ultimate
EXPANSION SYSTEM

MIDI and HQ Sound Sampler User Guide

MIDI and HQ Sound Sampler User Guide

Written by

Christopher Maughan

THIS MANUAL WAS PRODUCED ON AN ARCHIMEDES A4000 FITTED WITH 80MB IDE HARD DRIVE, MIDI + HQ SOUND SAMPLER, HIVISION DIGITISER AND SCSI MICROPODULES.

Contents

The HQ Sound Sampler

The HQ Sound Sampler SoftWare Interrupts

Introducing MIDI

What is MIDI?

MIDI is the acronym for Musical Instrument Digital Interface. It is a means of connecting electronic instruments, such as synthesisers and drum machines, to each other and to sequencers and computers, so that they can interact. Thus music can be stored and edited by a computer, to be played on a synthesiser or rhythm machine. This is possible because MIDI provides a standard form of communication.

Each MIDI instrument will have a transmitter and a receiver, or occasionally just one of these, using a standard code to convey information about keys pressed, note lengths and a variety of other possible messages, such as which of the different voices of a synthesiser is to play the note.

These voices are the different programmed sounds which the instrument can make, such as piano or bongo.

What the MIDI interface can do

By installing the MIDI interface in your computer and connecting a music keyboard, you can use the computer as a musical instrument. The sound can be played through the speakers on the computer, through headphones or your Hi-Fi system.

The computer can also be used to extend the capabilities of the instruments you have, provided they have MIDI installed. By writing or buying application programs much more becomes possible. You can write such programs yourself using the high speed BASIC V on the computer.

Connecting up

Warning : MIDI sockets carry digital information; audio connections (such as those from your Hi-Fi system) must NEVER be made to these sockets as serious damage could result.

MIDI instruments must be connected to your computer via MIDI leads. These may look the same as your ordinary Hi-Fi leads, but the cables are different. The MIDI leads will have a five pin DIN plug on each end, connected by a shielded, twisted-pair cable, and are available from your HCCS dealer.

Your Music keyboard or synthesiser must have MIDI installed. This is sometimes an optional extra, so check before you buy.

The HCCS micropodule has two MIDI ports, they are labelled:

PORTI	PORT2
MIDI (1) OUT	MIDI (2) OUT
MIDI (1) IN	MIDI (2) IN

These connections can be found on the connector provided with your micropodule. This consists of 5 leads protruding from a shrouded D-Type plug. One of these leads is for the High Quality Sound Sampler, if fitted. The four leads for use with the MIDI should be clearly labelled and easily identified, they all terminate in a 5-pin DIN female socket of standard MIDI type; it is into these that your MIDI leads should be plugged.

Choose which of the ports you wish to use, and connect the MIDI IN socket on the instrument to the relevant MIDI OUT socket on the MIDI expansion card, and the MIDI OUT socket on the instrument to the relevant MIDI IN socket on the expansion card, see figure 1.

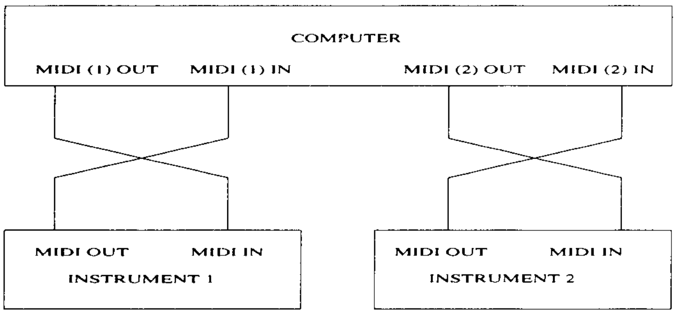


Figure 1: Connecting the MIDI ports.

To connect further instruments, use the MIDI THRU connectors provided on the instruments. Refer to the instrument user guide for details.

Testing

To check that you have connected your interface satisfactorily, type:

```
*modules
```

The system will now list the modules you have available, which should include:

MIDI

If not, your MIDI interface has not been installed correctly. Refer back to your supplier or check your installation instructions, if you have made the installation yourself.

Assuming that MIDI is on the list, type:

```
*midisound in
```

Now when you play the music keyboard it will play the computer sound system. You can either type

```
*speaker on
```

to hear it from the built in speaker or, for higher quality stereo sound, plug a pair of headphones or your Hi-Fi system into the Headphones socket at the back of the computer.

If, when you type *voices, a number of voices are listed on your screen, then you can switch your system between these voices using the program change buttons on your intrument.

MIDI Problems

Setting up MIDI equipment is usually quite an easy task, although getting software, hardware and instrument to work together can sometimes be not as simple a process as it first seems. If, while setting up your system, you encounter problems, you may wish to confirm the functionality of the HCCS MIDI MicroPodule. To this end HCCS provides, with your MicroPodule, a piece of test software to ensure that you are succesfully sending and receiving MIDI information via your MicroPodule.

As you have seen, the MIDI MicroPodule has two MIDI Ports (see the previous page). Using some MIDI leads connect Port (1) OUT to Port (1) IN, and Port (2) OUT to Port (2) IN. This creates a loop for the test program to check the ports with. Now switch on and double click on the supplied application - !MSTest. The program will ask how many ports you have connected (the default is 2), and the screen will then split into numbered sections, depending on the specified number of Ports . Number 1 corresponds to Port 1, Number 2 to Port 2. Each number will have a background colour which will fill a large portion of the screen. If all of the colours are green then everything is OK. If one of the areas is red then you may have a faulty MIDI lead in which case try swapping them. If both areas are red your MIDI interface may have developed a fault in which case please contact your dealer or HCCS.

Programmer's guide to MIDI

Writing programs

The very fast BASIC V enables you to write application programs in BASIC which use your MIDI interface. To do this, use the software interrupts listed in the next chapter.

Application software

The full potential of the MIDI system can be realised using application software. Only packages specifically written for the RISC OS operating system are suitable. This chapter should be read in conjunction with the *Programmer's Reference Manual,* available from Acorn.

Software structure

The MIDI micropodule interface consists of hardware and software to give an interface for both input and output of data to MIDI specification 1.0.

The software is in six main parts:

- A low-level interrupt routine which buffers data to and from the DUART (Dual Universal Asynchronous Receiver Transmitter), which transmits and receives the raw MIDI data. This routine also deals invisibly with transmission and reception of the System Real Time messages used to synchronise the system.

- A combined MIDI interpreter and sound system driver which can be driven from the incoming or outgoing MIDI data.

- A millisecond routine which time-stamps the incoming MIDI data, and which can time scheduled outgoing data.

- A sound interrupt (SIRQ) synchronous routine for background operation, and which can time scheduled outgoing data.

- A set of SWI (SoftWare Interrupt) calls (listed in the next chapter) to provide an interface between the application program and the MIDI interface.

- A set of operating system *commands.

The MIDI software is controlled and communicated with using SWI calls and can also be controlled, to a limited extent, with *commands and via the MIDI interface. The software provides low-level support for receiving and transmitting data bytes allowing great flexibility for an application program.

Data format

MIDI messages have one status byte followed by one or two data bytes, except System Real Time and System Exclusive messages - see below.

- Channel Voice messages contain a four-bit number in the status byte which addresses the message to one of 16 channels. An instrument is assigned to a basic channel over which mode messages are sent. If the instrument has several voices these can be assigned extra channels to control the voices separately.

- System messages are not given channel numbers; there are three types, as follows:

  - Common messages are for all the units in a system.

  - Real Time messages are for all the units in a system and contain only status bytes. They can be sent at any time, even between bytes of another message.

  - Exclusive messages include a manufacturer's identification code, and any number of data bytes, terminated by an 'end of exclusive' or another status byte. Only equipment which recognises the identification will accept the data, the format of which is specified by the manufacturer.

Data types

Status bytes

Status bytes are eight bits with the most significant bit (MSB) set (1). These identify the message type. Status bytes command the receiver to adopt their status, except for Real Time messages. Unimplemented or undefined status bytes are ignored.

Running status

For voice and mode messages, the receiver will remain in the status of the last status byte until a different one is received. Therefore when many messages with the same status are being sent, the status byte may be omitted. This is particularly useful for long strings of Note On/Off messages. Real Time messages will only change the running status temporarily.

Data bytes

One or two data bytes follow the status bytes, except in Real Time messages. Data bytes are eight bits, with the most significant bit set to 0. Each status byte must be followed by the correct number of data bytes, and action will wait until they are all received.

Channel modes

Mode messages control the way in which Note On/Off information is routed to the instruments and their voices via the 16 MIDI channels. The mode messages available for this purpose are defined in the table below. A MIDI receiver or transmitter can only operate in one mode at a time.

Each receiver is assigned to a basic channel N, and may have a number of voices M assigned to adjacent channels.

Mode	Omni		
1	On	Poly	Voice messages are received from all channels and assigned to voices polyphonically by the receiver.
2	On	Mono	Voice messages are received from all voice channels and control only one voice, set by the receiver, monophonically.
3	Off	Poly	Voice messages are received in voice channel N only, and are assigned to voices polyphonically by the receiver.
4	Off	Mono	Voice messages are received in voice channels N to N+M-1 and assigned monophonically to voices 1 to M respectively. The number of voices M is specified by the Mono Mode message, one voice per channel. This means that each voice in the receiver can be controlled separately by the transmitter.

Each transmitter is also assigned to a basic channel N. Those without the capacity to select channels will normally use channel one.

MIDI interpreter and sound system driver

This background process allows MIDI data from the MIDI IN or MIDI OUT sockets to drive the sound system directly, without user intervention, and with the computer still able to perform other tasks normally. This facility is disabled by default, and is enabled from the command line using *midi sound. The state of this driver does not effect the data received by MIDI_RxCommand and MIDI_RxByte.

The MIDI interpreter does not respond to notes of very short duration (less than about 20ms). This means that certain drum machines cannot be used to trigger notes on the computer sound voices.

Program Change messages are interpreted as Sound Voice requests (like *Channel Voice). If a program change message with a data value greater than the number of installed sound voices is interpreted; a non-existent voice will be selected and the sound will apparently stop working. To restart the sound a lower Program Change data value should be used. Which Voice Channels are affected depends upon the MIDI Mode and MIDI Basic Channel in the normal way.

MIDI implementation

Some MIDI commands are ignored by the MIDI interpreter. These are listed below:

System messages:

- Song Position Pointer
- Song Select
- System Exclusive

Real time messages:

- Timing Clock
- Start
- Continue
- Stop
- System reset

Channel voice messages:

- Control Change 0-121
- Polyphonic Key Pressure (After touch)
- Channel Pressure (After touch)

and all bytes undefined in MIDI specification 1.0.

These commands are, however, available for use by application software.

Pitch Bend

The pitch bend algorithm used by the sound system driver has the shortcoming that if notes are newly triggered while pitch bend is in operation, they will start with the normal pitch, and will only be pitch changed when the next pitch bend message is received.

Midi ports

With one MIDI micropodule installed, you will have two MIDI ports, numbered 0 and 1. Up to four MIDI ports are supported by the MIDI module, which can be obtained by installing another micropodule. Application software supporting multiple ports should use the SWI MIDI Init to find the number of recognised ports currently installed.

MIDI *commands

MIDI *commands can be divided into two groups:

• Interpreter control

• Timing clock generator control.

These are as follows:

Interpreter control

*MidiSound	Parameter 1 'in','out' or 'off' to specify incoming data, outgoing data or disable port.
*MidiTouch	Parameter 2 (optional) <1-4> to specify MIDI port number. This enables or disables the MIDI interpreter and specifies the source of MIDI data.
*MidiChannel	Used to set the Basic Channel of the MIDI interpreter. Parameter is 1-16.
*MidiMode	Used to set the MIDI mode of the MIDI interpreter. Parameter is 1-4

Timing clock generator control

*MidiStart <time>	Used to transmit a Start message and start automatic transmission of Timing Clock messages every <time> milliseconds. Parameter, if specified, is integer time in milliseconds 1-&FFFF, or 0 to leave unchanged.
*MidiStop	Used to transmit a Stop message and stop automatic transmission of MIDI Timing Clock messages.
*MidiContinue	Used to transmit a Continue message and to restart automatic transmission of Timing Clock messages.

Software Interrupts (SWIs)

General

Software interrupts provide an interface between an application program and the MIDI interface itself.

Values given for RO and R I are the contents of those CPU registers; where no values are given the command does not use the registers.

SWI calls used for setting up and controlling the MIDI are detailed on the following pages. They have been divided into the following groups:

- General interface commands
- Data reception commands
- Data transmission commands
- Service calls
- Events.

MIDI_Sound_Enable (&404C0)

Enable the MIDI interpreter/sound system driver, so that MIDI data will be interpreted, and control the sound system. The mode will default to I (Omni On:Poly), but may be changed under MIDI or SWI control.

On entry	R0	= 0	to disable Sound Interpreter. (It is disabled on initialisation).
		= 1	for connection to Port 0, input buffer.
		= 2	for connection to Port 0, output buffer.
		= 3	for connection to Port 1, input buffer.
		= 4	for connection to Port 1, output buffer.
	R1	= 1	to enable touch sensitivity of interpreter.
		= 2	to disable touch sensitivity.
		Any other values of R1 ignored at present.	
On exit	R0	undefined	

Example In BASIC:

```
SYS "MIDI_SoundEnable",1
```

does the same as:

```
*MidiSound in
```

MIDI_SetMode (&404C1)

Set the MIDI channel mode of the internal sound system controller. Use 0 to read current values.

On Entry	R0	=	mode number, 1-4, if 0 then unchanged.
	R1 byte 0	=	basic channel number N, 1-16, if 0 then unchanged.
	byte 1	=	Number of channels in mode 4, 1-8 (M), if 0 then unchanged.
On Exit	R0	=	new (or current) mode number (1-4).
	R1	=	new (or last) settings of N (1-16) and M (1-8).

The possible modes are defined under Channel Modes.

MIDI_SetTxChannel (&404C2)

Set the MIDI channel and port number for subsequent transmitted commands to be sent on. This applies to all SWI commands prefixed by MIDI_Tx except MIDI_TxByte and MIDI_TxCommand.

On entry	R0	=	new channel number (1-64), if 0 then unchanged.
		1-16	MIDI channels 1-16 of receivers connected to MIDI port number 0
		17-32	channels 1-16 connected to port 1.
		33-48	channels 1-16 connected to port 2.
		49-64	channels 1-16 connected to port 3.
On exit	R0	=	new (or current) channel number.

If selected MIDI port is not installed, then it is undefined which port this and other SWIs will use instead. Use MIDI_Init to find the maximum port number installed, and never exceed it.

MIDI_SetTxActiveSensing (&404C3)

This puts the transmitter into Active Sensing mode, which causes dummy bytes to be transmitted in the absence of any other MIDI activity for longer than 280ms. The receiver should automatically switch to Active Sensing mode and expect this activity, or switch off all voices if it stops. This prevents voices becoming 'stuck on' if the MIDI cable becomes disconnected. Some MIDI receivers do not support this.

			Active Sensing Messages in requested Port.
		= 1	to start automatic regular transmission of Active Sensing Messages in requested Port.
	`R0` bits 1-2	=	Midi Port Number.
On exit	`R0` bits 0-3	=	corresponding to Midi Ports 0-3; bits set if Transmit Active Sensing is enabled for this Port.
	`R0` bits 4-7	=	corresponding to Midi Ports 0-3; bits set if this Port is receiving Active Sensing.

MIDI_InqSongPositionPointer (&404C4)

Return the value of the internal Song Position Pointer, which is the value of the MIDI beat counter divided by six. See the chapter entitled Timing.

On entry Unimportant.

On exit	R0	=	Song Position Pointer.
	R1	=	bits set according to current state of MIDI:
	bit 0		set if in External Timing Mode (Start message has been received).
	bit 1		set if in Internal Timing Mode (Start message has been transmitted. Timing Clock transmission is automatic).
	bit 2		set if in Fast Clock Mode.
	bit 3		set if new (version 3) facilities enabled (`MIDI_FastClock` has been called). This flag is only reset on MIDI_Init with R0=0, `*RMReInit` midi or Ctrl-Break.
	bit 4		set if in special mode to store System Real Time Messages in receive buffer.
	bit 5		set if in special mode to cause System Real Time Messages not to be treated in a special way.

Bits 0 and I are determined by the current timing mode. See the chapter Timing.
Bits 2 and 3 are set by calling MIDI_FastClock with relevant parameters.
Bits 4 and 5 are set by calling MIDI_Init with bits 30 and 31 of R0 set.

MIDI_InqBufferSize (&404C5)

Return the number of empty bytes in the transmit or receive buffer. These buffers can fill (rx buffer) or empty (tx buffer) at a maximum rate of 320 microseconds per byte. Default buffer sizes are:

- Transmit buffer size - 512 bytes
- Receive buffer size - 1024 bytes
  (Programmable with MIDI_SetBufferSize).

On Entry	R0 bit 0	= 0	to read rx buffer size.
		= 1	to read tx buffer size.
	R0 bits 1-2	=	MIDI port number 0-3.
On exit	R0	=	number of bytes free in selected buffer.

MIDI_InqError (&404C6)

Return the value of the MIDI error bytes.

On entry	Unimportant.		
On exit	R0	=	Up to four error bytes, corresponding to one byte per installed MIDI port.

Possible values of the error byte (shown as the ASCII character with the decimal value following in brackets) are:

'A' (65)	Active Sensing failure error (MIDI connection removed, or Active Sensing transmission was stopped by the transmitter).
'B' (66)	Receive data FIFO buffer was (and still may be) full and data has been lost. The application program should take the data more quickly.
'O' (79)	DUART overrun error. This means that the received data arrived, but was not read from the DUART receive register before it was overwritten by the next data byte. This might occasionally happen when there is a lot of other processing occurring with the processor interrupt flag clear (high-numbered screen modes with simultaneous sound and intensive processor activity might also occasionally cause this error).
'F' (70)	Framing error. This flag is generated by the DUART when serial data arrives which does not fit the expected protocol, i.e. not sensible MIDI data.
'V' (86)	Received MIDI data has caused the interpreter to attempt to use more than eight voices of the internal sound system, which are allocated on a first-come first-served basis.
'T' (84)	Transmit data FIFO buffer has overflowed, and data to be transmitted has been lost. The application program should transmit data more slowly, or use MIDI_InqTxBufferSize.

'L' (76)	Note too low (or too high) for the internal sound system, received by the interpreter and ignored. The lowest note that the sound system can make is the C four octaves below middle C, or MIDI value 12. The highest note is MIDI value 96 or three octaves above middle C.
0 (zero)	No error.

NOTES:

- Only the latest error is shown. Previous errors are overwritten.
- Overrun and framing errors are also returned as standard SWI errors by MIDI_RxCommand and MIDI_RxByte at the time that they read the corrupted byte.
- The error is cleared when read.

MIDI_IgnoreTiming (&404DF)

This operates as a switch, instructing the system either to ignore any further received Timing messages: Start, Continue, Stop and Timing Clock, or to revert to normal reception of them.

On entry	R0	= 0	receive messages normally (default).
		= 1	ignore received timing messages.
On exit	No change.		

NOTE: There is a subtle distinction between the mode set by this SWI, where received Timing Messages are completely ignored, and the mode set by calling MIDI_Init with bit 31 set, which only disables special actions on Timing Message reception. The messages may still be stored in the receive buffer in the later case, if MIDI_Init is called with bit 30 set, but not if Ignore Timing mode is set.

MIDI_SynchSoundScheduler (&404E0)

On entry	R0	= 0	set normal mode where sound scheduler is synchronised to the Sound Interrupt (SIRQ).
		= 1	set special mode where the sound scheduler is synchronised to incoming MIDI Timing Clock Messages (prefixed by Start, ended by Stop). Scheduler time is incremented by one tick for each Timing Clock message received.
On exit	R0	=	previous value of SynchSoundScheduler flag.

MIDI_FastClock (&404E1)

On entry	R0	= <0	read current value of fast clock.
		= 0	stop fast clock; revert to normal timing.
		= >0	= t = Timing Clock Transmission rate. Reset and start fast clock. Incoming data will be time stamped with the time in milliseconds shown on this clock. When started with MIDI_TxStart or *midistart, Timing Clock messages will be automatically transmitted at a rate of one every t milliseconds. The transmission will be stopped with SWI MIDI_TxStop or *midistop.
	R1	=	time to reset clock to if R0 >= 0.
On exit	R1	=	previous value of fast clock.

The fast clock increments every millisecond.

This SWI should be called at least once by new applications that want to use the MIDI scheduler.
On calling MIDI_FastClock with R0 > 0, Fast Clock Timing mode is set. In this mode the value in R1 when calling SWI MIDI_TxCommand will determine the schedule time in milliseconds, as registered by the Fast Clock. If the value in R I is zero, then the message will be sent immediately.

User Timer — The fast clock uses Timer 1 (the User Timer). Obviously this cannot be used simultaneously by other software while the Fast Clock is running, or the Fast Clock will not work correctly.

MIDI_Init (&404E2)

On entry	R0	= 0	do an Internal System Reset (reset to power-on state).
	R0	> 0	
		bit 0	set to clear current transmitted running status (ensure status is included with next transmitted message).
		bit I	set to clear receive buffers and reset midi interpreter.
		bit 2	set to clear transmit buffers and reset midi interpreter.
		bit 3	set to clear MIDI scheduler.
		bit 4	set to clear current error.
			(special mode bits, only cleared by a call with R0=0).
		bit 30	set to enable special mode so that received System Real Time messages will be stored in the receive buffer, so that they can be read with SWI MIDI_RxCommand and SWI MIDI_RxByte.
		bit 31	set to prevent special actions on reception of System Real Time messages. Use SWI MIDI_IgnoreTiming in preference to this, to just cause Timing messages to be ignored.
On exit	R0	=	number of recognised Midi Ports installed. Subtract one from this number for the maximum port number, which should not be exceeded.

Certain MIDI recording and replaying applications may need to set bits 30 and 31 so that they can precisely reproduce the recorded data with the Real Time messages.

MIDI_SetBufferSize (&404E3)

Clears the buffer and then claims the requested buffer size from the RMA. Returns No room in RMA error (&102) if unable to claim the new buffers, and leaves the previous buffers intact.

NOTE: This should only be used when the buffers are empty, otherwise data will be lost.

On entry	R0	=	0	for set receive buffer size (nothing else currently supported).
	R1	=		new buffer size in bytes.
		=	0	interrogate current value.
On exit	R0	=		buffer size in bytes.
	R1	=		total space in bytes claimed from RMA for new buffers.
		=		5 x size x number of MIDI Ports installed for receive buffer (for each byte received, four bytes of time stamp is also stored)

Use SWI MIDI Init to just clear the buffers, (R0, bits 1 and 2).

MIDI_Interface (&404E4)

(For advanced use only.) Get addresses for more efficient access to critical SWIs.

On exit	R0	=	workspace pointer, moved to R12 when calling a SWI through this interface.
	R1	=	SWI code pointer.

When calling SWIs using this interface, the CPU should be in supervisor mode.

The MIDI SWIs do not support re-entrancy, so they should not generally be called from an interrupt routine. R11 should contain the SWI offset from chunk base (MIDI_SoundEnable = 0). R12 should contain the workspace pointer. The addresses become invalid if the MIDI module is re-initialised, or finalised, so watch for MIDI service calls to warn of this, and re-call this SWI.

MIDI_RxByte (&404C7)

Return the next received MIDI byte, excluding Real Time messages which are processed internally (see section on special messages). In general, MIDI_RxCommand should be used in preference to this command, for reduced SWI time overhead, although this SWI should be used for reading the 'raw' data.

On entry	R0	=	port number (0-3) to receive message from.
		_ -1	to look at each port in order of increasing port number, until one is found in which new data as been\ received, and return that data.
On exit	R0 byte 0	=	next received MIDI byte.
		= 0	if receive buffer empty, or incomplete message received. If entered with R0 =-l, distinguish between these cases by checking the port number returned in bits 28-31.
	bit 24	= 1	if byte received.
	bits 28-31		MIDI port number where this byte was received.
	R1	=	received time (see section on timing).
		= 0	if receive buffer empty or clock disabled.

In the case of a low-level error registered by the DUART, this SWI returns an error number:

- &20402 if there was a Framing Error when this byte was received
- &20403 if there was an Overrun Error when this byte was received.

The error number &20404 will be returned if the receive buffer of the interrogated port overflowed.
The corrupted byte will be returned on the next SWI MIDI_RxByte or MIDI_RxCommand.

MIDI_RxCommand (&404C8)

Return the next complete MIDI command as a set of bytes (normally excluding System Real Time). A status byte should always be returned, even when the incoming data is on running status, except when the receive buffer is empty.

NOTE: System Exclusive messages will be received as one status byte (&FO and one data byte, and successive calls to MIDI_RxCommand will treat the system exclusive as Running Status, with one data byte, until EOX (end of exclusive: &F7) or any other status byte, except System Real Time, is encountered (&80-&F7).

On entry	R0	=	port number (0-3) message will be received from.
		_ -1	look at each port in order of increasing port number, until one is found in which new data has been received, and return that data, if Midi Message entirely received, or 0 if not or if no port has received any data.
On exit	R0 byte 0	=	status.
	byte 1	=	data byte 1, or 0 if none expected.
	byte 2	=	data byte 2, or 0 if none expected.
	bits 24-25		number of bytes in this message = 0-3.
	bits 28-31		MIDI port number of port that this message was received by.
		= 0	if receive buffer empty, or incomplete message received.
	R1	=	received time of last byte in message.
		= 0	if receive buffer empty or clock disabled.

In the case of a low-level error registered by the DUART, this SW I returns the following error numbers:

- &20402 if there was a Framing Error when a byte in this message was received.
- &20403 if there was an Overrun Error when a byte in this message was received.

The error number &20404 will be returned if the receive buffer of the interrogated port overflowed.
The corrupted byte will be returned on the next MIDI_RxByte or MIDI_RxCommand.

MIDI_TxByte (&404C9)

Transmit a byte from the MIDI OUT. This will be transmitted, regardless of Running Status

On entry	R0 byte 0	Byte to transmit.
	bits 28-31	Port number to transmit from (0-3).
On exit	Unchanged.	

Returns error number &20401 if this fails because the transmit buffer is full.

Data transmission commands with automatic running status optimisation

Automatic Running Status optimisation applies to all the commands in this section; that is, the status byte may not be transmitted if the last command had the same status. However. a limit applies to long chains of commands under Running status, and the status byte is periodically re-transmitted.

MIDI TxCommand (&404CA)

Transmit or schedule on the MIDI schedule queue a complete MIDI command. The status byte may not be transmitted if Running Status applies. The command will be ignored and not transmitted if byte 0 is not a status byte (bit 7 set), or if the data bytes have bit 7 set.

NOTE: The format is the same as MIDI_RxCommand, so that received commands can be simply re-transmitted without decoding or encoding. It can be used to transmit a MIDI command immediately, or to schedule one to be transmitted at some future time (if in Fast Clock mode, and RI>0).

On entry	R0	byte 0	=	status.
		byte 1	=	data byte I, if required by specified status.
		byte 2	=	data byte 2, if required by specified status.
		bits 24-25		(optional) number of bytes in command; only needed for status undefined in MIDI Specification 1.0.
		bits 28-31		port number to transmit from (0-3.)
	R1		=	schedule time (0 for immediate) in: Fast clock time if in Unset or Internal timing modes. Timing Clock received count time if in External timing mode.

On exit	If R I was non-zero on entry:		
	R0	=	number of scheduler slots free in queue.
		= -1	if it failed because the scheduler was full.
		= -2	if it failed because the schedule time requested was earlier than the time of the previous event in the scheduler queue.

Returns error number &20401 if this fails because the transmit buffer is full.

All commands will be transmitted in the current MIDI transmission channel defined by the last MIDI_SetTxChannel.

NOTES:

- The size of the scheduler queue is 1023 commands.

- All calls to RxCommand with non-zero R1, from clearing the scheduler should be with non-decreasing schedule time. For efficiency, only the schedule time of the next item on the queue in the order they were put in.

- For backwards compatibility, the value of R1 on entry to this routine is ignored if SWI MIDI_FastClock has not been called since the module was initialised. This state can be interrogated with MIDI_InqSongPositionPointer. R 1, bit 3 is zero if in 'backwards compatible' mode.

- Scheduling too many commands for the same time will cause an overflow of the transmit buffer. The maximum size of the transmit buffer (in kbytes) can be read using SWI MIDI_InqBufferSize. If a transmit buffer overflow does occur with too many scheduled commands at the same time, it may be necessary to clear the scheduler using SWI MIDI_Init, since it can get into a state where it repeatedly tries to schedule the commands and fails.

MIDI_TxNote0ff (&404CB)

Transmit a MIDI Note Off command.

On entry	R0	=	note number, 0-127 (60 = middle C, 1 unit = 1 semitone).
	R1	=	key off velocity, 0-127
On exit	No change.		

MIDI TxNoteOn (&404CC)

Transmit a MIDI Note On command.

On entry	R0	=	note number, 0-127 (60 = middle C, 1 unit = 1 semitone).
	R1	=	key on velocity, 0-127.
On exit	No change		

MIDI_TxPolyKeyPressure (&404CD)

Transmit a MIDI Poly key Pressure (after touch) command. This will apply to any one voice and may affect volume, modulation or pitch depending on the setting of the receiver.

On entry	R0	=	note number, 0-127 (60 = middle C, 1 unit = 1 semitone).
	R1	=	key pressure value, 0-127.
On exit	No Change		

MIDI_TxControlChange (&404CE)

Transmit a MIDI Control Change command.

On entry	R0	=	control number 0-121, 122-127 reserved for channel mode messages, which can be transmitted using this command, or by using the Channel Mode SWIs; see the chapter *MID/ interface data specification.*
	R1	=	control value, 0-127.
On exit	No change.		

MIDI_TxLocalControl (&404CF)

Transmit a MIDI Local Control command with control number of 122, and R0 having one of the values defined below.

On entry	R0	= 0	local control off.
	R0	= 127	local control on.
On exit	No change.		

MIDI_TxAllNotesOff (&404D0)

Transmit a MIDI All Notes Off command.

MIDI_Tx0mniMode0ff (&404D1)

Transmit a MIDI Omni Mode Off command.

MIDI_TxOmniModeOn (&404D2)

Transmit a MIDI Omni Mode On command.

MIDI_TxMonoModeOn (&404D3)

Transmit a MIDI Mono Mode On command.

On entry	R0	=	M	where M is the current number of channels 1-16
On exit	No change			

MIDI_TxPolyModeOn (&404D4)

Transmit a MIDI Poly Mode On command.

MIDI_TxProgramChange (&404D5)

Transmit a MIDI Program Change command. The program referred to is the 'voice', 'tone' or 'patch' number in the receiver.

On entry	R0	=	program number, 0-127.
On exit	No change.		

MIDI_TxChannelPressure (&404D6)

Transmit a MIDI Channel Pressure command. This will affect all the notes on that channel, and may alter volume, modulation or pitch depending on the receiver setting.

On entry	R0	=	pressure value, 0-127.
On exit	No change.		

MIDI_TxPitchWheel (&404D7)

Transmit a MIDI Pitch Wheel command

On entry	R0	=pitch wheel change, 0-16383 (&3FFF). 8192 (&2000) is centre position value (no pitch change).
On exit	No change.	

MIDI_TxSongPositionPointer (&404D8)

Transmit a MIDI Song Position Pointer command. This automatically updates the internal copy which could affect the time-stamping of received data.

On entry	R0	=	Song Position Pointer, 0-16383 (&3FFF).
On exit	No change.		

MIDI_TxSongSelect (&404D9)

Transmit a MIDI Song Select command.

On entry	R0	=	song number, 0-127.
On exit	No change.		

MIDI_TxTuneRequest (&404DA)

Transmit a MIDI Tune Request command.

MIDI_TxStart (&404DB)

Transmit a MIDI Start command, reset the MIDI beat counter to zero (and the internal Song Position Pointer), enable automatic transmission of Timing Clock messages every 16 internal microbeats (or else timed by fast clock; see MIDI_FastClock), and disable reception of Start, Continue, Stop and Timing Clock messages. This affects all installed MIDI ports.

MIDI_TxContinue (&404DC)

The same as TxStart, but without resetting the beat counter.

MIDI_TxStop (&404DD)

Transmit a MIDI Stop command, and stop transmission of Timing Clock messages, thus allowing reception of Start, Continue, Stop and Timing Clock messages.

MIDI_TxSystemReset (&404DE)

Transmit a MIDI System Reset command.

Service_MIDI (&58)

Service call.

Service_MIDIAlive	= 0	module about to be initialised.
Service_MIDIDying	= 1	module about to be removed.

Enters module service with R 1 = Service_MIDI, and R0 = sub reason code.

This is necessary for advanced use only, where the actual address of the MIDI module or its workspace is being implicitly or explicitly used.

Event_MIDI (&11)

MIDI_DataReceivedEvent=0
receive buffer was empty and has received some data.

MIDI_ErrorEvent=1
an error has occurred in the background; use MIDI_InqError to find out what it is.

MIDI_ScheduleEmptyingEvent=2
the MIDI scheduler will empty within the next lOms. This event does not occur in cases where there are fewer than four free slots in the scheduler when the event would normally be triggered, which will only occur if more than 1020 commands are scheduled to happen, all within l Oms

Enters an event handler routine with R0 = Event MIDI and R 1 = sub reason code.

In simple BASIC programs, the SWIs may be called as in the following example:

```
SYS "MIDI_TxNoteOn",        60, 64
```

The two parameters are the values of R0 and RI. Case is important - upper and lower case must be used as above. The most efficient way to use these SWIs in BASIC is to define integer variables at the start of the program.

To find the SWI number from the string, OS_SWINumberFromString:

```
SYS "OS_SWINumberFromString",0,"MIDI_TxNoteOff" TO
TxNote Off%

SYS "OS_SWINumberFromString",0,"MIDI_TxNote0n" TO
TxNote On%

SYS "OS_SWINumberFromString",0,"MIDI _InqError"            TO
IngError%
```

Then to use them:

```
v%=64

a%=60

SYS NoteOn%, a%, v%

key%=INKEY (80)

SYS NoteOff%, a%, v%

SYS IngError%,0 TO Error%

IF Error% <> 0 THEN PRINT CHR$(Error%)
```

(the line beginning key% provides a delay of 80 centiseconds).

T iming

The timing mode can be:

- Unset
- Internal
- External

Internal timing mode is set if timing messages are being transmitted.

External timing mode is set if timing messages are currently being received and not transmitted or ignored.

Three possible types of time-stamp can be returned by RxCommand and RxByte depending on which of four internal timing modes is current. These four modes are:

- MIDI Timing Clock transmission counting. The time-stamp is the value of the MIDI beat counter, which is equal to the number of Timing Clock messages transmitted since the last Start transmitted.
- MIDI Timing Clock reception counting. The time-stamp is the value of the MIDI beat counter which is equal to the number of Timing Clock messages received since the last received Start command.
- The sound system beat counter (see the chapter on sound in the *RISC OS User Guide).* The time-stamp is the value of the sound system bar/beat counter. The bar counter is incremented every time the beat counter resets to zero, and the time-stamp is a combination of the sound system beat value and an incrementing bar value.

  The beat count is off by default, so BEATS must be set to a positive value to set the beat count in operation, and so give non-zero values for internal BAR/ BEAT time-stamping.
- Fast Clock Timing mode. The Fast Clock is reset and stated by SWI MIDI_FastClock, and incremented every millisecond. The timestamp is set equal to this count.

NOTE: The types of time-stamp are related to the different time periods, or beats, used. For the purposes of this guide, a sound system beat, as defined in the *RISC OS User Guide, is* referred to as a microbeat, in order to distinguish it from a MIDI beat.

One MIDI beat is equal to 16 microbeats, when microbeats govern the timing of MIDI beats.

MIDI Timing Clock transmission counting

This mode is set by the call MIDI_TxStart, if the fast clock is not running, and any received Start, Continue, Stop or Timing Clock messages are ignored. Incoming data is time-stamped with the current value of the MIDI beat counter, the value being returned in R1 on calling MIDI_RxCommand, or MIDI_RxByte.

The call MIDI_TxStart enables automatic transmission of Timing Clock messages. They can then be stopped with MIDI_TxStop, and restarted with MIDI_TxContinue. The frequency can be controlled by programming the sound system tempo, using Sound_QTempo (or TEMPO in BASIC). A MIDI Timing Clock message will be transmitted every 16 sound system microbeats. The song position pointer is equal to the value of the MIDI beat counter divided by six.

For example, if the sound system is programmed to the default tempo of &1000 (=100 per second), then, when enabled by MIDI_TxStart, Timing Clock messages will be transmitted at a rate of 6.25 per second, and the song position pointer will increment approximately once per second. For the maximum tempo of &FFFF the pointer will increment at a rate of 16.66 per second and Timing Clock messages will be transmitted at 100 per second.

NOTES: 1. The sound system beat counter must be started to enable this mode to work (for example, BEATS 100 in BASIC).
2. The CLI command *MidiStart enables Fast Clock Timing mode (see below), and NOT this mode.

MIDI Timing Clock reception counting

When Timing Clock messages are being transmitted, they take priority. If not, any Timing Clock messages received (preceded by Start) will cause the MIDI beat counter to increment. This can be prevented by disabling reception of timing messages, using MIDI_IgnoreTiming with R0=1.

The MIDI beat counter is updated automatically on reception of a Song Position Pointer message only if the MIDI beat counter is currently being controlled by received Real Time messages. The MIDI beat counter is set to the received value multiplied by six. The internal MIDI beat counter is always updated by the call MIDI_TxSongPositionPointer.

Sound system bar/microbeat counting

If Fast Clock Timing mode is not enabled, and if Timing Clock messages are neither being received nor transmitted, which may be because they were never started, or because they were stopped with a MIDI Stop command, then the received bytes are given a sound system bar/microbeat time stamp, with the value of microbeat in the bottom 16 bits, and an incrementing bar count in the top 16 bits. The bar counter is reset on transmission or reception of a MIDIStop, or on changing the value of BEATS (microbeats per bar), or upon incrementing beyond &FFFF.

Queueing of MIDI data to be transmitted on a future microbeat can be done using the general-purpose sound system call:

Sound_QSchedule

On entry:	R0 = the schedule period
	R 1 = &F000000 + SWI number to schedule.
	R2 = SWI parameter for R0
	R3 = SWI parameter for R1
On exit:	R0 = 0 if SW 1 successfully added to queue.
	R0 < 0 if queue is full and command failed.

NOTE: The queueing time is the value of the sound system microbeat counter and NOT the MIDI beat counter.

For details of the microbeat timing see the RISC OS *User Guide.*

The Sound Scheduler may be synchronised to an external source of MIDI Timing Clock messages by using SWI MIDI_SynchSoundScheduler with R0 = 1.

NOTE: For new applications it is preferable to use the MIDI Scheduler by calling SWI MIDI_TxCommand with R1 = Schedule time (see Fast Clock Timing mode below).

Fast Clock Timing mode

This mode is enabled by calling SWI MIDI_FastClock. Incoming messages will be stamped with the current Fast Clock time, unless timing messages are being received, in which case MIDI Timing Clock Reception Counting (see above) will apply. Scheduling to this time is done using the special MIDI Scheduler by calling the SWI MIDI_TxCommand with the schedule time in R1. This mode automatically set by the CLI command *MidiStart.

Reception of special messages

Most MIDI messages when received have no further effect than to be stored in the receive buffer to be read by the SWIs MIDI_RxCommand and, MIDI_RxByte, or by the MIDI interpreter. Certain messages, however, have other side-effects, and some are not stored in the receive buffer at all, making them invisible to the application, except by their side-effects. This behaviour can be re-programmed if required using MIDI_Init with special bits set.

Song Position Pointer as a side-effect updates the internal MIDI beat counter to the received value multiplied by six.

System Real Time messages, except for System Reset, are invisible to the application except in their side-effects.

Real Time messages

Side-effects on receiving System Real Time messages are:

Start	If Timing Clock messages are not being transmitted, i.e. the call MIDI_TxStart has not been made since the last Stop, then reset the internal MIDI beat counter, and the Song Position Pointer) and enable received Timing Clock messages to increment it.
Timing Clock	If external timing is enabled (see Start) then increment the MIDI beat counter. The Song Position Pointer is incremented on every sixth Timing Clock message received.
Stop	Disable external clocking. Received Timing Clock will have no further effect.
Continue	The same as start, without resetting the MIDI beat counter.
Active sensing	This will set the receiver into Active Sensing mode, so that if no message is received for more than 300ms for any reason (for example if the MIDI cable is pulled out), then all MIDI-triggered notes are turned off. The transmitter should send regular Active Sensing status messages, when enabled, in the absence of any other activity. An application program can be warned of an Active Sensing timeout by polling the error flag, or by enabling the MIDI error event.

Exceptions

It is the task of the application program to deal with exceptions and errors. For example, when Note On messages have been sent via the MIDI, and then an error occurs, or escape is pressed, the program must automatically send a matching set of Note Off messages, to avoid notes becoming stuck on. For this reason, if you are using the Sound Scheduler to schedule MIDI data, use Sound_Qlnit only with great care to avoid losing Note Off messages and leaving notes on in the receiver. Similar considerations apply to the MIDI Scheduler of course.

Removing sound modules

If a sound module is removed for any reason, the MIDI module may stop functioning normally. If this occurs, replace the sound module, reinitialise ALL Sound modules and Voice modules in ascending order of module number and type *RMReInit MIDI.

Operating system

The MIDI module is only compatible with RISC_OS version 3.

User timer

The User Timer (IOC timer 1) is used by the Fast Clock. This means that other software cannot use this timer while in Fast Clock timing mode, or if it does not use the correct claim SWI (OS_ClaimDeviceVector), but writes to it directly, it will make the fast clock go wrong.

MIDI interface data specification

Summary specification

MIDI equipped instruments contain a receiver and a transmitter, the receiver being optoisolated. It interprets and acts on MIDI commands. The transmitter sends messages in MIDI format via a line driver.

Data rate

31.25 (+/- I%) Kbaud, asynchronous. There is a start bit, eight data bits and a stop bit, so each serial byte consists of 10 bits lasting 320 microseconds.

Summary of status bytes

	Status D7-DO	No.of Data bytes	Description
Channel voice messages			
	&8n	2	Note Off
	&9n	2	Note On (velocity=0, note off)
	&An	2	Polyphonic Key Pressure (after touch)
	&Bn	2	Control Change if data byte I in the range 0-121
	&Cn	1	Program Change
	&Dn	1	Channel Pressure (after touch)
	&En	2	Pitch Wheel Change
Channel mode messages			
	&Bn	2	Channel mode message if data byte I in the range 122-127
System messages			
	&F0	any	System Exclusive
	&Fs	0 to 2	System common
	&Ft	0	System Real Time

	where	n=N-1; N is channel number.
		s=1 to 7
		t=8 to 15	

Introducing the HQ Sound Sampler

The HQ Sound Sampler, for use with the MicroPodule MIDI, provides a facility for sampling and digitising sound. The easy to use application provided allows sampling and playing back of sound through the computer's speakers. You can view samples and export them to disk for later play-back, for use in games, other applications, or musical compositions.

This section of the manual describes the application software and its function, for fitting instructions please refer to the instruction sheet provided.

Connecting up

Assuming you now have your MIDI-Sampler card correctly installed, you should find a dual purpose lead supplied. This consists of 5 leads protruding from a shrouded D-Type plug. Four of these leads are for the MIDI module and their use is described in the previous section. One of the leads terminates in a standard mono jack socket, it is into this that you should plug your sound source, or microphone. It must accept a 100KOhm input impedance. The shrouded D-Type connector must be connected to the MicroPodule.

Testing

To check that your Sound Sampler has been installed correctly, press F12 to get to the command line and type:

```
*modules
```

The system will now list all the currently available modules. Near the bottom of this list you should see:

```
MuSampler
```

If not, your Sound Sampler has not been installed correctly. Refer back to your supplier, or check the installation instructions, if you have made the installation yourself.

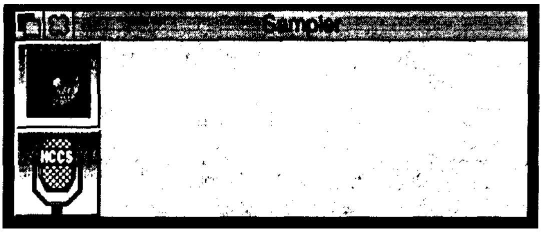
Assuming that MuSampler is on the list, you can now go onto the application instructions.

The !Sampler application

The !Sampler application allows the user to collect samples, view them, play them back and save them to disk. The sample rate, gain, duration and frequency are all configurable from the settings menu, which also provides an option for automatic recording when a sound triggers the Sound Sampler.

The !Sampler applicaton is loaded onto the icon bar, just like any other RISC_OS application, by double clicking the select (left mouse) button with the arrow on its icon, shown left. The Sound Sampler installs itself on the right hand side of the icon bar.

To start up the Sound Sampler, click on the select button whilst the mouse pointer is over the microphone symbol on the icon bar. This will bring up a window on the screen similar to the one below, depending on your version of RISC_OS and current screen mode:



The window supports standard RISC_OS style dragging etc. The two icons on the left are the function buttons; the speaker on the top is for play-back and the microphone on the bottom is for record.

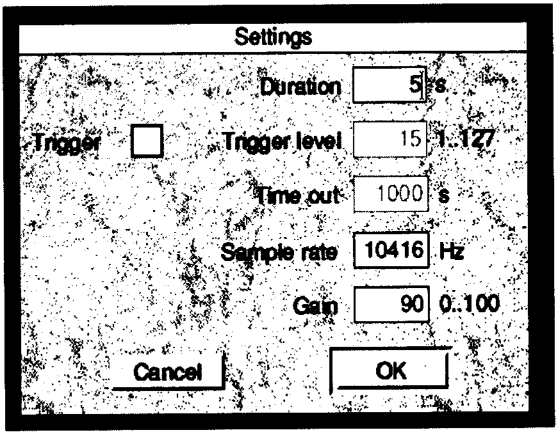
To see this in action, simply connect a suitable sound source, or microphone, and click on the record button. The main bar of the window will now read - ** Sampling ** - and the Sound Sampler will record whatever is present on the line, for a set time (see later for how to alter this, and other settings). When the Sound Sampler has finished an image of the waveform will be displayed in the empty box, to the right of the play and record buttons.

To play back your sample, simply click on the play-back button. This will cause the computer to replay your sample through the loudspeakers. The quality and volume of the sampled sound will depend a great deal on how you have the Sound Sampler configured. This will be dealt with on the next page.

When you have collected a suitable sound sample, you will wish to save it. This is done in the normal RISC OS way; position the mouse pointer anywhere over the window and click the middle menu button. Move to the save option and the sample icon, shown left, will appear. Drag this icon to a suitable directory on your hard/floppy disk drive and the sample will be saved. If you wish to view the sample at a later date, simply drag its icon onto the ! Sampler window and it will be re-loaded. If you want to play this sample, simply double click on the icon and it will play. Note: If you want to play samples on computers without MIDI hardware a play only module called SamplerNS is provided, simply load this before trying to play the samples. The SamplerNS module is public domain and can be distributed.

Changing the settings of the Sound Sampler

To alter the configuration of your Sound Sampler, move the icon over the window and click on the middle button. The menu now displayed has a settings option, select it. You will now see a window similar to the one below:



Settings

Trigger ☐

Duration 5 s

Trigger level 15 1..127

Time out 1000 s

Sample rate 10416 Hz

Gain 90 0..100

This is the settings box. To alter any of the settings, simply click on the required value and edit it in the normal way. The meanings of the settings are as follows:

Duration

This controls the amount of time that !Sampler records for. Note that it is not limitless, and will greatly depend upon the amount of RAM you have available to hold the sample. It also depends upon the sample rate; the more samples per second (in Hz), the more memory the sample will take.

Trigger level

If you click on the left hand box labeled 'trigger', the two boxes to the right will become active. Setting a trigger level causes !Sampler to start recording only after a sound above that level is heard. If no sound reaches that level then the Sound Sampler will stop trying to obtain one after the set 'time out'. The trigger level can be set between 1 and 127. The time out can be set up to a maximum of 10000 seconds. Note: the sampler will not start waiting for a trigger until the record button is pressed.

Sample rate

This is the number of times that the sound source is sampled per second (in Hz). The more samples per second, the higher the quality of the sound. There is a trade-off however; as the number of samples increases, so does the amount of memory required to store a second of sound. The maximum sample rate is 44.1Khz (44100). This is the standard for CD quality sound.

Gain

The volume at which the Sound Sampler records is controlled by this setting. If the sample is distorted on play-back then the gain is to high; if the sample is too quiet on play-back then the gain is too low. For general use, the best settings will be in the 90-100 range.

After setting your required values, click on the 'OK' button and they will be set accordingly. Sometimes the software may set a slightly different sample rate to the one you have chosen, in order that the hardware can efficiently digitise the sound.

SWI MuSampler_NewSample

Create a new sample buffer in RMA, optionally naming it. If there is not sufficient room in RMA an error will be returned. The returned handle must be used to specify this sample to any of the other SWIs.

On entry	R0	=	sample size in bytes.
	R1	=	pointer to sample name, maximum 16 characters, optional.
	R2	=	flags (must be 0 at present)
On exit	R0	=	sample handle (V clear), or error message (V set)

SWI MuSampler_Sample

Record into a previously created sample. SWI MuSampler_Poll may be used to monitor the progress of sampling.

On entry	R0	=	sample handle.
	R1	=	sound trigger level (1..127), or 0 to sample immediately.
	R2	=	sample rate in kHz, or 0 for default.
	R3	=	timeout in cs (in the event of no trigger), or 0 to wait forever.
	R4	=	attenuation level (0 is min. vol., 100 is max. vol.).

SWI MuSampler_AttachSample

Attach a sample to the playback voice 'MuSampler'. To play a sample it must be attached to the voice generator using this SWI. The voice must then be attached to a sound channel using SWI Sound_AttachVoice, and finally a sound command must be issued to play the sample. To play the sample at its correct pitch the note value for middle 'C' should be used.

On entry	R0	=	Sample handle.
	R1	=	Offset within sample to start playing from in bytes (0 means start).
	R2	=	Amount of sample data to play, 0 means whole sample.

SWI MuSampler_KillSample

Remove a sample from RMA. See also MuSampler_KillAndShrink.

On entry	R0	=	sample handle.

SWI MuSampler_SaveSample

Save a sample to a named file. Samples have the filetype &DF5.

On entry	R0	=	sample handle.
	R1	=	filename.

SWI MuSampler_LoadSample

Load a sample from a named file, and return the handle allocated to it. The file must have the correct file type.

On entry	R1	=	filename.
On exit	R0	=	sample handle.

SWI MuSampler_FindSample

Return the address and size of a sample, and the address and size of the data portion of the sample. The sample data is stored as a series of 8 bit excess 128 values (i.e. 0 is represented by 128, -1 by 127, 1 by 129 etc).

On entry	R0	=	Sample handle.
On exit	R0	=	unchanged.
	R1	=	base address of sample.
	R2	=	size of sample.
	R3	=	base address of raw data.
	R4	=	size of raw data.

SWI MuSampler_CopySample

Make a duplicate of a sample with a new name.

On entry	R0	=	sample handle.
	R1	=	name for new sample.
On exit	R0	=	new sample handle

SWI MuSampler_Squelch

Modify a sample to suppress any quiet portions.

On entry	R0	=	sample handle.
	R1	=	squelch level (0-127)

Any portions of the sample whose level is less than R 1 will be set to zero. Squelching only modifies the portions of samples between zero crossings to avoid clicks.

SWI MuSampler_Version

Return MuSampler version number.

On exit	R0	=	version number * 100 (e.g. 1.50 is returned as 150).

SWI MuSampler_SetFlags

Set or read a sample's internal flags. You should not alter the values of these flags but it may be useful to read them, for example to find out if a sample is playing.

On entry	R0	=	sample handle.
	R1	=	FOR mask.
	R2	=	AND mask.

New value = (Old value AND R2) FOR R 1.

On exit	R1	=	previous value of flags.
	R2	=	preserved.

The flags have the following meanings:

Bit 0	sample is playing.
Bit 1	sample is recording.
Bit 2	sample is waiting to play.
Bit 3	sample has been deleted, but is either playing or recording. It will be deleted when it finishes.
Bit 4	sample has claimed a voice (only used by *MuSampler_Play).
Bit 5	reserved.
Bit 6	shrink RMA as far as possible when freeing this sample. Used intemally by SWI MuSampler_KillAndShrink.
Bit 7+	reserved.

SWI MuSampler_Trim

Trim a sample to remove unwanted sounds at its start or end.

On entry	R0	=	sample handle.
	R1 Flags		
	bit 0	=	remove samples from start, number in R2.
	bit 1	=	trim start of sample, level specified in R3.
	bit 2	=	remove samples from end, number in R4.
	bit 3	=	trim end of sample, level specified in R5.
	bit 4	=	apply fade to remaining end of sample over R6 samples.
	R2	=	number of samples to remove from start if bit 0 of R 1 set.
	R3	=	level below which to trim samples if bit 1 of R 1 set.
	R4	=	number of samples to remove from end if bit 2 of R 1 set.
	R5	=	level below which to trim samples if bit 3 of rl set.
	R6	=	duration of tail fade in samples if bit 4 of rl set.

If bits 0 or 2 of the flags are set the specified number of samples will be removed from the start or end of the sample. If bits 1 or 3 are set, the start (end) of the sample will be removed up to the first sample whose value exceeds the level in R3 (R5).

If bit 4 of the sample is set the remaining sample data after any of the above steps have been performed will be faded over R6 samples.

SWI MuSampler_Echo

Add an echo effect to a sample

R0	=	sample handle.
R1	=	echo delay (may be -ve for'pre echo').
R2	=	echo amplitude (0-128).

An echo is added to the specified sample. The echo volume is in the range 0 to 128, where 0 is inaudible and 128 is an echo which is as loud as the original sample.

SWI MuSampler_SetPitch

Set the natural pitch of a sample

On entry	R0	=	sample handle.
	R1	=	pitch (&4000 is middle C).

Each sample 'knows' what its natural pitch is: they all default to middle C. If you were sampling a musical instrument you might sample a range of several octaves then use this call to adjust the natural pitch of each sample so that they were all normalised, and middle C would always sound like middle C.

SWI MuSampler_Poll

Monitor sampling progress

On entry	R0	=	sample handle.
On exit	R0	=	state:
		= 0	finished sampling.
		= 1	waiting for trigger level.
		= 2	sampling.

During sampling this call can be used to monitor progress. It should be used after MuSampler Sample. If a sample times out while waiting for a trigger level this call will return an error message indicating that the timeout'has occured, and sampling will be aborted.

SWI MuSampler_Summary

Get data for an amplitude envelope display.

On entry	R0	=	sample handle.
	R1	=	number of points in display.
	R2	=	buffer for data (one byte per point).
On exit	R2	=	filled buffer containing bytes in the range 0-127 indicating the overall amplitude of the corresponding 'slice' of sample.

It is useful when writing a sampling application to give the user some indication of the amplitude of the sample, so that volume adjustments may be made. This SWI performs the processing to prepare the data for a simple amplitude envelope display. It divides the sample into R1 slices then fills the buffer with values which represent the overall amplitude of each slice. By plotting these values connected by a solid line above and below the X axis of a graph a waveform display may be generated.

Bibliography

The *MIDI Specification Document* (no.MIDI-1.0, August 5th, 1983), is available from:

International MIDI Association,
11857 Hartsook Street,
North Hollywood,
CA91607,
USA.

This specification is included in:

MIDI for musicians by Craig Anderson, AMSCO Publication.

(ISBN 0 8256 1050 8).

Music through MIDI by Michael Boom, Microsoft Press.

(ISBN 1 55615 026 1).

HCCS Associates Ltd
575-583 Durham Road
Gateshead
Tyne and Wear
NE9 5JJ

Tel (091) 487 0760
Fax (091) 491 0431