that require Serial Input Boards. These are,
Keyboards
Pedalboard
Stop Sense Switches
Pistons
Determine the number of Serial Input Boards required by adding together the total number of inputs.
Serial Input Board:

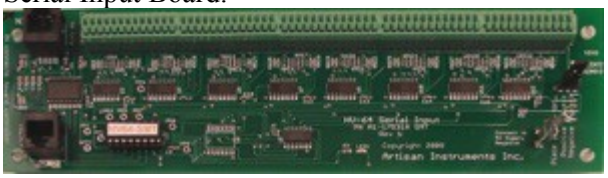

Illustration 1: Serial Input Board

A keyboard, for example, requires 61 inputs. The remainder of the organ elements are quite variable, and require an input for each sense switch or piston contact.
Since the Serial Input Boards have up to 64 inputs, the total number of inputs divided by 64 determines the number of Serial Input Boards required.
Inputs are connected to HV-64 boards. HV-64 boards sense voltages. There are two varieties of HV-64 boards, the HV-64P and the HV-64N. By default, the HV-64P is activated when the applied voltage is between 5 and 30 volts, and thus must be wired to switches which have a voltage connected to the other side of the switch. This voltage should be between 5 V and 30 V. HV-64N boards respond to voltages between -30V and 0V, so HV-64N boards may be connected to switches which have a common ground. Up to four HV-64 boards may be connected to one Micro-MIDI board, so up to 256 inputs may be sensed per Micro-MIDI board. However, for improved latency, it is recommended that only two HV-64 boards be used per Micro-MIDI board when the inputs are time-critical, such as keyboards.

Illustration 2: uMIDI Board

Pots (controllers) may be connected either to HV-64 boards or directly to the Micro-MIDI board. Up to four pots may be connected to a Micro-MIDI board. (Note: Early Micro-MIDI boards did not allow this.) Up to seven pots may be connected to one HV-64 board; connecting n pots will use n+1 inputs on that board.
Rank driver boards may be used to drive lamps or magnets. Up to 96 outputs may be driven with one rank driver board, and one Micro-MIDI module is required for each rank driver board.

Are Other Modules Required

In some cases, extra Micro-MIDI modules are required for processing the MIDI data stream.

Is a Relay Module Required

Some configurations which are driving a MIDI sound engine require an extra Micro-MIDI board to implement a relay function. If the sound engine for a system knows how to implement stop functions using note information from some channel, then it is not required. However, if the sound engine responds with a specific rank on each of a number of channels, a relay function is necessary to distribute keyboard information to the various channels as selected by stop tabs.

Are Modules Required for Other Features

If reiteration or pizzicato are to be implemented, these will usually require an extra module to do this processing. Similarly for sostenuto. These three features can likely share a single module.

Power Requirements

The Artisan Micro-Midi system needs 5V to operate. The requirements are a 5V power supply rated for the maximum current required by the system. The power supply should be rated at 5V +/- 5%. Any voltage greater than 5.5V may irreparably damage the system.
The current requirements are modest. A 1 Amp supply should be adequate for most systems.

The uMIDI board itself should take less than 25 mA, and each serial board connected to it should take less than 20 mA.
In addition, if your system includes Lighted Stop boards, a 12 V supply is required for the lamps. The lamps take about 60 mA each at 12V. For a 32 stop tab box (four lighted stop modules with 8 lamps each), a supply of about 2 A is required.

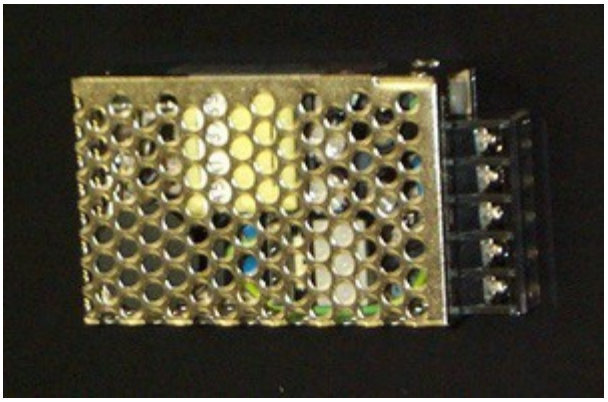

Illustration 3: Power Supply

Assigning or Determining MIDI Channels

The next task in implementing the Micro-MIDI Control System is to establish the MIDI channels that will be used. This, in turn, will relate to the way the software is set up in the Micro-Processor on the Micro-MIDI Board.
While the following may not be the same for all organs, it is the convention that Artisan Instruments has adopted. Many of the manufacturers follow this convention.
First, there is one MIDI Channel per manual division, with the Great always being Channel 1.
The second manual division is always Channel 2
The pedal division is always Channel 3
Then, if there are other manuals, they follow in sequence with the third manual division being Channel 4, the fourth manual division being Channel 5, etc.
A typical 2-manual organ would have:
Great = midi channel 1
Accomp = midi channel 2
Pedal = midi channel 3
A typical 3-manual organ would have:
Great = midi channel 1
Accomp = midi channel 2
Pedal = midi channel 3
Solo = midi channel 4
A typical classic organ would have:
Great = midi channel 1
Swell = midi channel 2
Pedal = midi channel 3
Choir = midi channel 4
Bombarde = midi channel 5
Echo = midi channel 6

It has become the industry standard to transmit and receive stops as midi note information, starting on midi note 1 over midi channel 12. Other channels above 12 can be used, as necessary. The industry has designated to transmit and receive midi controller information such as volume over the same channel as its note information, i.e. 1 through n. The accepted midi controller number for volume is controller 7, with a value of 0 to 127 .
While not an industry standard, Artisan has chosen to use channels 15 and 16 for pistons.

Host Computer Requirements

In order to configure a Micro-Midi system, one needs a Microsoft Windows-based computer with a MIDI interface. Many computers have a joystick port, which is also capable of MIDI I/O. The only needed addition is a special MIDI cable, which is available at many computer or music stores.
The drivers for the MIDI interface will also need to installed, if they are not already installed. If the drivers are not installed, this will become obvious the first time you try to run the configuration software.
If a computer is not capable of MIDI I/O, a number of USB to MIDI interfaces are available commercially.

Illustration 4: USB MIDI Interface Cable

Count how many Micro-MIDI boards will be required, what I/O boards will be needed, and what order they should be wired in. Remember that MIDI information travels from module to module, and inputs needed by a particular module need to be connected to a module which comes before that module. For example, piston and tab inputs must be located before mag tab drivers or lighted stop modules.
First count the input modules. One Micro-MIDI board may receive inputs from up to four HV-64 boards (256 input bits). As described elsewhere, up to seven inputs per HV-64 board may also be connected to potentiometers and generate controller signals, although one of the inputs then needs to be dedicated to a reference signal. Micro-MIDI boards used for inputs may not be used for any other purposes.

One Micro-MIDI board may control up to 8 Lighted Stop boards, or 64 stops. This Micro-MIDI board may not be connected to any other I/O board.

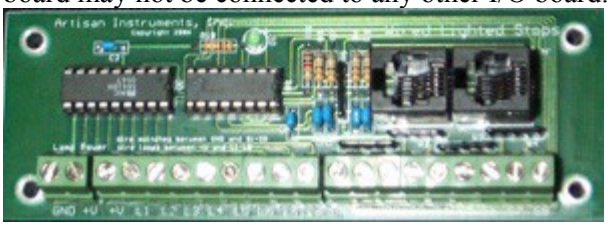

Illustration 5: Lighted Stop Board

One Micro-MIDI board may be connected to one rank driver board, or up to 96 output bits. If the rank driver board is used to drive dual mag tabs, it may not drive any pipes and the Micro-MIDI board may not perform any other functions (such as relay). As this is written, up to three HV-32 output boards may be used by lying to the configuration program and claiming that one rank driver board is connected.

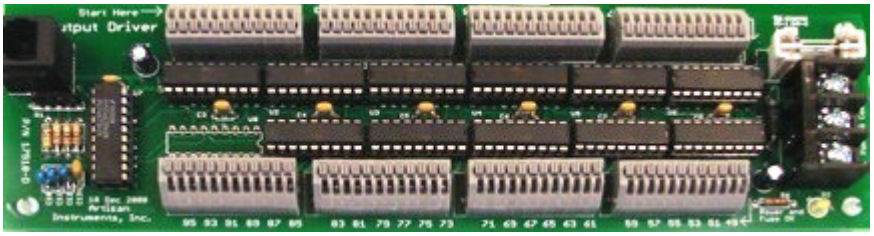

Illustration 6: Driver Board

One Micro-MIDI board may also be used to perform relay functions. These functions can either generate MIDI messages which are transforms of the MIDI messages which the module receives, or may generate pipe chest control messages for a rank driver board. A Micro-MIDI board which performs relay functions may not perform any other input or non-rank output functions.
Note that when a Micro-MIDI board is used to transform its input MIDI signals to different output MIDI signals, the output MIDI stream usually no longer contains the information which would be necessary to perform further relay functions on subsequent Micro-MIDI boards. That is, if the MIDI input contains division information on channels 1, 2, and 3, and the MIDI output contains rank information on channels 1 through 16, the raw division information is no longer present in the data stream. This limits the output of a system to 16 channels of information. This limitation can be overcome by splitting the MIDI data stream and feeding multiple Micro-MIDI modules in parallel. Each Micro-MIDI module can then generate up to 16 channels of output information in parallel. This scenario is implemented by programming the relay boards as if they were going to be connected in series. Connect all the boards in a series chain in the order they are described in the configuration file, and download the program. Now, reconnect the boards in parallel using the splitter to operate the system.
In general, inputs should be to the left of (or precede in the MIDI message chain) any output functions. The exception is keyboard inputs: If none of the inputs on a module are needed by other Micro-MIDI modules, it is best to put those modules last in the MIDI chain so as to minimize the latency between pressing a key and the MIDI message exiting the Micro-MIDI system.

Software Installation

The configuration program, uConfig, is installed from a windows installation program. The needed file may be downloaded from http://www.artisanorgans.com/
Simply double-click on the file after it has been downloaded, and the installation should proceed. The file is a self-contained installer.

Mounting and Wiring

This section describes the physical implementation of the Micro-MIDI system

Mount the Modules

Input and output modules should normally be mounted near to the switches or magnets they connect to, in order to minimize the amount of wire required. The Micro-MIDI boards are normally grouped together for convenience. When mounted vertically, it's probably most convenient to place the MIDI connectors at the bottom of the module, so that gravity assists in keeping the MIDI cables out of the way.
Cables between the Micro-MIDI board and the various I/O boards may be up to 25 feet. MIDI cables may be up to 50 feet (15 meters) long.

Wire the Power

A five volt supply is needed for the Micro-MIDI modules. This power supply must be a regulated supply. Applying voltages higher than 5.5V, or wiring the supply backwards, even momentarily, may permanently damage the system. The negative terminal of the supply must be connected to the "GND" terminal of the Micro-MIDI boards, and the positive terminal of the supply must be connected to the "+5V" terminal of the Micro-MIDI boards.
If there are more than two supplies, their commons or negative supplies are normally connected together.

Wire Inputs

HV-64P boards must be wired so that operating the attached switch causes a voltage of 5 to 30 volts to be applied to the terminal. When the switch is open, the terminal will drop to 0V thanks to a resistor to ground on the board. Controller potentiometers may be wired to inputs 57 through 63 as desired. In this case, an external reference voltage must be applied to one end of all of the pots. The common voltage, which can be found at the tab on the end of the HV-64 board, should be applied to the other end of all the pots. The wipers are then connected to the appropriate HV-64 inputs. The reference voltage used for the pots must then be connected to input 64.

HV-64N boards must be wired to switches with a common voltage between -30V and ground. When the switch is open, the terminal will rise to 5V thanks to a resistor on the board. Lowering the voltage to 0V or below with a switch enables the input.

Wire Outputs

Rank driver boards are designed to operate magnets or lights. One terminal of each controlled device should be connected to one of the terminals on the rank driver board. When connecting to dual magnet tabs, the magnets should be connected to adjacent pairs. For example, the on magnet

of the first tab should be connected to output 1, and the off magnet of the first tab should be connected to output 2.
All devices connected to a rank driver board should operate from a single power supply. Rank driver boards have three terminals. If held so that the terminal strip is to the right, the bottom terminal is the negative terminal, and should be connected to the negative terminal of the power supply. The middle terminal should be connected to the positive terminal of the power supply. The top terminal should be connected to the common bus of all of the controlled devices.

Wire the RJ-11 Connectors

Cables should be prepared to connect the Micro-MIDI board to the various input and output boards as follows. Up to 25 feet of cable may be used per Micro-MIDI module. When using multiple boards, this is the total length of all hops.
1) Cut the cable to the required length, and strip both ends with a special cutting and crimping tool designed for this type of termination. These tools can be found in stores that sell telephone cable or telephone installation supplies. The stripped length must be long enough so that the wire can reach the end of the connector, but must be short enough so that the outer jacket fits inside the connector and acts as a strain relief.
2) Make sure the stripping tool has not cut into any of the conductors. Then, hold the cable as shown so that the White wire is on the right. (The Blue wire will be on the left). Verify that the six conductors are in the same order on the two ends of the cable.
3) Place the stripped end of the cable into the RJ-11 phone plug. Note that the locking tab is on the underside. A good "Rule-of-Thumb" is to establish the cable and connector position so that "WHITE IS ON THE RIGHT".

Illustration 7: 6 Strand Cable

4) Crimp the connector tightly with the crimping tool. This method provides the proper orientation for the cables when plugged into the receptacles on the Micro-MIDI board(s) and the various other boards. In effect, it provides the 180 degree twist in the cable ---- which is correct. The cables can now be plugged into the appropriate RJ-11 jacks. When connecting multiple boards to one Micro-MIDI module, the cable should go from the Micro-MIDI module to the IN connector on the first board. Subsequent boards are wired using cables between the NEXT connector of one board and the IN connector of the next board.

Illustration 8: Crimping Tool

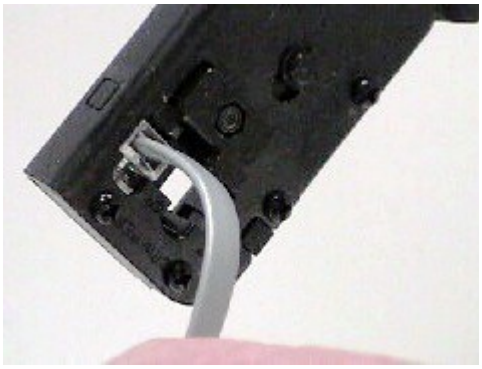

Illustration 9: Crimping RJ-11 Plug

Wire the MIDI

Connect MIDI cables from the MIDI OUT connector of the first Micro-MIDI module to the MIDI IN connector of the second. In like manner, connect subsequent modules. For configuration purposes, the MIDI IN connector of the first Micro-MIDI board should be connected to the MIDI OUT connector of the configuring computer, and the MIDI OUT connector of the last Micro-MIDI module should be connected to the MIDI IN connector of the configuring computer. The end result should be a MIDI loop, from the computer, through each Micro-MIDI board in order, and then back to the computer.

Writing a Configuration File

A text-based configuration file needs to be created which describes the layout of the system. The configuration file lists the Micro-Midi boards in order from left to right, and describes what messages the module should generate or sense.
After all of the system hardware is in place, the next step is to develop a definition file that corresponds to the hardware. More specifically, the definition file will conform to the overall specifications of the application and how this hardware will perform; encompassing keyboards, stop/couplers, pistons, controls, volume, and all.
For that, there needs to be a procedure for the definition, showing the various Key Words (those preceded by an asterisk), and other terminology that will form a complete definition, or Configuration File.

General format rules for the configuration file

All keywords begin with an asterisk (*). Keywords may be written in capital letters, lowercase letters, or any combination desired. Thus, *INPUT_BIT, *input_bit, or *Input_Bit are all valid for the *INPUT_BIT keyword.
Names that the user supplies can also be in upper or lower case, but they must be consistent. For example, “Diapason_8” and “diapason_8” would not be recognized as the same name. Usersupplied names must begin with a letter, and may contain letters, digits, and the characters ! @ # $ % ^ & * _ = + | / ? - and *. Spaces are not allowed within names.
As much, or as little, information as desired can be entered on each line, although don't go overboard in either direction.

Comments may be added by using a semicolon (;). All text between the semicolon and the end of the line is ignored.
When saving the configuration file, the editor being used must be told to output the file in plain text mode. The standard extension for Micro-MIDI configuration files is .ucf (uMIDI configuration file). When saving the file, this .ucf should be entered as part of the file name to make it easier to locate these files.
The last line of a configuration file must be
*END
Any text located after *END will be ignored.

Module Declaration

Each Micro-MIDI module has a configuration section, which begins with the keyword *UMIDI_MODULE. All inputs described up until the next *UMIDI_MODULE (or *COMBINATION_ACTION) keyword are assumed to be connected to the corresponding Micro-MIDI board.
Expanded Memory Module affords additional memory and processing power specifically for use with combination action driver boards. The proper declaration for the expanded memory module would be as follows:
*UMIDI_MODULE_B

Input Board Declaration

When using HV-64 boards as inputs, the *UMIDI_MODULE keyword should be followed by the *HV64 keyword. There should be as many *HV64 keywords after the *UMIDI_MODULE as there are HV-64 boards connected to that module. Each *HV64 keyword should be followed by a description of what is connected to that input board. There may be up to four HV-64 boards connected to a Micro-MIDI module. Note that HV-64 boards respond to voltage inputs (5-30V), and can also be programmed to generate up to seven controller signals.
More recently, the HVX-64 board has been introduced in two variants, one for positive voltage sensing (the same as the HV-64 board), and one for negative voltage sensing. The negative voltage sensing board can sense switch closures to ground or voltages as low as -35V.
A number of features have been added which greatly increase the flexibility of these boards to sense signals in various situations. In particular, the voltage threshold can be changed, or the sense can be inverted so that higher voltages are on and lower voltages are off. More information about the HVX-64 boards can be found in the following document: The Artisan HV-64

Keyboard inputs

A keyboard is simply a collection of sequential input bits. It is described thusly:
```
*DIVISION=great *BITS= 1,61 *MIDI_CHANNEL = 3 *MIDI_NOTE =
36 ; This is a comment
```
This specifies that a keyboard is to be sensed, and turned into note ON and note OFF messages. This keyboard is connected to bits 1 through 61 of the input board.
When key presses are sensed, they will be transmitted on MIDI channel 3.
The first note (that is, bit 1) will be transmitted on MIDI note 36, the second (bit 2) on MIDI note 37, etc. There is also an example of a comment here: all characters after the semicolon are ignored.
Note that the parameters after `*BITS` specify the first bit, and the number of bits. For a 32 pedal pedalboard connected to a serial input board, one would specify:
```
*BITS=1,32
```
There are two uses of the `*division` statement. This section covers the creation of a division from an input board connected to the Micro-MIDI system. However, sometimes divisions come from an external source, such as a MIDI keyboard. In this case, a `*division` statement with a slightly different syntax is used. This is covered in section *Initial Declarations*. Initial declarations always precede the first `*umidi_module` statement.
*DIVISION may be abbreviated as *DIV
*MIDI_CHANNEL may be abbreviated as *MCH
*MIDI_NOTE may be abbreviated as *MNO
*MIDI_CHANNEL = 3 *MIDI_NOTE = 36 may be abbreviated as c3n36

Single Bit Inputs

To define stop tabs or other single-switch inputs connected to an input board, then after the `*HV64` keyword, each tab should be described in a manner similar to this:
```
*INPUT_BIT=tibia_8 *BIT=2 *MIDI_CHANNEL=12 *MIDI_NOTE=1
```
The `*INPUT_BIT` keyword introduces the description of a single bit on an input board which is to be sensed, and changed into MIDI note -ON and -OFF messages.
It is optionally followed with an equal sign (=) and a user-supplied name which links it to other uses within this configuration file.
The `*BIT` keyword specifies which bit of the input board the signal is connected to.
The `*MIDI_CHANNEL` keyword specifies which channel the note-ON and -OFF messages should be transmitted on.
The `*MIDI_NOTE` keyword specifies which MIDI note should be used for this particular bit.
In this example, if the tab connected to bit 2 of the input board is turned on, the Micro-MIDI board will transmit a note ON message on channel 12, note 1, with a velocity of 127. If the same tab is turned off, the Micro-MIDI board will transmit a note OFF message, which is actually a note ON message on channel 12, note 1, with a velocity of 0. This tab is assigned the name "tibia_8".
Instead of generating note messages, program change messages may be generated. To do this, replace `*MIDI_NOTE` with `*PROGRAM_CHANGE`. This syntax will cause the given program change command to be sent when the input turns on, and nothing to happen when the input turns off. For example:
```
*input_bit *bit=15 *mch=16 *program_change=5
```
To generate a program change message when an input turns off, add the `*INVERT` keyword to the line. An example of this:

```
*input_bit *bit=15 C16p6 *invert
```
*INPUT_BIT may be abbreviated as *IBIT
*MIDI_NOTE may be abbreviated as *MNO
*MIDI_CHANNEL = 12 *MIDI_NOTE = 1 may be abbreviated as c12n1
*MIDI_CHANNEL = 16 *PROGRAM_CHANGE = 6 may be abbreviated as C16P6
SPECIAL NOTE: The upper 3 inputs of a manual division input board are not usable for Single Bit Inputs.

Alternate Action Inputs

An alternate action input is an input which switches on when the input is activated, and then switches off when the input is activated again. In the Micro-MIDI system, when an alternate action input is turned on the first time, it sends a note on message. When it turns off, nothing happens. When it turns on the second time, it sends a note off message. Note on and note off messages alternate every time the input turns on.
To create an alternate action input, use syntax which looks almost the same as a single bit input. The only difference is that *ALT_ACTION_INPUT replaces *INPUT_BIT, and program change messages are not allowed. Here's an example of an alternate action input:

```
*ALT_ACTION_INPUT=sfz *MCH=4 *MNO=7 *bit=15
```

In this example, every other time input 15 turns on, a MIDI note on message will be transmitted on channel 4, note 7. The other times, a note off message will be transmitted.

Controllers

Controller signals may be generated in several ways: From analog signals, such as from pots or from binary or one-of-n inputs. There are two ways to capture analog inputs: Directly on the Micro-MIDI module, and on the HV-64 board

Controllers on Micro-MIDI module

One potentiometer (pot) may be connected to the three terminals on the edge of the board labeled "GND", "A1", and "+5V". The wiper on the pot should be connected to the A1 pin, and the other two pins to GND and +5V. Usually, the pin corresponding to counter-clockwise rotation of the pot will be connected to GND.
The wipers for three more pots may be connected to A2, A3, and A4, with the other two pins on each tapping off the GND and +5V pins.
Other analog voltages may be connected to the A1 through A4 pins, but it is essential that the voltages be limited to the range of 0V to the power supply voltage, or damage to the microprocessor may occur.
Under the appropriate `*UMIDI_MODULE` heading, include a definition like the following:

```
*control=c2vol *midi_channel=2 *bit=1 *midi_controller=7
*range=10,50
```

This specifies that the voltage at pin A1 is to be converted into a controller 7 message on channel 2, with 0V resulting in a controller value of 10, and 5V (actually the power supply voltage) resulting in a controller value of 50. The optional "=c2vol" gives this controller the name of "c2vol", which may be referenced later on in the configuration file. The other pins A2 through A4 are accessed by using `*bit=2` through `*bit=4`.

Note: Controllers for Micro-MIDI module inputs should appear directly under the `*umidi_module` line, otherwise an error message may be generated.

Controllers on HV-64 board

NOTE: It is always preferred to use the analog inputs directly on the uMIDI Control board but if absolutely necessary we do make provision for the following controllers:
Controllers on the HV-64 board differ in a number of ways.
First, up to seven controllers may be defined, using input bits 57 through 63.
Second, the allowable voltage at these pins is 0 through 40V. (+5V through -35V for HV-64N)
Third, the reference voltage must be applied to pin 64 as well as the "high" end of the potentiometer used for the controller.
Each of the (up to) seven controllers connected to a HV-64P (HV-64N) will be at its minimum value when the input voltage at its pin is 0V (5V), and will be at its maximum value when the input voltage at the pin is equal to the reference voltage at pin 64. Any higher voltage will be clamped to the maximum value.
The definition should be located under the appropriate `*HV64` keyword:

```
*control *midi_channel=2 *bit=62 *midi_controller=7
*range=10,50
```

The resolution of the A/D converter is 256 counts, and the maximum count is at about 40V (-35V) or eight times the power-supply voltage (-7 times the power-supply voltage). If you use the 5V supply for your reference voltage, there will only be about 32 (256/8) counts available. Thus, if you are trying to generate a controller with values from 0 to 127, it will jump from 0 to 127 in about 32 steps, missing many of the intermediate values. This effect can be mitigated by using a higher voltage for the reference voltage.
Note that the input resistance of an HV-64 board is about 3.7 kOhms. Connecting to a higher resistance source, such as a 10 kOhm pot, may cause the resulting controller to be non-linear. In particular, most of the counts at the high end will be bunched together in a small amount of rotation.
The newer HVX-64 boards have a higher 37 kOhm input impedance, making a 10 k pot a good choice.
When using the HVX-64N boards (which sense negative voltages), the pot must be connected between +5V and a negative voltage, with the negative voltage applied to pin 64. The HVX-64N arithmetic is modified so that more negative voltages result in larger controller values, and +5V is the minimum value. This behavior can be disabled with the keyword *positive_voltage, or it can be selected with the keyword *negative_voltage. If the default behavior is correct, it is best not to include either keyword.

Binary Controllers

A group of up to 8 input bits may be used to create a controller. This may be done using an HV-64 board. The syntax for this is:

```
*control=myName *binary=23,6 *midi_channel=5
*midi_controller=12
```

This will use 6 input bits starting at bit 23, and treat them as a binary number. Bit 23 will be the least significant bit. The optional "=myName" gives this controller a name which may be used later in the definition file. Without further modifications, this definition will create a controller which will vary from 0 to 63. You may add scaling to this:

```
*range=10,20
```
This causes the input values of 0 through 63 to be scaled linearly so that 0 creates a controller value of 10, and 63 creates a controller value of 20.
As a trivial case of a binary controller, an on-off controller may be created from a single input bit:
```
*control=myName *binary=23,1 *midi_channel=5
*midi_controller=17 *range=0,127
```

One-of-n Controllers

A group of up to 64 input bits may be used to create a controller. When using a one-of-n controller, it is assumed that only one of the n inputs is active at a time. Whenever an input transitions from off to on, the controller will change to the value of that input. An example is:
```
*control=myName *one_of_n=10,12 *midi_channel=7
*midi_controller=17
```
This specifies that 12 inputs, starting at input 10 are to be used to generate controller messages. If input 10 goes from off to on, a controller message will be transmitted with a value of 0. If input 21 goes from off to on, the controller message will have a value of 11. Inputs which remain on are ignored, and nothing happens when an input turns off. Again, the controller can be given a name with the optional "=myName".
A one-of-n controller may be scaled in the same way as a binary controller.

Computed Controllers

An arbitrary expression may be turned into a controller. The syntax for this looks like this:
```
*control (g_tibia * 8) *midi_channel=4 *midi_controller=17
```
This statement will generate a controller which has the value of 0 when the g_tibia tab is off, and 8 when the g_tibia tab is on. The controller will be on channel 7, controller number 17.
*MIDI_CHANNEL = 4 *MIDI_CONTROLLER = 17 may be abbreviated as c4c17

Increment Decrement Controllers

See Keywords section for full implementation. This function may be used when it is desired to use two buttons (up and down) to step controller value for Memory Level or Transpose.

Sending Program Change Messages

Program Changes based on an Analog Input

You may wish to send a program controller message based upon the position of a control, such as a potentiometer. This can be done with syntax similar to a controller statement, as follows:
```
*send_program_change *midi_channel=4 *bit=1 *range=1,10
```
This statement will watch the position of the pot connected to analog input 1 on the Micro-MIDI module, scale it to a number between 1 and 10, and generate a program change message on channel 4. When the position of the pot changes enough to change the scaled value, a new program change message will be generated.

Computed Program Changes

A program change message may be sent based upon an arbitrary expression. This can be done as follows:

```
*send_program_change (c3c5 + 2) *midi_channel=6
```

This example would calculate the value of controller 5 on channel 3, add two, and send out the resulting value as a program change on channel 6. Whenever controller 5 on channel 3 changes, a new program change message will be generated.

*SEND_PROGRAM_CHANGE may be abbreviated as *SPC

Lighted Stop tab inputs

There are either 4, or 8, lighted stops in each Lighted Stop Module. That is, a single row, or a double row. And, up to 4 Lighted Stop modules can be placed in a rack panel, or 32 stops overall.

When defining either a single Lighted Stop module, or the group of Lighted Stop modules in a complete panel, the numbering scheme is 1 to *n* across the upper, or first row of switches. Then, the numbering folds back and continues with the first stop on the second, or lower row of switches.

A Lighted-stop box is specified like this:

```
*UMIDI_MODULE
*LIGHTED_STOP *COUNT=4 *MIDI_CHANNEL = 3 *MIDI_NOTE = 1
```

`*COUNT` specifies how many Lighted Stop modules are in one row of modules. Each module is four switches wide, so for 32 bits of Lighted Stops, arranged as two rows of 16 switches each, `*COUNT=4` is the correct setting.

The `*MIDI_NOTE` specifies which MIDI note should be used for the first button on the first module.

In the above example, turning on stops on the top row of switches will cause transmission of MIDI note ON messages on channel 3, with the notes being from 1 to 16 from left to right. The second row is then notes 17 through 32, left to right.

For more than one row of Lighted Stop modules, that is more than two rows of switches, each row should be specified with another `*LIGHTED_STOP` section.

Some lighted stop modules are supplied with only a single row of switches; that is, four switches per module. In this case, add

```
*SINGLE_ROW
```

to the definition so that resources are not reserved for the non-existent switches.

*MIDI_CHANNEL = 3 *MIDI_NOTE = 1 may be abbreviated as c3n1

Piston inputs

In order to incorporate a combination action, piston switches which can be sensed should be added. In addition, one or two more switches are needed to configure the combination action: one labeled “SET,” and an optional one labeled “MAP”. These switches may also be connected to input boards, along with keyboards and stops. Each piston switch should be identified with an `*INPUT_BIT` definition.

Note that the switches in the Lighted Stop Modules cannot be used for pistons or set/map

switches. These must be in the form of general single touch thumb pistons.
Either note messages or program change messages may be used for piston actuation. In order to change an *INPUT_BIT definition from a MIDI note message to a MIDI program change message, change *MIDI_NOTE=x to *PROGRAM_CHANGE=x. This will cause a program change message to be used for the piston, instead of a note message. Note that MIDI program change messages may not be used for the set switch, the map switch, or tabs, because these uses require an *off* as well as an *on* signal.

Outputs

In order to incorporate a combination action with dual-magnet tabs, two outputs are needed for each tab. Up to 96 bits of output are available from each Micro-Midi module. Thus, for a system with up to 48 tabs, one Micro-Midi module will be sufficient. For each additional 48 tabs, one more Micro-Midi module will be necessary.

Magnet drivers

To specify that a Micro-MIDI module is connected to a driver board for dual magnet tab stops, supply the following:

```
*UMIDI_MODULE
*RANK_DRIVER=mag1
```

This associates the name “mag1” with this set of outputs.
This is all there is to it. The individual connections will be specified in the *COMBINATION_ACTION section, coming up later.
Also available are high-current HD-32 driver boards. In order to use these boards, specify (for example) *HD32=mag1. Unlike rank driver boards, HD32 boards may be cascaded, up to four boards for 128 outputs.

Individual Bit Outputs

An individual bit can be controlled by using the *OUTPUT_BIT keyword. An expression in parentheses specifies when this output should be on, and the *BIT keyword specifies which output bit should be controlled. This should be located under the *RANK_DRIVER section which has the output being controlled, and it must be before any *RELAY statements. An example:

```
*rank_driver=mag1
*output_bit(bass_drum) *bit=7
```

When the expression (bass_drum) is non-zero, this output bit will be on. When (bass_drum) is zero, this output will be off. See section Expressions.

Combination Action

In order to configure the combination action, a *COMBINATION_ACTION section needs to be included after all of the *UMIDI_MODULE sections.
There are two special pistons used for the combination action. The optional MAP (or RANGE) piston is used to establish a group of stops that will be used for subsequent registration.
The SET piston is used to “set” the desired registration within the mapped area for each individual piston.

This will be discussed further later in this section.

The combination action section starts off with the *COMBINATION_ACTION keyword. The set and map controls are next specified as follows:
*SET=setInput
*MAP=mapInput
Here, setInput and mapInput need to have been defined earlier in the configuration file in an *INPUT_BIT declaration. Following this, all the piston inputs should be named:
*PISTON=Solo1
*PISTON=Solo2
and so on. Again, Solo1 and Solo2 need to have been defined as *INPUT_BITs above.
If it is desired to have more than one memory bank for the combination action, a controller may be used to select the memory bank. This is done like this:
*memory_select=memorySelectController
memorySelectController must have been defined previously in a
*control=memorySelectController
definition. The controller should be defined so that it generates values from 0 through n, where n is the number of memory banks. If the controller generates a value which is larger than the number of memory banks which will fit into the available memory, the highest numbered memory bank which is available will be used. [See memory determination below]
When using Lighted stop boxes, that's all that is required.
When using dual-magnet drivers, all the magnets must now be specified as follows:
*DUAL_MAG_TAB=diaphone_16 *ON=mag1:3 *OFF=mag1:4
What this is saying is that on the *RANK_DRIVER named mag1 (which must have been previously defined), output number 3 is connected to the ON magnet, and output number 4 is connected to the OFF magnet of the diaphone_16 tab (which must have been previously defined in an *INPUT_BIT).
When the system is first put into operation, all pistons will be set to turn on all stop tabs. To set a piston, set the stop tabs to the desired setting, press and hold the SET button, and then press the piston which is to be set.
If a MAP button has been implemented, then the map or range of each piston may be set. By unmapping a tab from a particular piston, that piston will not affect that tab. To set the map for a piston, turn on all tabs which are to be affected by the piston, press and hold both the MAP and SET buttons, and then press the piston which is to be mapped.
In order to view the mapping of a piston, press and hold the MAP button. Then, pressing the desired piston will set all tabs to display the mapping of that piston.
There are about 4000 bytes of memory available in each Micro-MIDI module which drives tabs for piston settings. Each tab requires two bits, but the result is rounded up to a power of two. Thus, for an organ with 65 tabs and 30 pistons: If 32 tabs are put on one driver board, and 33 on another, then on the second driver, 66 bits are required, or 9 bytes. This is rounded up to the next power of two, or 16 bytes (Lots of wasted memory!). Thus, 16*30 or 480 bytes of memory are required. This leaves room for about eight memory banks. The other driver board, with one fewer tabs, requires half as much memory, and can thus accommodate about 16 memory banks.

When dual magnet tabs are used, the default is for the driver board to activate its outputs for 100 mSec. To change this, use a command such as:
*DUAL_MAG_TIME=50
Valid numbers are 1 through 255 mSec. This command may go anywhere in the
*combination_action section.
*DUAL_MAG_TAB may be abbreviated as *DMT.

Reversibles

Reversibles are closely coupled to the combination action, and can be used only when a combination action is defined. A reversible uses some input bit to cause a stop tab to change into its opposite state. For example, if you have a toe stud which you wish to use to turn the bass drum on and off, you would do the following.
First, you need to connect up the toe stud input. In the appropriate `*hv64` section, you would include a line such as

```
*input_bit=toe_drum *bit=17 c15n7
```

Then, in the combination action section, directly under the `*dual_mag_tab` entry for the bass drum, you would include

```
*reversible = toe_drum
```

That's all it takes!

Relay

If the micro-midi system is to be driving a synthesizer or sound engine which doesn't know how to respond to tab stops, a module can be used to implement a relay function. A relay function allows one or more keyboards to be routed to selected MIDI channels with optional pitch offsets, depending upon the state of selected input bits, such as a lighted stop box or tabs connected to an input board.
The template for a relay definition is:

```
*UMIDI_MODULE
*RELAY (<expression>)
*DIVISION = <division name>
*MIDI_CHANNEL = <channel>
*MIDI_OFFSET = <offset>
*VELOCITY (<expression>)
```

<expression> determines when this relay statement will be active. It is often just the name of a stop tab, which must have been defined previously.
<division name> identifies the source of the note information. It must be previously defined.
<channel> is the desired output channel.
<offset> allows the output to be offset from the input; for example, if <offset> is set to 12, the output will be one octave higher than the input. This line is optional if the offset is zero.
The *VELOCITY line is optional. Normally, note on commands are sent with a velocity of 127. A particular relay may request a different velocity with this line. The expression may be a constant such as (63), or a controller such as (c1c7), or a more complicated expression. The expression should evaluate to a number between 1 and 127, or the results may not be what you expect. If it evaluates to zero, the note will not sound.
An example might look like this:

```
*umidi_module
*relay (tibia_8) *division=swell *midi_channel=1
*midi_offset=0
*relay (tibia_4) *division=swell *midi_channel=1
*midi_offset=12
*relay (flute_8) *division=swell *midi_channel=2
*velocity=(c1c7)
```

and so on.
The <expression> may be any expression, as defined in section Expressions. If that expression is true, or

non-zero, the relay statement will be active. If the expression is false, or zero, the relay statement will be skipped.
For most purposes where a single tab name is not selective enough, an AND function is needed. This is specified by separating tab names with ampersands (&). In addition, if a tab name is prefaced with an exclamation mark (!), that tab must be off for the expression to be true.
An example should make this clear:

```
*relay (tibia_8 & !trem) *div=swell *mch=1 *mof=0
*relay (tibia_8 & trem) *div=swell *mch=2 *mof=0
```

In this example, when the tibia_8 tab is on, note information from the swell division is routed to MIDI channel 1 if the trem tab is off, or to MIDI channel 2 if the trem tab is on.
What if the division information is arriving on the same channel that will be used as an output? No problem. The system automatically swallows note information on any channels which are named as output channels in `*RELAY` statements. In the above case, note information arriving from MIDI on channels 1 or 2 would not be forwarded directly to the output. Controllers and other information on channels 1 and 2 would, however, be forwarded from the input to the output.
The forwarding of MIDI input may be controlled in more detail. For example, if channel 12 is not named as an output in a `*RELAY` statement, but note information arrives on channel 12 (for example, stop tabs), that information will be sent to the output. In order to swallow all information arriving on MIDI channel 12, and not forward it to the output, use the statement

```
*MIDI_CHANNEL_TRUNCATE = 12,0
```

See *midi_channel_truncate in section Keywords for more details.

Rank Outputs

Closely related to `*RELAY` definitions are rank definitions. In fact, they start out looking identical. A rank definition allows a MIDI stream to control an output driver board, for example to drive a pipe chest. The rank output is implemented in two steps: First, define the divisons, ranks and tabs needed. Then, specify how the divisions and tabs are to be used to drive the rank. Examples of divisions and tabs have been seen above. A rank definition is done under the output board (for example, a `*RANK_DRIVER` definition) which drives the pipe chest. For example,

```
*rank_driver
*rank=flute *bits=1,61
```

The `*RANK` definition connects the given name (flute) to a series of bits on the rank driver board, in this case 61 bits starting with bit 1.
Now, add a `*RELAY` definition, substituting the rank for the MIDI commands:

```
*relay(great_flute_4)
*division=great
*rank=flute
*offset=12
```

Here, the (great_flute_4) specifies when these notes should drive the rank, the `*division` specifies the MIDI notes which are to drive the rank, and the `*rank` specifies which outputs should be driven. Without the optional `*offset` specification, the lowest note on the great division would drive the lowest note on the flute rank. In this example, the `*offset` specification causes a transpose of 12 notes up, or one octave.
In order for a rank output to work correctly, the `*rank` must be defined before it is used, and it must be used under the same `*umidi_module` as the rank definition.
If only a portion of the rank is desired you may add the delimiters after the *rank=flute, 1, 12

This would play only the lowest 12 notes of that particular rank making it possible to borrow notes from one rank to another. Note the commas! There must be one after the rank name and another after the first note to be played then a space and the number of the last note to be played.

Melody Stop

A melody stop is a special stop in which only the top note played sounds. The top note of a chord is often the melody of whatever is being played, and this allows a special stop to automatically emphasize the melody of a performance.
A melody stop is implemented just like a relay or rank output, by replacing the `*relay` with `*melody`.

Bass Stop

A bass stop is a special stop in which only the bottom note played sounds
A bass stop is implemented just like a relay or rank output, by replacing the `*relay` with `*bass`.

Trap Stop

A trap stop is a special stop that has only one output note. Usually, that note is a special sound effect, or trap. The single note is played whenever one or more notes on the source division are played.
A trap stop is very similar to a relay or rank output, with two differences. First, the `*relay` is replaced with `*trap`.
Second, the syntax changes slightly to specify a single note instead of a rank or range of notes. See the following examples.
For a relay trap, where the output is a MIDI note, a trap stop might look like this:

```
*trap (g_triangle) *division=great *midi_channel=10
*midi_note=8
```

Note that an extra parameter has been added to specify which note on the selected channel is to turn on. In this case, it is assumed that channel 10 note 8 is a triangle. When the g_triangle stop is engaged, playing any note on the great division will sound the triangle.
For a rank trap, where the output is a rank driver board, a trap stop might look like this:

```
*trap (g_triangle) *division=great *rank=traps *bit=4
```

In this case, it is assumed that a rank named ‘traps’ has a triangle connected to its fourth output. When the g_triangle stop is engaged, playing any note on the great division will sound the triangle.
*MIDI_CHANNEL = 10 *MIDI_NOTE = 8 may be abbreviated as c10n8

Implementing Reiteration

Reiteration can be implemented in the Micro-MIDI system in a rather roundabout way. In most cases, a Micro-MIDI module must be dedicated to the processing of the reiteration, and this module must be placed between the source of the division information and the destination of the reiterated notes. Reiteration is implemented by looking for notes on one particular MIDI channel, and then sending out like notes on a different channel while the note is being held.
In many cases, it is desired to have the notes of a reiterated stop to repeat slower for lower notes and faster for higher notes. The Micro-MIDI system allows different speeds to be specified for

the bottom and top notes of a division, and the speeds of notes in between will vary between those values, with lower notes repeating slower and higher notes repeating faster.
The repeat speed must be between 4 and 100 times per second, and the speed must be slower for low notes than for high notes.
When reiteration is operating, it generates many MIDI messages. In order to keep down the extra chatter, the generation of reiteration can be turned off if desired.
A reiteration command looks like this:

```
*REIT=swell_reit (enable_reit) *DIVISION=swell
*MIDI_CHANNEL=3 *RATE=10,20
```

In this example, the swell division is reiterated into a new division called “swell_reit”. This reiteration only occurs when the signal “enable_reit” is true. The reiteration occurs at the rate of 10 beats per second at the low end of the swell division, and 20 beats per second at the high end of the swell division.
The signal “enable_reit” can now be used in downstream modules to enable certain ranks to replace the swell division with the swell_reit division, thus causing them to reiterate.

Implementing Pizzicato

Pizzicato is implemented in a manner very similar to reiteration. The differences are that when a note is played, it is only sounded once in the output division. Also, the length of the pizzicato is specified in milliseconds instead of beats per second. The valid lengths are 1 through 255 mSec. The length for the lowest note must be at least as long as the length for the high note.
A pizzicato command looks like this:

```
*PIZZ=swell_pizz (enable_pizz) *DIVISION=swell
*MIDI_CHANNEL=4 *DELAY=100,50
```

In this example, the swell division causes pizzicato notes to sound in MIDI channel 4, which is named swell_pizz. The lowest note of the division sounds for 100 mSec, and the highest note sounds for 50 mSec.
In most cases, pizzicato and reiteration will be implemented in a Micro-MIDI module to itself, although they could likely share with some inputs (if those inputs were not related to the reiteration and pizzicato).

Implementing Sostenuto

Sostenuto can be implemented in a Micro-MIDI system. Sostenuto may be implemented in a Micro-MIDI system by using an input (or some other condition) to enable a hold of notes on a particular division. When the sostenuto is enabled for a division, any notes which currently being played on that division are latched, and will stay on no matter whether the key is held or released. Other notes are not affected, and will play and release normally. When the sostenuto is disabled, any previously latched notes which are not being played at the moment will turn off immediately. Sostenuto requires a fair amount of resources, and will usually require a module by itself. Sharing with pizzicato and reiteration will likely work.
A sostenuto command looks like this:

```
*SOSTENUTO (sostenuto_switch) *division=swell
```

When the input sostenuto_switch becomes true, this will cause the note on and note off information for the (previously defined) swell division to be modified appropriately.

Couplers

Couplers require a lot of system resources, and hence require a Micro-MIDI module all to

themselves. This module should go between all the note and control inputs, and any relay or other output modules. Couplers are closely related to transpose; see the next section. If a transpose is specified, couplers must be specified for all divisions, as the transpose will only be applied to divisions mentioned in a coupler statement.
A coupler statement looks like this:

```
*coupler (<expression> <from_division> to <to_division>
[*offset = <note_offset>]
```

The expression may be the name of a coupler tab, or it may be (1) if the coupler is to be true always. The <from_division> is the name of the division on which the coupled notes are to sound. "to" is just the word "to" exactly as shown. The <to_division> is the name of the division that the note on/off information comes from. The brackets [..] indicate that the *offset part of the statement is optional, and need not appear if the <note_offset> is zero. The <note_offset> specifies how much higher the output of the coupler should be than the input.
For the first example, a unison coupler:

```
*coupler (1) solo to solo
```

This specifies that the solo division should always sound in unison. Without this statement, no unison notes will be heard on the solo division. If you have a unison off tab for solo, this might look like

```
*coupler (!solo_unison_off) solo to solo
```

An octave coupler would be

```
*coupler (solo_to_solo_4) solo to solo *offset=12
```

And a cross-division coupler might be

```
*coupler (solo_to_great_16) solo to great *offset=-12
```

Cross-division couplers between divisions with different base notes (`*midi_notes=x`,…) are implemented so that an offset of zero causes the same note number to sound as the input note. Couplers with offsets may cause notes to be generated which are outside the specified range of the "to" division. This extension will be between 0 and 14 notes, and depends upon a number of factors including the transpose and the offset, and the number of notes in the division(s).

Transpose

Transpose is tightly coupled with couplers, and should be specified in the same Micro-MIDI module as your couplers. In order for transpose to function, at least one coupler must be specified for each division that needs to be transposed. In addition, the amount of the transpose is specified thusly:

```
*transpose (<expression>)
```

Suppose you have a controller number 17 on channel 12, which has values from 1 to 13, where 1 is to represent a transpose down of 6 semitones, through 13 representing a transpose of up 6 semitones. And, just in case the value of that controller has not been seen by the code, we want the default controller value of zero to result in no transpose. The transpose statement might then appear

```
*transpose ( (c12c17 != 0) * (c12c17 - 7) )
```

Now, if channel 12 controller 17 (c12c17) is zero, the first term is false, which is zero, the whole expression is zero, and there is no transpose. If c12c17 is non-zero, the first term is true, which is one, and the transpose is c12c17 – 6.
If no couplers are desired, but transpose is, specify unison couplers for all used divisions:

```
*coupler (1) solo to solo
```

Setting Velocity

This is a feature which works rather strangely, but may be useful in some rare situations. Normally, notes which are generated by the Micro-MIDI system have a velocity of 127 for notes turning on. In most cases, this is reasonable for organ applications. The *velocity keyword allows the velocity generated by a Micro-MIDI module to be controlled.
The syntax is:
```
*VELOCITY (<expression>)
```
From the point where this is encountered, any notes generated by the module where this statement is will have the velocity of the expression. The expression can be a fixed number such as (25), a controller such as (c3c7), or a more complicated expression.
This velocity will apply to any notes generated by the module. This includes reiterated notes, pizzicato notes, and all sostenuto notes.

Initial Declarations

Initial declarations can be placed before the first `*umidi_module` statement to specify messages which will be flowing into the Micro-MIDI system via the MIDI input in the first module. This section covers these statements.

What if the division to be used in a `*relay` definition is actually an external MIDI keyboard which is plugged into the first Micro-Midi module? There needs to be a way to specify this in the `*relay` statement. The solution is to put an initial declaration at the beginning of the configuration file, defining the external input to the Micro-Midi system. The statement for this looks like this:
```
*DIVISION=swell *MIDI_CHANNEL=1 *MIDI_NOTES=36,61
```
The `*relay` definition can now refer to the “swell” division, and uConfig will know to look for notes on channel 1. Note that the `*division` statement is also used with a slightly different syntax to create a division from inputs. This is covered in section *Keyboard inputs*.
Similarly, an external input which is a single note can be defined as follows:
```
*input_bit=wahwah *midi_channel=8 *midi_note=12
```
An external input which is a program change can be defined as follows:
```
*input_bit=piston3 *midi_channel=9 *program_change=17
```
And, an external controller can be defined as follows:
```
*control=doodah *midi_channel=10 *midi_controller=7
```

Sending MIDI Messages

It is possible to send an arbitrary MIDI message when some condition is met. For example, one might wish to use an input to cause an “all notes off” MIDI message to be transmitted when a panic button is pressed. At this point, the features of this section cannot be implemented in an input module, so at least two modules are necessary: one to act as an input module and sense the switch, and the other to generate the MIDI message.

Notes Program changes and Controllers

An arbitrary MIDI note on message may be sent when an expression goes from zero to non-zero:
```
*midi_command (expression) c3n17, velocity
```
Here, `(expression)` can be any legal expression. When its value first evaluates to a non-zero number, or when it changes from zero to non-zero, then note 17 of channel 3 will be transmitted

with a velocity of `velocity`. `Velocity` may be any number from 0 to 127. Note that a note on message with a velocity of zero is equivalent to a note off message. Here is an example:

```
*midi_command (crescendo) c8n5, 127
*midi_command (!crescendo) c8n5, 0
```

Assuming that `crescendo` is a controller, this pair of commands will send a channel 8 note 5 on message when `crescendo` is non-zero, and a corresponding note off message when `crescendo` returns to zero.

Similarly, program change messages may be sent:

```
*midi_command (expression) c4p7
```

Or, controller messages may be sent:

```
*midi_command (expression) c5c17, value
```

Unfortunately, `value` must be a fixed number, and can not be an expression.

Fixed MIDI messages

The following MIDI messages may be sent when the expression in parentheses goes from zero to non-zero:

```
*midi_command (expression) *system_reset
```

This sends a SYSTEM RESET (ff hex or 255). The module sending this message will not respond to it, but any subsequent Micro-MIDI modules will go through their power-up reset sequence when they receive this command.

```
*midi_command (expression) *all_notes_off channel
```

This sends a controller 123 value 0 message

```
*midi_command (expression) *local_control_off channel
```

This sends a controller 122 value 0 message

```
*midi_command (expression) *local_control_on channel
```

This sends a controller 122 value 127 message

```
*midi_command (expression) *omni_mode_off channel
```

This sends a controller 124 value 0 message

```
*midi_command (expression) *omni_mode_on channel
```

This sends a controller 125 value 0 message

```
*midi_command (expression) *mono_mode_on channel, nchannels
```

This sends a controller 126 value nchannels message

```
*midi_command (expression) *poly_mode_on channel
```

This sends a controller 127 value 0 message

Here, (expression) may be the name of an input bit, such as (panic_button). Channel must be the channel number (from 1 to 16) of the channel on which the message is to be sent. For the *mono_mode_on message, nchannels is the third byte of the message, normally between 0 and 16.

Sending arbitrary MIDI messages

An arbitrary MIDI message may be sent when an expression becomes non-zero

```
*midi_command (expression) b1
*midi_command (expression) b1, b2
*midi_command (expression) b1, b2, b3
```

Here, (expression) is the expression which causes the MIDI message to be sent when the expression goes from zero to non-zero. For a one-byte message, b1 is the status byte to send. For two or three byte messages, b2 and b3 are the data bytes to send. Note that in order for the system

to work properly, the number of bytes must agree with the value of b1. This is not enforced by the software – user beware! In all cases, 128 ≤ b1 ≤ 255, 0 ≤ b2 ≤ 127, and 0 ≤ b3 ≤ 127, and this is enforced. b1, b2, and b3 may be expressed in decimal, or in hexadecimal by prepending 0x. For example, 0x80 is the same as 128.

Expressions

Expressions are used in `*relay` and `*if` statements, to select which `*relay` statements should be active at any time. Expressions are also used in `*OUTPUT_BIT` definitions. An expression is one or more constants and variables combined with arithmetical operators. Order of computation may be controlled using parentheses. The Micro-MIDI board computes the expression when needed, and then acts accordingly.
All evaluation is done using 8-bit unsigned arithmetic. This means that only the values 0 through 255 are valid, and any attempt to compute a value (including intermediate values) outside of this range will result in an overflow. In most cases, the result of this overflow can be predicted, and may be useful.

Constants

Constants may be integers from 0 to 255.

Literals

As a convenience, literals may be substituted for variables in some cases. A literal names a specific note, channel, or controller directly. This usage is normally discouraged, because it bypasses some of the error checking facilities of the Micro-MIDI system, and it removes the self documenting nature of using descriptive names for signals. However, if you insist, here's how to do it.
To refer to a specific MIDI note, use "CnNm", where n should be the channel number, and m should be the note number. Thus, C1N23 would refer to MIDI channel 1, note 23. Capitalization is ignored: c1n23 or C1n23 would do just as well.
Similarly, to refer to a specific MIDI program change, use "CnPm", where n is the channel number and m is the program change number. For example, C3P17.
Similarly, to refer to a specific MIDI controller, use "CnCm", where n is the channel number and m is the controller number. For example, C1C7.
These literals may also be used as abbreviations for the excessively wordy syntax. For example,
*midi_channel = 12 *midi_note = 17
can be replaced with c12n17.

Variables

A variable is a reference to a previously defined input bit or controller. In the case of an input bit, the variable has a value of 1 if the bit is on, or 0 if the bit is off. In the case of a controller, the variable has the value of the controller.

Arithmetic operators

These are the operators allowed, in order of decreasing precedence
! Logical not. If x is 0, !x is 1. If x is not zero, !x is 0
~ Complement. ~x is the one's complement of x, which is the same as 255-x

* Multiplication. If the product exceeds 255, the low-order 8 bits of the product are used.
/ Division. The remainder is lost. Division by zero results in an undefined number.
% Modulo: Remainder after division. Thus, 5%3 is 2, because 5/3 is 1 with a remainder of 2
+ Addition
- Subtraction
<< Shift left. x << 2 is x shifted left two places, equivalent to multiplying by four. Note that multiplication is usually faster.
>> Shift right. x >> 2 is x shifted right two places, equivalent to dividing by four. Note that division is usually faster.
MIN Minimum. x MIN y has the value x if x is less than y, otherwise it has the value y.
MAX Maximum. x MAX y has the value x if x is greater than y, otherwise it has the value y.
< Less than. x < y has the value 1 if x is less than y, otherwise it has the value 0.
<= Less than or equal to. x <= y has the value 1 if x is less than or equal to y, otherwise it has the value 0.
> Greater than. x > y has the value 1 if x is greater than y, otherwise it has the value 0.
>= Greater than or equal to. x >= y has the value 1 if x is greater than or equal to y, otherwise it has the value 0.
== Equal to. x == y has the value 1 if x is equal to y, otherwise it has the value 0. (That's two equal signs next to each other)
!= Not equal to. x != y has the value 1 if x is not equal to y, otherwise it has the value 0.
& Binary AND. x & y has the value of the binary AND of x and y.
^ Exclusive or. x ^ y has the value of the exclusive or (XOR) of x and y.
| Binary OR. x | y has the value of the binary OR of x and y.
As an example, if you wish a relay statement to be active when the swell_diap tab is on and the my_controller controller has a non-zero value, you could write

```
*relay(swell_diap & my_controller > 0)
```

Since > has a higher precedence than &, the > will be evaluated before the &. (my_controller > 0) will be true, and have a value of one, when my_controller is non-zero. Thus, the whole expression will be true when swell_diap is on and my_controller is non-zero.
Surplus parentheses make it clear in what order operations occur, and don't cost any execution time. Thus, it might be better to write the above as

```
*relay(swell_diap & (my_controller > 0))
```

IF Statements

Groups of `*relay` statements (and other `*if` statements) can be enabled with a single expression by using an `*if` statement. This is advantageous if there are a number of `*relay` statements which can sometimes be skipped with a single test, because this increases the execution speed of the downloaded program.
The form of the `*if` statement is

```
*if (<expression>) { <statements if true> }
```

An `*else` clause may also be used, like this

```
*if (<expression> { <statements if true> } *else {
<statements if false> }
```

Implementing Crescendo

There are no special keywords for implementing a crescendo function. Instead, *if and *relay statements can do the job. Here's an example. Assume that *crescendo* is a controller which is zero when crescendo is off, and varies from one to (say) 12. The first *if statement saves the Micro-MIDI board from having to evaluate all the *relay statements if the value of *crescendo* is zero.

```
*if (crescendo) {
*relay ... ;stop for crescendo >= 1
*relay(crescendo>1) ... ;stop for crescendo >= 2
*relay(crescendo>2) ... ;stop for crescendo >= 3
*relay(crescendo>3) ... ;stop for crescendo >= 4
; etc.
}
```

Blind Combination Action

For consoles without computer control of the stop tabs, the Micro-MIDI system can implement a blind combination action. With blind combination action, when the organist operates a piston, there is no visible indication that the piston has activated, but nevertheless, MIDI messages are transmitted to change to the desired registration. At this point, operating stop tabs will modify this registration, but will not return it to agree with the stop tabs. A cancel piston is supplied, however, which will send the appropriate MIDI messages to make the registration agree with the stop tabs.

Let's describe how the blind combination action appears to the organist. In addition to a number of pistons, there is a special "cancel" piston, and a "set" switch. To set a piston, the organist presses the cancel piston (to make sure he has a clean slate), sets the physical switches in the desired configuration, and then while holding the set switch, depresses the piston he wishes to set. This may be repeated for any desired pistons.

Now, pressing a piston will set the registration to equal that piston. This is now not the same as the physical tabs. Some tabs may be physically off, but the corresponding stop enabled. If the organist turns one of these tabs on, nothing will happen. Turning the tab off again will cause that stop to be disabled, without affecting any other stops in the organ. Thus, starting from a piston setting, the registration can be modified incrementally.

At any time, the organist may press the cancel piston to return the registration so that it agrees with the physical setting of the switches.

After setting a number of pistons, the organist may wish to modify one of them. To do this, he should press the piston he wishes to modify, then alter the registration as desired, and then, while holding the set switch, press the piston which he desires to have the current registration.

To implement a blind combination action, a Micro-MIDI module must be dedicated to that function. It must be located after the inputs for the pistons, tabs, and set and cancel switches; and it must be located before any modules which are affected by tab stops.

The module program has the following outline:

```
*umidi_module
*blind_combination_action
*set = <set switch>
*cancel = <cancel switch>
```

```
*piston = <piston switch>
*tab = <tab switch>
*memory_select = <memory select controller>
```

Here, <set switch> should be replaced with the name of the input which is connected to the set switch.
<cancel switch> should be replaced with the name of the input which is connected to the cancel switch.
<piston switch> should be replaced with the name of a piston, and the whole line repeated for each piston.
<tab switch> should be replaced with the name of each tab which is to be controlled by the combination action, and the whole line repeated for each tab.
The optional *memory_select specifies the controller which is used to select the desired memory bank. See the Combination Action section for more details.
An example might appear

```
*umidi_module
*blind_combination_action
*set = set
*cancel = cancel
*piston = piston1
*piston = piston2
*piston = piston3
; etc
*tab = swell_diapason_8
*tab = swell_flute
*tab = swell_tuba
; etc
```

One Of N Output

The One of N output feature is a special-purpose function designed to go with the blind combination action. Its purpose is to give an indication of when a blind combination action override of the registration is in effect. The idea is to have one lamp (perhaps inside the piston button) for each piston, and light the lamp whenever that piston is active. This requires an output board connected to a separate Micro-MIDI module; the Micro-MIDI module which implements the blind combination action can not be used.
In action, pressing any piston will cause the corresponding lamp to light. Pressing any other piston will cause that lamp to go out, and the new appropriate lamp to light. Pressing the cancel button turns out all lights. When a lamp is lit, operating stop tabs will change the registration, but the same lamp will continue to be lit.
To implement this feature, use the following template:

```
*umidi_module
*rank_driver
*one_of_n *bits = <first>,<count>
*cancel = <cancel button>
*set = <piston button>
```

Here, <first> is replaced with the bit number of the output on a rank driver board which is to be associated with the first lamp; <count> is replaced with the number of bits to reserve for this function; <cancel button> is replaced with the name of the cancel button, and <piston button> is replaced with the name of the first piston. The *set line is repeated for each piston, in the order

that the outputs are to be associated with the pistons.
For the above blind combination action example, the following would be appropriate:

```
*umidi_module
*rank_driver
*one_of_n *bits = 1,10
*cancel = cancel
*set = piston1
*set = piston2
*set = piston3
; etc.
```

The module must be downstream from the outputs it is observing, but could be upstream from the blind combination action if desired. The module can also be used to drive other outputs on the same rank driver board.

Serial LCD Display

Artisan Instruments makes available a 4-line by 20 character LCD display that can be controlled by the Micro-MIDI system. To control this display, a dedicated Micro-MIDI module is required. This module should be placed in the MIDI stream after any inputs to which it must respond.

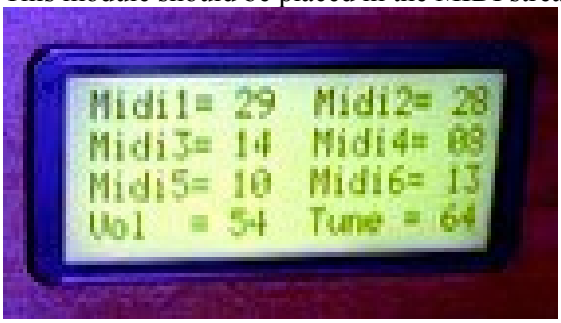

Illustration 10: Liquid Crystal Display

Simple Example

This example can be used as a template for a simple display.

```
*umidi_module
*serial_lcd
"\004\020\024"
*line 1 "MOLLER PIPE ORGANS"
*decimal 1
*line 2 "TRANSPOSE = " (transpose-6)
*unsigned *decimal 2
*line 3 "MEMORY LEVEL = " (memory_level)
*line 4 "CRESCENDO = " (crescendo)
```

This results in a display like this (colors are not correct):
Let's go over that in detail.
`*umidi_module` introduces a new module.
`*serial_lcd` specifies that this module will be used to drive an LCD.

`"\004\020\024"` are a special commands to the display to tell it to hide the cursor, and to turn off wrap and scroll. Without this, the cursor can flash intermittently and other strange things can happen, which can be disconcerting. It's probably a good idea to always include this in your definition, unless you have a reason not to.
`*line 1` tells the system that the following information is to be placed on line 1 of the display. Anything in quotes is treated as a text string, and will be written to the display at the current position. In this case, "MOLLER PIPE ORGANS" will appear on the first line.
`*decimal` 1 is a formatting command. More about that later.
`*line 2` introduces the second line. As before, the following text will be placed on the line. Next, comes an expression in parentheses. Assuming that transpose is the name of a controller which represents the transpose level, we wish to display this information. We're assuming here that transpose has a value of 6 when there is to be no transposition, with values from 0 through 5 representing transposes downward, and 7 through 12 transposes upward. By subtracting 6, we get a number which represents the actual magnitude of the transposition. Since this number is never less than -9 or greater than 9, only one character (in addition to the potential minus sign) is needed for the number. The `*decimal 1` command causes only one character position to be occupied by the number. Without this, three spaces would be reserved, and there would be a big gap.
`*unsigned` specifies that from here on out, all numbers will be positive, and a character position will not be reserved for a possible negative sign. The default is `*signed`, so on line 2, the transpose number always takes up two character positions. The first character is either a space or a minus sign.
`*decimal 2` specifies reserves two spaces for numbers from here on out.
`*line 3` and `*line 4` introduce no new features.

Available Commands

Now it's time to spell out all of the available commands for controlling the LCD. Some of this gets pretty involved, so if you're happy with a simple display as above, you don't need to hurt your head with the following. If you're really into doing everything possible, you may wish to read the data sheet for the display. It may be found at
http://www.crystalfontz.com/products/634/634_data_sheet.html

Text

Any characters enclosed in quote marks (") are sent to the display at the current position. There are some enhancements to allow characters which it wouldn't otherwise be possible to send. In order to send any arbitrary character to the display, write its decimal value as three digits, preceded with a backslash (\). For example, to send the character " to the display, use the characters `\034`, as in `"Here is a quote:\034"`. To send a backslash, place two backslashes in a row. One will be removed when the string is parsed.

Expression

Any expression enclosed in parentheses is evaluated and sent to the display. The manner in which it is sent is controlled by format commands, described in the next section.

Formatting Numbers

When an expression is evaluated and sent to the display, it will take from one to four character

positions in the display. The commands in this section will control all expressions from the point at which the command appears in the configuration file until a subsequent format command changes it. Note that *if commands don't affect format commands.
*signed, which is the default, reserves one character space for either a minus sign if the expression is negative, or a space if the expression is positive. It is possible to display numbers from -128 to +127.
*unsigned causes subsequent expressions to be evaluated as unsigned numbers. A character space is not reserved for the sign, and the displayed numbers may be from 0 to 255.
*decimal 3, which is the default, reserves three character positions for the number. This is in addition to any character position which may be reserved for a sign. Numbers less than 100 are displayed as a space followed by two digits; for example " 01" for one.
*decimal 2 reserves two character positions for numbers. For numbers greater than 99, the leading "1" or "2" is lost.
*decimal 1 reserves one character position for numbers. For numbers greater than 9, only the last digit of the number is displayed.
*binary is a special mode, which may be necessary to access some of the more esoteric features of the display. After *binary, any expressions will be evaluated, and the result will be sent to the display as a single binary byte.
*decimal is an alias for *decimal 3

Positioning Line

*line 1 causes the display cursor to be placed at the first character position of line 1. Similarly, *line 2, *line 3, and *line 4 place the cursor at the beginning of lines 2, 3, or 4.

Controlling the Display Contrast

If your display is washed out or too dark, you may need the *contrast command. You may wish to try *contrast (60). If this is too dark, try a smaller number. If it is to light, try a larger number. The number in the parentheses can be an expression, such as a controller, if you wish. The number must be between 0 and 100, although useful numbers are likely to be near 50.

Controlling the Display Backlight

To control the brightness of the LED backlight, use the *backlight command. For example, *backlight (50) would set the brightness to half. If the display is too bright, use a smaller number. If the display is too dim, use a larger number. The number should be between 0 and 100. The number may be replaced with an expression, such as a controller, if you wish.

Bar Graphs

The display has a feature which allows you to display a horizontal bar graph. This bar graph may occupy any row, and from 1 to 20 characters on that row. Its length may be set to any number of pixels, from zero to six times the number of characters in the bar graph. To do this, specify
*bar_graph <row> <first_character> <last_character> (bar_length)
where row is the row number, from 1 to 4
first_character is the first character position, from 1 to 20
last_character is the last character position, from 1 to 20
(bar_length) is an expression.

Example:
```
*line 4 "Solo " *bar_graph 4 6 20 (solo_swell * 9)
```
This will place the text "Solo " at the beginning of the fourth line, occupying five characters. The next 15 character positions (characters 6 through 20) are now used for a bar graph. 15 characters have a length of 90 pixels, so the expression should be between 0 and 90. As the controller solo_swell assumes the values of 0 through 10, the bar will grow from zero length (invisible) to full length.
Note that if the expression evaluates to a negative number, the bar will be drawn from the right side of the allocated region and grow toward the left.
If you don't know what custom characters are, the following may be safely ignored. The `*bar_graph` command uses custom characters 0 and 1 on row 1, 2 and 3 on row 2, 4 and 5 on row 3, and 6 and 7 on row 4.

Conditional Display If Statements

Sections of display code can be conditionally executed or skipped with the use of `*if` statements. The `*if` statement has the form
`*if (`expression`) {` <statements> `}`
If the (expression) evaluates to a non-zero number, then all <statements> between the braces { and } will be executed, otherwise they will be skipped. A more capable form is
`*if (`expression`) {` <true_statements> `} *else {` <false_statements> `}`

In this case, if (expression) is non-zero, all <true_statements> will be executed, otherwise all <false_statements> will be executed.
For example:
```
*line 2 "Crescendo = " *if (crescendo) { (crescendo) }
*else { "off" }
```
This causes the crescendo value to be displayed if the value of the controller crescendo is nonzero, otherwise the text "off" is displayed.
Remember that characters which are not modified stay the same. If you use an `*if` statement to sometimes display "Sforzando ON" and at other times "Sforzando OFF", you will likely see "Sforzando ONF" the first time you turn it on. To fix this, place a space after the N in ON, to clear out the F.

Once and Always

Any commands after the keyword `*once` are only executed once when the system is first powerd on, or when a MIDI reset is received. `*always` restores normal behavior for subsequent commands.

Complex Example

For this example, we will pack a large amount of information into the display, and also include bar graphs for the two swell shoes.
For this example, we assume the following:
The crescendo level is represented by a controller named crescendo.
The memory level is represented by a controller named mem_level
The transpose level is represented by a controller named transpose, which has the values from 0 to 13, representing transposes of -6 to +6.

The main and solo swell settings are represented by controllers named main and solo. Main may take on the values of 0 through 13, and solo 0 through 15.
The bit named sforzando is on when sforzando is active.
The bits named set and map are on for the set and map functions, respectively. Map by itself displays the map, while when they are both set, a map is saved.
Here is a program to display all this:

```
*umidi_module
*serial_lcd
"\004\020\024"
*unsigned *decimal 2
*line 1 "Cres=" (crescendo) " Mem=" (mem_level)
*signed *decimal 1
*if (transpose - 6) { " Tr=" (transpose - 6) } *else { " " }
*unsigned *decimal 2
*line 2 "Main=" (main) " Solo=" (solo)
*if (sforzando) { " SFZ" } *else { " " }
*line 3
*if (set | map) {
" "
} *else {
"Solo " *bar_graph 3 6 20 (solo * 6)
}
*if (!c1n49 & !c1n50) {
*line 4 "Main " *bar_graph 4 6 20 (main * 7)
}
*if (c1n49 & !c1n50) {
*line 4 "SET PISTON FUNCTION "
}
*if (!c1n49 & c1n50) {
*line 4 "MAP DISPLAY FUNCTION "
}
*if (c1n49 & c1n50) {
*line 4 " SET MAP FUNCTION "
}
```

The result may be something like
Note the special line to control the cursor, wrap, and scroll.
If a transpose is in effect, it is displayed as a one-digit signed number. If the transpose is zero, the transpose message is replaced with the same number of spaces.
There are four possibilities for the set and map bits. If neither one is on, bar graphs of the solo and main swell settings are displayed. If either one is on, line 3 is replaced with blanks, and line 4 contains a message about the mode in effect.

uCONFIG Configuration Program

This section describes the configuration download process, as well as various capabilities of the configuration program.

Software Install

The configuration program, uConfig, is installed from a windows installation program. The needed file may be downloaded from http://www.artisan-instruments.com/umidi/. Simply double-click on the file after it has been downloaded, and the installation should proceed.

Installing configuration

Now is the time to put it all together. Run the Micro-MIDI Configuration program, which should be found at “Start/Programs/Artisan Instruments/uMIDI Configuration”. When this program is run for the first time, it will search for MIDI interfaces. If none is found, it will give an error message. The likely cause of this error would be if the needed drivers were not installed. Of course, if the computer doesn’t have a MIDI interface, this would also be a problem.
Next, if there is more than one MIDI input or output device, the program will pop up a box containing descriptions of all possible devices. The appropriate device should be selected. If it’s not obvious which to use, look for descriptions which contain the words “MIDI” or “UART”.
Next, the program will ask for the configuration file (definition file). Navigate to the desired directory that contains the configuration file, and select it. If the desired file doesn’t have the extension “ucf”, select the “All Files” option to view all files.
The uConfig program will now show a box with the selections which have been made. The MIDI input and output devices are shown, along with the name of your configuration file. Any of these may be changed by clicking on the “Change” button next to the text. Note that if there is only one MIDI input or output device, it is not possible to change it.
The configuration may now be downloaded to your Micro-MIDI boards. Note that the MIDI cables need to be connected to the Micro-MIDI modules, and the modules need to have power. Click the “Download” button. This will pop up a window, where the progress of the download will be shown. If everything is in order, a message that the download has succeeded will appear. Otherwise, inspect the message and take the proper corrective action. When the message window has been read, press the “Enter” key to dismiss the window.
If there is a syntax error in the definition file, the line on which the problem was found should be indicated. The problem must be corrected, and the download attempted again. If it's not obvious where on the line the problem is, the offending line can sometimes be broken up into more lines. That is, instead of typing

```
*UMIDI_MODULE *RANK_DRIVER *NAME=mag1
```

type,

```
*UMIDI_MODULE
*RANK_DRIVER
*NAME=mag1
```

This may help narrow down the problem to a smaller amount of text.
If the program prints

```
Timeout waiting for sysex response: exiting
```

then something is likely broken in the MIDI data path. Check that all Micro-MIDI boards have power. If they do, the green LED should be blinking.

Check that all MIDI cabling is correctly installed. The MIDI OUT cable from the computer should be connected to MIDI IN on the first (leftmost) Micro-MIDI board, all the Micro-MIDI boards should be connected in order, and the MIDI OUT connector of the last Micro-MIDI board should be connected to the MIDI IN cable from the computer. Try again.
If uConfig prints:

```
Configuration successfully downloaded
```

then the Micro-MIDI boards have been configured with your program. They will retain this configuration even if power is removed. Success!!

Using Monitor

Once a configuration is downloaded, the "Monitor" button can be clicked to monitor the MIDI messages received by the computer. Inputs can be activated, and one can check that the proper MIDI messages are being generated. When through using the Monitor mode, type Ctrl/C to exit. Note that if a monitor window is left open, it will not be possible to do any other MIDI operations.

Saving Registration Information

Click the button "Save Registration to File" in order to upload all combination action data from the Micro-MIDI system and save it into a file. Supply the name of the file to save the registration in. All set and map data for all memory banks will be saved in one file.
Click the button "Load Registration from File" to restore the combination action presets to the state saved above. Navigate to the file which was saved earlier. In order for this function to operate properly, the Micro-MIDI modules must be connected to the computer in the same order, and the configuration in those modules must have the same number of memory banks, pistons, and stops as they did when the data was saved.

Updating the Executive

As new features are implemented, the executive area of the Micro-MIDI microprocessor sometimes needs to be upgraded. Since this area also contains the code which allows the Micro-MIDI module to communicate with the outside world, a chicken-and-egg situation arises. The Micro-MIDI system has a workaround for this situation which allows the executive to be upgraded, but during the time that the executive area is being upgraded, there is a small interval of time during which, if there is a power interruption or some other failure occurs, the microprocessor memory may be left in an unusable state. All this is to say that if a particular version of software is satisfying needs, don't update the executive.
To upgrade the executive, click on the "Advanced" tab of uConfig. Next, click on the "Program Executive" button. Watch the first Micro-MIDI module. The LED on all modules should be on for a while – this is the time when the code to reprogram the executive is being transferred to the first module. Next, the LED should blink four times. The first blink is the beginning of the program phase, the second blink is the end of the program and the beginning of the verify phase, the third blink is the end of the verification, and the fourth blink is the end of the cleanup. The cleanup code removes the module programming code so that it doesn't try to repeat the programming procedure. The module should then go through its normal start-up blink sequence. If the verify fails, the LED will blink slowly indefinitely. At this point, if you cycle the power to the Micro-MIDI board, the entire programming sequence will probably repeat (assuming that enough of the executive was programmed to do the startup). If it doesn't, the module will need to be repaired.

After the executive has been programmed, and the module has gone through its startup blinking, press “Enter” on the keyboard to go on to the next module.

Performing a Data Diagnostic

It’s a good idea to verify the data integrity of the MIDI loop. This can be done by clicking on the “Diagnostic” tab of uConfig, and then clicking the “Run Data Diagnostic” button. If everything is working correctly, no messages should be displayed in the window which pops up, and the LEDs on all Micro-MIDI modules should remain off. (Several seconds after the test finishes, the LEDs may begin to blink again).
For a more complete test, put a large number in the count field, and run the data diagnostic. Running for five or ten minutes with no indicated errors is a good test.
If there is an intermittent failure, the LED on each Micro-MIDI board which sees bad data should flash. With one or more Micro-MIDI boards, it may be possible to determine where the failures are occurring. If the LED on one Micro-MIDI board is not flashing, but the LED on the next board is, then a problem lies between those two boards.

Advanced techniques

On the Advanced tab, there are a few other features which may be useful. They may also be dangerous, so use with caution. In particular, clicking “Run” when no valid program is in a Micro-MIDI module will likely crash that module. A power cycle should recover from this.
Pressing “Verify Executive” will identify whether the executives of the attached Micro-MIDI modules are up to date.
Checking “Only talk to selected module” causes most commands to apply only to the module whose number is in the text box immediately below. The default number in this box is 0, which applies to the first module.
The Other Options text box is reserved for any other options which you wish to send to the program. This box should normally be left blank.
On the right side are a number of “Debug display” check boxes. Checking one or more of these boxes may display information about the internal workings of the system. It will not normally be necessary to check any of these boxes.
Below the “Debug display” check boxes is a check box labeled “Keep command window open”. Checking this box may help to figure out what is happening when the system is not working properly, and the command window disappears too quickly to read what the problem is.

Testing

The computer MIDI connectors may now be unplugged from the Micro-MIDI boards, and the other MIDI equipment substituted.
In order to verify the operation of the system first, do the following: With the MIDI IN cable of the computer still connected to the MIDI OUT connector of the Micro-MIDI system, click the “Monitor” button. This will again pop up a window, but this time it will show any MIDI messages which arrive on the computer’s MIDI IN connector.
Now, press keys on the keyboard, turn tabs on and off, and operate the combination action. The computer will display the MIDI messages which are generated by the system.
To exit from this box, press Ctrl/C. Note that if another download is attempted without exiting from the Monitor box, an error message will result.

Sample Programs

In order to make some of the concepts more concrete, this section contains several sample Micro-MIDI configuration programs. The first one is about the simplest useful program, and the last one sticks together a large system.
In order to simplify the presentation, repetition is avoided. Instead of including definitions for 45 pistons and 60 tabs for a combination action, just a few are shown. It should be obvious how to replicate and modify the sample entries.

Adding stop information to a MIDI keyboard

This sample program assumes that one has a MIDI keyboard which transmits note information on channel 1, and some kind of MIDI sound engine which accepts note information on channel 1. However, the sound engine also requires stop tab information as notes on channel 12, which allows the single channel 1 to generate multiple sounds. For this, we could either use existing switches connected to an input board, or use a lighted stop box. There will be examples with switches later, so this example will use a lighted stop box. Here's the program:

```
; There is one keyboard which is sending note information on channel 1.
; Since the Micro-MIDI system passes these notes ignored and untouched,
; no mention of them is necessary in this configuration file.
*umidi_module
*lighted_stop *count=4 *midi_channel=12 *midi_note=1
; Again, since no use is made of the outputs of the lighted stop box
; within this program, no further information is necessary
*end
```

Assuming that the lighted stop modules each have eight switches, this definition adds note on and off information for 32 stop buttons. These notes are generated on channel 12, notes 1 through 32. This is interleaved with any other MIDI information which might arrive at the MIDI IN connector of the Micro-MIDI module.

Stop information plus relay

In this sample, two complications are added to the previous sample. First, an input board is added which is connected to piston switches and a memory bank selector, and it is assumed that the sound engine is expecting note information on 16 different channels, as selected by the stop buttons in the lighted stop box. Thus, for this sample there are three Micro-MIDI modules. The first has an input board to sense switch closures for 32 pistons, two control switches, and four inputs which sense the position of a binary encoded switch with ten positions. The second Micro-MIDI module is a lighted stop box as above, but it will now respond to the piston actuations. The third Micro-MIDI board performs relay functions, taking the note information on channel 1 and routing it to channels 1 through 16, based upon the selected stops from the stop box.
As in the preceding sample program, an external MIDI keyboard supplies note information to the MIDI IN connector of the first Micro-MIDI board.

```
; Note information arrives on channel 1. This information is needed later, so the
; division must be declared at the beginning of the definition file.
; The keyboard supplies note information from 61 keys on notes 36 through 96
*division=great *midi_channel=1 *midi_notes=36,61
; The first Micro-MIDI module senses pistons and the memory bank select switch
*umidi_module
```

```
*input_64
*input_bit=piston_1 *bit=1 *midi_channel=15 *midi_note=1
; more pistons go here
*input_bit=piston_32 *bit=32 *midi_channel=15 *midi_note=32
; Define the memory bank select switch. Controller 20 is a randomly chosen value.
*control=memSelect *binary=61,4 *midi_channel=12
*midi_controller=20
; Two special-purpose switches allow the organist to set the pistons
; and to map which pistons affect which stops
*input_bit=setSwitch *bit=33 *midi_channel=15 *midi_note=33
*input_bit=mapSwitch *bit=34 *midi_channel=15 *midi_note=34
; The second module is like the previous sample, except that the
; individual bits must be defined so that they can be used for the
; relay statements
; Only a few stops are shown here
*umidi_module
*lighted_stop *count=4 *midi_channel=12 *midi_note=0
*input_bit = tib_16 *bit = 1
*input_bit = tib_8 *bit = 5
*input_bit = tib_4 *bit = 9
*input_bit = tib_223 *bit = 11
*input_bit = chrys_8 *bit = 15
*input_bit = trem *bit = 16
; Now, the third module senses the above switches and routes the
; division information to the appropriate MIDI channels.
; It is assumed that the sound engine has tibia on channel 1, tibia with
; trem on channel 2, and chrysoglott on channel 11.
*umidi_module

; Start by defining the relay mappings with the trem switch off
*if(!trem) {
; If the tib_16 switch is on, and the trem switch is off, route the great
; division to MIDI channel 1, but transpose down one octave
*relay(tib_16) *division=great *midi_channel=1 *midi_offset=-12
; Same for tib_8, except no offset is needed
*relay (tib_8) *division=great *midi_channel=1
; Same for tib_4, but now we transpose up one octave
*relay (tib_4) *division=great *midi_channel=1 *midi_offset=12
; tib_2-2/3 needs a transpose of 19 semi-tones
*relay (tib_223) *division=great *midi_channel=1
*midi_offset=19
; Now comes a closing brace to end the true part of the *if statement.
; Then, after the *else keyword, come all the *relay statements which
; should become active if !trem is not true, that is, if trem is true.
; These statements are just copies of the above, with channel 1 changed to channel 2.
} *else {
*relay (tib_16) *division=great *midi_channel=2 *midi_offset=-
12
*relay (tib_8) *division=great *midi_channel=2
```

```
*relay (tib_4) *division=great *midi_channel=2 *midi_offset=12
*relay (tib_223) *division=great *midi_channel=2
*midi_offset=19
} ; Remember the final brace to mark the end of the *else
; The chrys goes to channel 11, and we ignore the trem switch
*relay (chrys_8) *division=great *midi_channel = 11
; It might be that the sound engine would do weird things if it saw the
; stop and piston information on channels 12 and 15, so we toss it out
; in the module instead of resending it
*midi_channel_truncate=12,0
*midi_channel_truncate=15,0
; The original channel 1 notes are automatically truncated because of the first
; relay statement, which reserves channel 1 notes for another purpose.
; At the end, the combination action is specified
*combination_action
*set=setSwitch
*map=mapSwitch
*piston=piston_1
; more pistons go here -------
*piston=piston_32

; This will allow up to 16 separate banks of combination action memory to
; be selected, although if the physical switch only has ten positions, the last 6
; will be hidden and unusable
*memory_select=memSelect
*end
```

Stop switches and keyboards

For this sample, stop information and keyboard input now comes from switch closures. It is assumed that the sound engine can respond to stop information, and the `*relay` statements are not needed. Abbreviations for some of the keywords will also be used to reduce the clutter. This definition will allow up to 64 stop tabs, three 61-note keyboards and one 32-note pedalboard. HV-64N modules will be used, which are connected to bare switches.

```
*umidi_module
*hv64
; It would not be necessary to name the tabs, because the
; rest of the system ignores them. However, they must be
; listed so as to cause the system to generate the MIDI messages
*input_bit=pedal_tibia_16 *bit=1 *mch=12 *mno=1
*input_bit=pedal_tibia_8 *bit=2 *mch=12 *mno=2
; And so forth, for as many as 64 bits
; A second *hv64 would allow another 64 bits
; Here are the first two keyboards:
*umidi_module
*hv64
*div=great *bits=1,61 *mch=1 *midi_note=36
*hv64
*div=accomp *bits=1,61 *mch=2 *midi_note=36
```

```
; Up to four hv64 boards are allowed per Micro-MIDI module, but it's best
; to keep it to two
*umidi_module
*hv64
*div=pedal *bits=1,32 *mch=3 *midi_note=24
*hv64
*div=solo *bits=1,61 *mch=4 *midi_note=36
; At this point, another Micro-MIDI module could be inserted to perform
; relay functions, if this was needed.
; Otherwise, the definition is finished
*end
```

Combination Action with Dual Magnet Tabs

This example will be similar to the above, but some changes will be made for the sake of variety. HV-64 input boards will be used, of which up to 4 per Micro-MIDI module are allowed. HV-64 boards require a positive voltage of at least 5 volts applied to the input points in order to be sensed as "on".

A combination action using magnet drivers will also be implemented. This requires adding a rank driver board, which can drive up to 96 magnets (although 64 is a safer number). Program change messages will be used to encode the piston actuations. This is advantageous, because the "off" event does not have to be transmitted on MIDI, unlike a note message.

One of the HV-64 boards will also be used to generate two controller signals.

```
*umidi_module
*hv64
; Tabs
*input_bit = tib_16 *bit = 1 *midi_channel=12 *midi_note=1
; and so forth
; Special switches for combination action (must not be program changes)
*input_bit=setSwitch *bit=33 *midi_channel=15 *midi_note=1
*input_bit=mapSwitch *bit=34 *midi_channel=15 *midi_note=2
; Pistons
*input_bit=piston_1 *bit=35 *midi_channel=15 *program_change
=1
*input_bit=piston_2 *bit=36 *midi_channel=15 *program_change
=2
; etc.
; A one-of-n controller for a memory bank select
*control=memSelect *one_of_n=57,5 *midi_channel=12
*midi_controller=20
; The two controllers. Don't forget to connect the reference voltage to bit 64
*control=pot1 *bit=62 *mch=1 *midi_controller=7 *range=10,64
*control=pot2 *bit=63 *mch=2 *midi_controller=7 *range=0,127
; This definition has squished all the inputs onto one HV-64 board. To get
; greater numbers of inputs, up to four *hv64 sections could be included in
; this module, for up to 256 bits of input. Even more inputs can be
; accommodated by adding more Micro-MIDI modules.
; Now, a driver board for the magnet drivers
*umidi_module
```

```
*rank_driver=mag1
; Now all of the keyboards
*umidi_module
*hv64
*div=great *bits=1,61 *mch=1 *midi_note=36
*hv64
*div=accomp *bits=1,61 *mch=2 *midi_note=36
*hv64
*div=pedal *bits=1,32 *mch=3 *midi_note=24
*hv64
*div=solo *bits=1,61 *mch=4 *midi_note=36
;
; Relay functions could go here
*combination_action
*set=setSwitch
*map=mapSwitch
*piston=piston_1
*piston=piston_2
; other pistons listed here
*dual_mag_tab=tib_16 *on=mag1:1 *off=mag1:2
; other dual magnet tab magnet assignments here
; That's it
*end
```

Pipe chest drivers

For the finale, we'll pull out all the stops, so to speak. Let's go for an entire pipe organ with three divisions (2 keyboards plus pedalboard), four ranks of pipes, a combination action with dual mag tabs, and an external MIDI piano. We'll also add a couple couplers to show how they are done. Since this is intended to show a number of techniques without an overwhelming size, this will look like a really weird organ to those in the know. Again, the repetitive parts will be left out.

```
; Start with the input bits. Using HV-64 boards allows all inputs to
; be connected to a single Micro-MIDI module
*umidi_module
*hv64
*div=great *bits=1,61 *mch=1 *midi_note=36
*hv64
*div=swell *bits=1,61 *mch=2 *midi_note=36
*hv64
*div=pedal *bits=1,32 *mch=3 *midi_note=24
; Might as well put the pistons along with the pedalboard
*input_bit=piston_1 *bit=33 *mch=15 *mno=1
; more pistons go here
*input_bit=piston_32 *bit=64 *mch=15 *mno=32
*hv64
; Set and map for combination action
*input_bit=setBit *bit=1 *mch=15 *mno=29
*input_bit=mapBit *bit=2 *mch=15 *mno=30
; Memory select via an eight-position binary encoded switch
```

```
*control=memSelect *binary=3,3 *mch=15 *mco=27

; Now there's room for up to 59 tab inputs
*input_bit=pedal_tibia_16 *bit=1 *mch=12 *mno=1
*input_bit=pedal_flute_8 *bit=2 *mch=12 *mno=2
*input_bit=great_tibia_8 *bit=3 *mch=12 *mno=3
*input_bit=great_flute_4 *bit=4 *mch=12 *mno=4
*input_bit=great_trumpet_8 *bit=5 *mch=12 *mno=5
*input_bit=great_oboe_8 *bit=6 *mch=12 *mno=6
*input_bit=great_piano *bit=7 *mch=12 *mno=7
*input_bit=swell_to_great *bit=8 *mch=12 *mno=8
*input_bit=swell_tibia_2 *bit=9 *mch=12 *mno=9
*input_bit=swell_flute_1 *bit=10 *mch=12 *mno=10
*input_bit=swell_piano *bit=11 *mch=12 *mno=11
*input_bit=swell_octave *bit=12 *mch=12 *mno=12
; More tab inputs may go here
; Two rank driver boards are needed for more than 48 dual magnet tabs
*umidi_module
*rank_driver=mag1
*umidi_module
*rank_driver=mag2
; Another rank driver, this time driving a pipe chest with the tibia pipes
; Assume that the rank is a 16-foot rank
*umidi_module
*rank_driver *rank=tibia *bits=1,85
; Now list all the different ways that the tibia pipes could get played
; The lowest pedal on the pedal will play the lowest note in the tibia rank
*relay(pedal_tibia_16) *div=pedal *rank=tibia
; The lowest key on the great division should play the 12th pipe in the tibia rank
*relay(great_tibia_8) *div=great *rank=tibia *offset=12
*relay(swell_tibia_2) *div=swell *rank=tibia *offset=36
; The octave coupler
*relay(swell_tibia_2 & swell_octave) *div=swell *rank=tibia
*offset=48
; The above tab may get coupled to great
*relay(swell_tibia_2 & swell_to_great) *div=great *rank=tibia
*offset=36
; The next rank driver is for the flute, assumed to be 8-foot
*umidi_module
*rank_driver *rank=flute *bits=1,85
*relay(pedal_flute_8) *div=pedal *rank=flute
*relay(great_flute_4) *div=great *rank=flute *offset=12
*relay(swell_flute_1) *div=swell *rank=flute *offset=36

; Now the swell_octave and swell_to_great can be implemented.
*relay (swell_octave & swell_flute_1) *div=swell *rank=flute
*offset=48
*relay(swell_to_great & swell_flute_1) *div=great *rank=flute
*offset=36
```

```
; The 8-foot trumpet
*umidi_module
*rank_driver *rank=trumpet *bits=1,85
*relay(great_trumpet_8) *div=great *rank=trumpet
; The 8-foot oboe
*umidi_module
*rank_driver *rank=oboe *bits=1,85
*relay(great_oboe_8) *div=great *rank=oboe
; The last module, if it is a chest driver, can also perform relay functions
; for external MIDI devices. If this wasn't a chest driver, a separate
; Micro-MIDI module would be needed for relay functions
; Here is a use of the *if statement. When swell_piano is true, the first
; *relay becomes active. The other two are enabled when swell_octave or
; swell_to_great, respectively, are also true.
*if(swell_piano) {
*relay *div=swell *mch=1 *mof=36
*relay(swell_octave) *div=swell *mch=1 *mof=48
*relay(swell_to_great) *div=great *mch=1 *mof=36
}
; We need to get rid of the other MIDI channels since they might
; otherwise confuse the external MIDI device
*midi_channel_truncate=2,0
*midi_channel_truncate=3,0
*midi_channel_truncate=12,0
*midi_channel_truncate=15,0
*combination_action
*set=setBit
*map=mapBit
*piston=piston_1
; other pistons listed here
*piston=piston_32
*memory_select=memSelect
*dual_mag_tab=pedal_tibia_16 *on=mag1:1 *off=mag1:2
*dual_mag_tab=pedal_flute_8 *on=mag1:3 *off=mag1:4
*dual_mag_tab=great_tibia_8 *on=mag1:5 *off=mag1:6
*dual_mag_tab=great_flute_4 *on=mag1:7 *off=mag1:8
*dual_mag_tab=great_trumpet_8 *on=mag1:9 *off=mag1:10
*dual_mag_tab=great_oboe_8 *on=mag1:11 *off=mag1:12
*dual_mag_tab=great_piano *on=mag2:1 *off=mag2:2
*dual_mag_tab=swell_to_great *on=mag2:3 *off=mag2:4
*dual_mag_tab=swell_tibia_2 *on=mag2:5 *off=mag2:6
*dual_mag_tab=swell_flute_1 *on=mag2:7 *off=mag2:8
*dual_mag_tab=swell_piano *on=mag2:9 *off=mag2:10
*dual_mag_tab=swell_octave *on=mag2:11 *off=mag2:12
; Other tab magnets may go here
; That's all, folks
*end
```

MIDI Implementation

This section describes some details of the MIDI implementation in the Micro-MIDI system.
Unless disabled by an explicit or implicit `*midi_channel_truncate`, all MIDI messages arriving at a Micro-MIDI module MIDI input are immediately re-transmitted to the MIDI output. The input is not retransmitted until an entire message has been received, and the module makes use of running status to perhaps reduce the number of bytes which need to be transmitted. This occurs whether or not the input uses running status.
If desired, running status can be disabled on a per-module basis. This is done by adding the keyword `*no_running_status` under the `*umidi_module` which is not to use running status for its MIDI sending.
Note information which is retransmitted is not modified (except perhaps by adding/deleting status words by the use of running status). All note information generated by the Micro-MIDI system uses a velocity of 127 for note on messages, and a note on message with a velocity of 0 for note off. This is true of notes transmitted by a `*relay` definition: All transmitted notes have a velocity of 0 or 127. Note that sostenuto does not preserve the velocity of incoming notes.
Except where configured by the user, all MIDI messages not listed below are ignored (other than being retransmitted).
If a system reset command is received, the Micro-MIDI module will retransmit the system reset command, and will then perform a power-on reset.
For configuration purposes, the Micro-MIDI module responds to system exclusive messages with a manufacturer ID of 47h. Since it is possible that other MIDI devices may respond to this manufacturer ID, the Micro-MIDI configuration program should never be run when the MIDI cables are connected to any other device.
Polyphony, the short answer: In most cases, the Micro-MIDI system does not limit the available polyphony. If you can hold down all the keys on the keyboard, the Micro-MIDI system will do its part to pass on the information properly. However, in certain cases, the Micro-MIDI system has a maximum polyphony of 64. The cases where this might be of concern are rare.
The long answer: When performing unification and rank processing using the *relay function for MIDI outputs, the Micro-MIDI board must remember any note-on messages which it transmits. This information is used to prevent multiple note-on messages for a given note from being transmitted. The use of couplers or unification may result in a given note being played several times simultaneously. In order to function as expected, note on messages must not be generated for the extra note instances, and the note off message must not be transmitted until all reasons to

play that given note have terminated. The size of the table which remembers these on notes is 64. If an attempt is made to play more than 64 notes at once, the 65th and subsequent notes will be ignored and note on messages will not be transmitted.
As keys are released, and notes turn off, these notes will be removed from the note on table. This will make room for more notes, and the previously ignored notes may begin to sound at this time (if they are still being played).
Each note on each channel counts. Thus, if a Micro-MIDI module is performing the *relay function and generating notes on all 16 output channels simultaneously, only four notes per channel could be sounding at any one time.
Note that no applicable MIDI synthesizers which we are aware of have a polyphony capability of greater than 64 notes, so this is never a limiting factor. Some customers are splitting the MIDI stream and using *relay statements in multiple Micro-MIDI modules to drive multiple synthesizers. The 64-note limit applies separately to each Micro-MIDI board implementing MIDI *relay functions.

Note that this applies only to MIDI output. Rank driver boards driving individual pipes have no polyphony limit.

Keywords

This is an alphabetical list of all keywords, with a brief explanation of their purpose.

*all_notes_off
Used in a `*midi_command` to specify an ALL NOTES OFF MIDI message
Example:
```
*midi_command(blue_button) *all_notes_off 3
```
*alt_action_input
Used to specify an alternate action input.
Example:
```
*alt_action_input=sfz *mch=12 *mno=7 *bit=15
```
*always
In an LCD display program, negates the effect of a `*once` command for subsequent commands.
*backlight
Used to control the intensity of an LCD backlight.
Example:
```
*backlight (25)
```
*bar_graph
Used to display a bar graph on an LCD display.
Example:
```
*bar_graph 3 1 20 (c1c7)
```
*bass

Used to specify a relay function in which only the bottom note is active.
Used in `*umidi_module`
*binary
Used to define a controller input from the binary state of up to 8 inputs.
Used in `*control`
Example:
```
*control=mem *binary=23,7 *mch=1 *mco=20
```
In a `*serial_lcd` section, `*binary` specifies that subsequent expressions should be output to the display as a single binary byte.
*bit
Used to specify which bit of a multi-bit device to use.
Used in `*control`, `*input_bit`
Example:
```
*input_bit=flute_8 *bit=4 *mch=12 *mno=4
```
*bits
Used to specify a range of bits on a multi-bit device
Used in `*division`
Example:
```
*div=pedal *bits=1,61 *mch=3 *mno=36
```
The first number specifies the first bit, and the second number specifies the number of consecutive bits starting at that bit.

*blind_combination_action
Used to introduce a blind combination action definition
*cancel
Used to specify the cancel control in a blind combination action, or the cancel input for a one-of-n output.
*combination_action
Used to introduce a combination action definition
*contrast
Used to control the contrast of an LCD display.
Example:
```
*contrast (65)
```
*control
Used to introduce a controller definition
Used in `*umidi_module` and `*hv64`
*count
Used to specify the number of lighted stop boards in one horizontal row
Used in `*lighted_stop`

*decimal
Used to specify how many character positions a number should occupy in an LCD module.
*delay
Used to specify the number of milliseconds that a pizzicato output should be active. When two numbers are specified, the first corresponds to the lowest note of the division, and the second to the highest note of the division.
Examples:
```
*delay=50,100
*delay=75
```
*division
Used to introduce a division (keyboard) declaration
Used at the beginning of the definition file to define an external MIDI keyboard
Used in `*input_64` and `*hv64`
Used in `*relay` to specify a previously defined division
Example:
```
*division or
*division=great
```
*div
An abbreviation for `*division`
*dmt
Abbreviation for `*dual_mag_tab`
*dual_mag_tab
Used to introduce a dual magnet tab definition
Used in `*combination_action`
Example:
```
*dual_mag_tab=pedal_flute_8 *on=mag1:1 *off=mag1:2
```
*dual_mag_time
Used to specify the number of milliseconds to energize relay drivers for dual magnet tabs
Default: 100 mSec
Used in `*combination_action`

Example:
*dual_mag_time=100
*else
Used in *if statements
*end
The last line of a definition file. Any text after *end may cause a syntax error.
*hd32

Used to introduce the definition of a hd-32 driver board
Used in *umidi_module
Example:
*hd32 or *hd32=magnets1
*hv64
Used to introduce the definition of a hv-64 input board
Used in *umidi_module
*ibit
Abbreviation for *input_bit
*if
Used to define a group of *relay statements which are active based upon the value of an expression
The general format is *if (expression) { <statements active if expression is true> }
or *if (expression) { <statements if true> } *else { <statements if false> }
*if may also be used to control commands being sent to an LCD display.
*Increment_decrement
Increment/Decrement is used when it is desired to utilize two buttons to step up or down a controller value.
Example:
*increment_decrement=memory *bits=15,16 *range=0,11 c3c56
This must be part of a *hv64
This creates a controller named memory. It starts with a value of zero
(but won't send that value right away). Input 15 going true will cause
the controller to increment (to 1) and that value to be sent on channel
3 controller 56. It will increment as high as 11. Further inputs on 15 will
cause the controller to be retransmitted, but its value will not change.
Input 16 will cause the controller to be decremented as low as zero.
We have another feature:
*inc_dec=transpose *bits=5,6 *range=1,13 *init=7 c4c17
*inc_dec is an abbreviation for *increment_decrement
The controller will have an initial value of 7, but the consumer won't know
that and will assume it's 0 (uMIDI) or -1 (sound engine).
It will increment up to 13, or decrement down to 1. The transpose statement
now looks something like
*transpose (transpose * (transpose - 6)) when used in (uMIDI) or
*transpose ((transpose != -1) * (transpose - 6)) when used in the sound engine.
*input_64
Used to specify a 64-bit serial input board. This is an obsolete board, and no longer sold by Artisan Instruments.
*input_bit
Used to define a single MIDI note used for some control purpose
Used at the beginning of the definition file to define externally generated notes

Used in *lighted_stop, *hv64 and *input_64
Example:
*input_bit or *input_bit=pedal_bourdon_16
*invert
Used to invert the on/off sense of bits on hv-64 input boards.
With *invert, voltages above the threshold (nominally 2.5V) are considered off, and voltages below the threshold are considered on.
*invert can also be used to define an input bit which sends a program change message when the input turns off.
*lighted_stop
Used to introduce the definition of a horizontal row of lighted stop modules
*line
Used to place the cursor on an LCD module
Example:
*line 2
*local_control_off
Used in a *midi_command command to specify a LOCAL CONTROL OFF MIDI message
Example:
*midi_command(button_3) *local_control_off 4
*local_control_on
Used in a *midi_command command to specify a LOCAL CONTROL ON MIDI message
Example:
*midi_command(button_4) *local_control_on 5
*map
Used to specify which bit should be used to map the operating range of pistons. This keyword is optional, and if not used, all pistons will always affect all stops.
Used in *combination_action
Example:
*map=map_bit
*mch
Abbreviation for *midi_channel
*mco
Abbreviation for *midi_controller
*melody
Used to specify a relay function in which only the top note is active.
Used in *umidi_module
*memory_select

Used to specify a controller which selects a memory bank to use for combination action.
Used in *combination_action
Example:
*memory_select=myBank
*midi_channel
Used to specify which MIDI channel will be used for messages.
Used in *division, *input_bit, *control, *lighted_stop, *relay
Example:
*midi_channel=1

*midi_channel_truncate
Used to specify that a Micro-MIDI module should not pass certain MIDI messages from its input to its output.
Used in *umidi_module
The first number following *midi_channel_truncate is the MIDI channel which should be affected
The second number following *midi_channel_truncate is a bit map representing the messages which should be transferred. Specifying zero causes no messages to be passed from the input to the output.
In order to transfer some messages from input to output, specify the second number as the sum of one or more of the following:
128 Transfer note off messages
64 Transfer note on messages
32 Transfer polyphonic key pressure messages
16 Transfer control change messages
8 Transfer program change messages
4 Transfer channel pressure messages
2 Transfer pitch wheel change messages
Example:
```
*midi_channel_truncate=2,0
```
Note: An appropriate *midi_channel_truncate is automatically supplied by the system to truncate all note on and note off messages for all output channels used in relay functions. Supplying a *midi_channel_truncate for such a channel overrides this automatic setting.
*midi_command
Used in an output *umidi_module
Introduces a MIDI message which is to be sent when an expression becomes true.
Example:
```
*midi_command (panic_button) *all_notes_off 1
```
*midi_controller

Used to specify a controller number
Used in *control
Example:
```
*control=loud *mch=2 *midi_controller=7
```
*midi_note
Used to specify which MIDI note should be used for a message, or the bottom note for a range of messages (such as a division).
Used in *input_bit, *division, *lighted_stop
Example:
```
*midi_note=24
```
*midi_notes
Used to specify which MIDI notes are used by an external division.
Used in *division at the top of the definition file.
Example:
```
*division=great *mch=1 *midi_notes=36,61
```
*midi_offset
Used to specify a pitch modification in a relay function

Used in *relay
Example:
*relay *div=great *tab=flute_4 *midi_offset=12
*mno
Abbreviation for *midi_note
*mof
Abbreviation for *midi_offset
*mono_mode_on
Used in a *midi_command command to specify an MONO MODE ON MIDI message
Example:
*midi_command(button_two) *mono_mode_on 8,4
*name
Used to attach a name to some resource.
Used in *division, *input_bit, *hd32, *rank_driver, *offset_driver, *dual_mag_tab
Names can be supplied with the section name, so *name is never necessary. For example,
*division=swell
Implies
*name=swell.
*negative_voltage

Used on hv-64 boards to notify the microprocessor on the board that the board is configured to accept negative voltages. Only necessary if the jumper has been moved from its default location.
*no_invert
Used on hv-64 boards to cause voltages higher than the threshold voltage (nominally 2.5V) to be considered as on, and voltages lower than the threshold voltage to be considered as off.
*no_running_status
Used to specify that a Micro-MIDI module should not utilize running status when transmitting MIDI messages. This will cause more bytes to be used to transfer data, and should not normally be needed.
Used in *umidi_module
*off
Used to specify which output bit should be used to turn off a dual magnet tab
Used in *dual_mag_tab
Example:
*off=magnets1:4
The name after the = specifies the appropriate output driver card, and the number after the colon (:) specifies the bit on that driver card.
*offset
Used in a relay rank driver definition to create a transpose up with a positive number, or down with a negative number.
Example:
*offset=12
*offset_driver
Used to introduce the definition of a 16-bit offset driver board
*omni_mode_off
Used in a *midi_command command to specify an OMNI MODE OFF MIDI message

Example:
```
*midi_command(purple_button) *omni_mode_off 7
```
*omni_mode_on
Used in a `*midi_command` command to specify an OMNI MODE ON MIDI message
Example:
```
*midi_command(orange_button) *omni_mode_on 7
```
*on
Used to specify which output bit should be used to turn on a dual magnet tab
Used in `*dual_mag_tab`
Example:
```
*on=magnets1:5
```

The name after the = specifies the appropriate output driver card, and the number after the colon specifies the bit on that driver card.
*once
Used in an LCD program to specify that LCD commands are to be sent only once at powerup time, instead of continuously.
*one_of_n
Used to define a controller input from n individual inputs.
Used in `*control`
Example: `*control=bank *one_of_n=20,32 *mch=12 *mco=23`
*output_bit
Used to control an individual output bit
Example:
```
*output_bit(drum) *bit=5
```
*piston
Used to specify that an input bit should be treated as a piston actuation.
Used in `*combination_action`
Example: `*piston=general_5`
*pizz
Used to specify a pizzicato function.
Example:
```
*pizz=swell_pizz (pizz_enable) *div=swell *mch=3
*delay=50,100
```
*poly_mode_on
Used in a `*midi_command` command to specify a POLY MODE ON MIDI message
Example:
```
*midi_command(b8) *poly_mode_on 9
```
*positive_voltage
Used on hv-64 boards to notify the microprocessor on the board that the board is configured to accept positive voltages. Only necessary if the jumper has been moved from its default location.
*program_change
Used to specify the program number that should be used to represent an input bit.
Example: `*program_change=15`
*range

Used to specify the desired range of a controller

Used in *control
Example:
*range=10,40
The number after the = specifies the value the controller should have when the input voltage is ground, and the second number specifies the value the controller should have when the voltage is equal to its reference value.

*rank
Used to define a set of outputs as a rank, or to reference that rank in a relay statement to drive that rank.
Example:
*rank=flute *bits=1,85

*rank_driver
Used to introduce the definition of a rank driver board
Used in *umidi_module
Example:
*rank_driver=magnets2

*rate
Used to specify the repeat rate in beats per second for a reiterated division. If two numbers are given, the first is for the lowest note on the division, and the second is for the highest note on the division.
Examples:
*rate=10
*rate=10,20

*reit
Used to specify a reiteration function.
Example:
*reit=solo_reit (reit_enable) *div=solo *mch=4 *rate=10,20

*relay
Used to introduce the definition of a relay function
Used in *umidi_module

*reversible
Used in a *combination_action section to create a reversible.

*serial_lcd
Specifies that a module is to be used to drive an LCD module with a serial interface
Used in *umidi_module

*set
Used to specify which input bit should be used to set pistons.
Used in *combination_action
Example:
*set=set_bit

*signed
When displaying numbers on an LCD module, specifies that a character position should be reserved for a minus sign, and the number should be treated as a signed number between -128 and 127.

*single_row
Used to specify that a row of lighted stop modules has only a single row of switches (four per module). This prevents the system from reserving resources for the non-existent switches.

Used in *lighted_stop

*sostenuto

Used to introduce a sostenuto function.

Example:

```
*sostenuto (sos_switch) *division=accomp
```

*system_reset

Used in a *midi_command command to specify a system reset message

Example:

```
*midi_command(red_button) *system_reset
```

*tab

The *tab keyword has been obsoleted by a syntax enhancement, and should not be used in new designs. It may disappear in future versions.

Used in a relay definition to specify an input bit which needs to be on (or off) for the relay function to be active.

Used in *relay

Example:

```
*tab=pedal_flute_8 or,
*tab=!unison_off
```

The extra "!" in the second example specifies that the tab "unison_off" must be off for this relay function to be active.

*threshold

Used on hv-64 input boards to modify the voltage at which inputs switch between off and on. For positive voltage boards, the default is 16, and for negative voltage boards, the default is 240. In both cases, the threshold is 2.5V.

Example:

```
*threshold=32
```

*trap

Used to specify a relay function in which only one output note (or trap) is active whenever one or more notes are played on the input division.

Used in *umidi_module

*umidi_module

Introduces the definition of one Micro-MIDI module. Each *umidi_module in the definition file corresponds with one Micro-MIDI module in the system.

*unsigned

When displaying numbers on an LCD module, specifies that a character position should not be reserved for a minus sign, and the number should be treated as an unsigned number between 0 and 255.