Kurzweil 1000 Series
Developer Information

Kurzweil Music Systems, Inc.

Contents of this package:

1. Kurzweil 1000 Series MIDI Implementation

2. Kurzweil 1000 Series MIDI System Exclusive Messages

3. Kurzweil 1000 Series Binary Data Transfer Protocol
   How to talk to the K1000 over MIDI.

4. Kurzweil 1000 Series Object Overview
   Describes the basic structure of K1000 binary objects. Illustrated!

Kurzweil 1000 Series
MIDI Implementation

Kurzweil Music Systems, Inc.

1st Edition: March ’88

MIDI Modes

The 1000 series currently supports three modes of MIDI reception: Omni On/Poly, Omni Off/Poly and Multi. In Omni On mode, the channel number is ignored. In Omni Off (aka Poly) mode, only messages received on the basic channel are recognized. In Multi mode, all enabled MIDI channels are recognized.

Note On’s and Off’s

NOTE: The 1000 series products (in fact, all Kurzweil products) refer to Middle C (MIDI key number 60) as C4, in contrast to numerous software products which mistakenly call it C3. This is, as far as we know, an international standard to which we’ve been adhering since before we ever heard of MIDI. Included with this document is an appendix which lists the key numbers and their proper names.

The 1000 series responds to the entire range of MIDI key numbers although the actual, playable range depends on the selected program. (E.g., some of the sampled instruments have natural ranges which do not cover the entire keyboard.) It is also possible to restrict the MIDI key range on an individual MIDI channel via the master parameters. In general most programs respond over the range C0 (=12) thru C8 (=108).

The 1000 series also allows multi-layered programs, in which a single MIDI Note On may start multiple voices in the instrument. This technique is provided to allow creation of complex timbres from combinations of raw sounds; there is no efficiency gain by using layers. I.e., it takes the instrument just as long to start four layers from a single MIDI note on as it does to start four individual notes.

The lowest octave of key numbers, C-1 (=0) thru B-1 (=11), is available to control the intonation table reference key. When used with a suitable MIDI controller, this allows chromatic modulation in real-time while using a non-equally tempered intonation.

In the K1000 keyboard instrument, Note Off’s are transmitted using the MIDI Note Off message `$8x kkk vvv`. This can cause problems with certain simple minded MIDI processors (such as the Yamaha MEP4) which transpose the MIDI stream by only altering key numbers in Note On messages (`$9x`) on the assumption that the MIDI controller is sending Note Off’s as zero velocity Note On’s. Currently, there is no solution to this problem but future versions of the software will add a master parameter to select the type of Note Off (real Note Off or zero velocity Note On) which is transmitted.

There is a layer-level parameter which allows the Note Off message to be ignored. This is provided so that programs can be created which play through their envelopes completely, until each note decays to silence. If the envelope doesn’t decay, the note will sustain indefinitely.

In 1000 series software, both attack and release velocity are available (in normal and inverse form) as internal control sources; programs can be created which respond to these parameters. When a note is started, its release velocity is set to zero (and inverse release velocity is set to one). When the Note Off is received, the actual release velocity becomes available. Zero velocity Note On’s which are received are treated as Note Off’s with a velocity of 64.

Program Changes

The 1000 series responds to the full range of MIDI program change numbers. The numbers are mapped through an editable list (selected by the master parameter RxPMap) which selects the real, internal program number. Program

changes which reference non-existent programs are ignored. In general, it takes the instrument about as much time to change a program as it does to start a single layer note.

Control Changes

All MIDI control numbers are mapped thru a master table which may be replaced by a RAM copy.

Internally, the 1000 series represents all control values as signed, 15 bit fractions with a range of ±1. MIDI controls (except for the pitch wheel) are treated as uni-polar values with a range of 0 to 1. When a control source is used as a switch, it is considered to by ON if its value is greater than or equal to 0.5. This translates into an MSB value of 64 or greater. The details of how MIDI control values are mapped to internal control values are as follows:

The continuous control MSB's (control numbers 1 thru 31) and LSB's (control numbers 33 thru 63) are recorded for all sixteen MIDI channels. When an MSB message is received, the corresponding LSB is forced to zero if the MSB value is zero; otherwise the LSB is set to all ones. Thus, a control value of 0 yields an internal value of 0, while a control value of 127 ($7F) yields an internal value of -1 ($7FFF). Thus, it is desirable to transmit the MSB of any continuous controller first, followed by the LSB.

The continuous 'switches' (control numbers 64 thru 95) are treated similarly to the continuous control MSBs. When an MSB of zero is received, the LSB of the internal value is set to zero, otherwise it is set to all one's. Thus the internal value of any control is

```
value = 256 * MSB + 2 * LSB
```

In general, the 1000 series software makes no distinction between internal control sources (e.g., envelopes or LFO's) and external control sources (i.e., MIDI controls). This allows any MIDI control source to be patched to any control input.

Certain well-defined control numbers are given special treatment:

7 Volume MSB
39 Volume LSB
The volume control may be thought of as controlling an imaginary output VCA. Thus it is completely independent of the note amplitude. This can cause problems with certain wind controllers which transmit MIDI volume as a way of creating a real-time envelope. Switches are provided to ignore MIDI volume at both the layer and the MIDI channel. In addition, the mapping the control values to loudness is controlled by a master table which may be overridden by a RAM copy.

64 Sustain Pedal.
When this switch is on, playing notes are held until the switch is off or until they have decayed to silence. A layer-level parameter is provided to ignore the sustain pedal.

66 Sostenuto Pedal
This switch behaves like the sustain pedal, but only notes whose keys were down when the pedal was depressed are held. A layer-level parameter is provided to ignore the sostenuto pedal.

67 Soft Pedal
At note start, the value of this control (0 to 1) is multiplied by the layer-level soft pedal range parameter (±dB) and the result is subtracted from the initial amplitude of the note. The sign of the range parameter determines whether the notes get louder or softer. When this control is used with a switch pedal, its value will be either 0 or 1, but since it responds to the full range of MIDI control values, it can actually be used as a continuous attenuator. Once a note has been started, this control has no effect.

69 Suspend Pedal (aka Freeze Pedal)
This switch behaves like the sostenuto pedal, but in addition to sustaining the note, the envelopes are frozen as well. A layer-level parameter is provided to ignore the suspend pedal.

121 All Controls Off

When this message is received, all controls are reset to zero, with the following exceptions: volume (#7 and #39) is set to maximum ($7FFF), balance (#8 and #40) and pan (#10 and #42) are set to center point ($4000).

122 Local Control On/Off

This message is only recognized by the K1000 on the basic MIDI channel (it is ignored in rack-mount equipment).

123 All Notes Off

The All Notes Off message is recognized in all MIDI modes. In particular, it is not ignored in Omni On mode. This is contrary to the MIDI spec but represents the unanimous opinion of our users.

NOTE: most Roland equipment (e.g., MKB-1000, D-50, etc.) sends an All Notes Off message when all the keys are released. In order to deal with this, the 1000 series includes a master parameter which allows you to ignore the all notes off message. We call it the "Roland Switch." And I just discovered (as I am working on this document) that Kawai equipment exhibits the same behavior.

124 Omni Off

125 Omni On

When not in Multi Mode, the Omni On/Off messages are recognized on the basic MIDI channel only. In Multi Mode, these messages are ignored. This is contrary to the MIDI spec but represents the unanimous opinion of our users.

When the instrument is in edit mode, the following control numbers are also recognized:

6 Data Entry Slider MSB

38 Data Entry Slider LSB

96 Data Increment

97 Data Decrement

The data entry slider (control numbers 6 and 38) and the data increment/decrement buttons (control numbers 96 and 97) are recognized when the instrument is in edit mode.

98 Non-Registered Parameter Select LSB

99 Non-Registered Parameter Select MSB

When the instrument is in edit mode, the non-registered parameter select MSB selects the edit menu and the LSB selects the parameter within the menu. However, because the menu selection varies based on high or low level editing mode and in some cases (e.g., envelopes) the actual parameter list varies, there is no absolute mapping between select values and actual parameters.

Pitch Wheel

Currently, the pitch wheel is the only MIDI control which has an internal range which is bipolar (i.e., ±1). The actual conversion from MIDI data values to internal form is

```
pitch wheel value = 256 * MSB + 2 * LSB - $4000
```

Future implementations of the software will provide an absolute value pitch wheel control source with a range of 0 to 1 in both directions, to allow effects to be tied to pitch bending in either direction.

Mono and Poly Pressure

Both Mono Pressure and Poly Pressure messages are recognized. Internally they are treated as additional control sources but their internal values are derived by a mapping table rather than a direct conversion. Note that both pressure sources are treated as separate entities. Thus it is possible to create programs which have separate responses to

both Mono and Poly Pressure, provided you have a MIDI controller which transmits both. On the other hand, a program which responds to Poly Pressure will not respond to Mono Pressure, and vice versa. Future versions of the software will provide some master parameter to allow global selection of the pressure source.

Poly Pressure messages are only recognized for keys which have notes playing. Note that Poly Pressure messages transmitted after a Note Off message has been received may still be acted upon if the note is still playing (e.g., as a result of a slow decay). This can actually occur if you strike the same key twice. If the envelope has a long decay, the pressure messages for the new note will also affect the previous note.

System Common/Real-Time

All system common (except for System Elusives [SIC]) and real-time messages are ignored.

Note Names and Key Numbers

Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	119	
9	120	121	122	123	124	125	126	127				

Kurzweil 1000 Series
System Exclusive Messages

Kurzweil Music Systems, Inc.

2nd Edition: April ’88
1st Edition: February ’88

The 2nd edition of this document provides a correction to the front panel button code list.

These notes describe the current state of system exclusive messages in the K1000 series. All values are expressed in hexadecimal.

Version Request

The K1000 responds to the standard MIDI version request message:

```
$F0   $7E <device ID> $06 $01 $F7          ; request
```

The 1000 defaults to a device ID of $00, although it may be set to any value from $00 to $7F

```
$F0   $7E <device ID> $06 $02              ; response
      $07                                  ; Kurzweil ID
      p1 p2 p3 p4                          ; product ID
      e1 e2                                ; engine software version
      s1 s2                                ; setup software version
$F7
```

p1 through p4 represent four hexadecimal numerals which constitute the product ID. p1 is the major product heading (150 FS, 250, or 1000), and p2 through p4 distinguish the various models within the major headings. The product code for a 1000 GX, for example, would be: p1 = $64; p2 = $01; p3 = $04; p4 = $00. The table below shows the codes for all Kurzweil products with SysEx capabilities.

ID#				**Product**
$15				K150
$19				K250 or 250RMX
$64	$01	$00	$00	1000PX
	$01	$01	$01	PX Plus
	$01	$02	$00	1000SX
	$01	$03	$00	1000HX
	$01	$04	$00	1000GX
	$01	$05	$00	AX Plus
	$01	$05	$02	1200 Pro
	$02	$01	$00	K1000 SE
	$03	$01	$01	1000EX
	$04	$01	$00	EGP
p1	*p2*	*p3*	*p4*	

e1, e2, s1, and s2 represent the hexadecimal values indicating the software version of the 1000. e1 and e2 indicate the engine (operating system) software, which is common to all 1000 Series products. s1 and s2 indicate the setup software. It is the setup software which distinguishes a GX from a PX, etc. For both engine and setup software, the first numeral is the major version number, and the second is the subversion, if any. For example, version 1.0 would be represented as `$01 $00`. Version 2.14 would be `$02 $0E`.

Remote Front Panel

The remote front panel messages allow control of the 1000 series over MIDI. The format of the message is

```
$F0 $07 <device ID> $64 $01 <buttons> $F7
```

where $64 is the major product ID (i.e., 1000 series) and <buttons> are any number of button codes:

Code	Button	
$00	digit '0'	(K1000 only)
	...	
$09	digit '9'	(K1000 only)
$10	play/edit	
$11	mode/layer	
$12	chan/menu increment	
$13	chan/menu decrement	
$14	chan/menu incr & decr (double press)	
$15	prog/param increment	
$16	prog/param decrement	
$17	prog/param incr & decr (double press)	
$18	value increment/yes	
$19	value decrement/no	
$1A	value incr & decr (double press)	
$1B	enter	(K1000 only)
$1C	store	(K1000 only)
$20	bank 'A'	(K1000 only)
$21	bank 'B'	(K1000 only)
$22	bank 'C'	(K1000 only)
$7F	send display	

Whenever the special "send display" button ($7F) is sent, the K1000 responds with

```
$F0 $07 <device ID> $64 $02 <display text> $F7
```

where <display text> is the contents of the K1000 display as ordinary ASCII characters.

Dump Request Message

```
$F0   $07 <device ID> $64 $03
      <type-msb> <type-lsb> <id-msb> <id-lsb> <RAM-flag>
$F7
```

This message may be used to request a dump of any object or group of objects in the 1000's memory. The type and ID number select the object to be dumped. A type of $00 means all types and and ID number of $00 means all objects of the requested type. If the RAM-flag is true (i.e., not zero) only objects stored in non-volatile RAM will be dumped. For example, to request a dump of all RAM based programs, use:

```
$F0 $07 <device ID> $64 $03   $00 $50  $00 $00  $01  $F7
```

To request a dump of the Master Parameter Table, use:

```
$F0 $07 <device ID> $64 $03   $00 $42  $00 $00  $01  $F7
```

To request a memory dump (User Objects and RAM), use:

```
$F0 $07 <device ID> $64 $03   $00 $42  $00 $10  $01  $F7
```

Channel Setup Message

```
$F0   $07 <device ID> $64 $04
      <chan-a> <mode-a> <chan-b> <mode-b> ...
$F7
```

This message may be used to set the MIDI mode (Omni, Poly, or Multi) and to enable or disable specific MIDI channels. If <chan-a> (or subsequent) = $00, then it signals a mode change. A value of $01 for <mode-a> (or later) indicates Omni mode, $02 is Poly, and $03 is Multi.

If <chan-b> or subsequent is $01 through $10 (16), it is read as a MIDI channel number. In this case, the following <mode> byte value determines whether that MIDI channel is active. $00 enables the channel, and $01 disables it. For example, to put the instrument in Multi mode, with channels 1 thru 4 enabled, use:

```
$F0   $07 <device ID> $64 $04 $00 $03     ; multi mode
      $01 $00 $02 $00 $03 $00 $04 $00     ; enable 1-4
      $05 $01 $06 $01 $07 $01 $08 $01     ; disable 5-8
      $09 $01 $0A $01 $0B $01 $0C $01     ; disable 9-12
      $0D $01 $0E $01 $0F $01 $10 $01     ; disable 12-16
$F7
```

Kurzweil 1000 Series
Binary Data Transfer Protocol

Kurzweil Music Systems, Inc.

2nd Edition: April '88
1st Edition: February '88

Notes on 2nd Edition

The 2nd edition of this document corrects an error in the description of the data packet checksum. The checksum is computed over the data bytes only and does not include the packet number and size as the previous document described.

Introduction to the 1st Edition

These notes describe a simple(?) mechanism for reliable binary (eight bit) data transmission over MIDI. The implementation is a two level approach; the transmission of binary information is separated from the purpose (e.g., file transfer) of the information.

First, a basic packet level protocol is defined which allows for bidirectional exchange of binary (eight bit) data packets with error checking and re-transmission. This protocol features a synchronization sequence which allows both parties to negotiate details such as transmission speed (to allow for higher-than-MIDI baud rates), packet size and number of outstanding packets (for machines with large buffers).

Within the context of the packet protocol, higher level protocols may then be defined to implement various forms of data transfer (remote file systems, interactive streams, etc).

This protocol, as described, is currently implemented on the new 1000 series of rack-mount MIDI expanders as well as the new K1000 keyboard. A description of the 1000 data transfer messages is also included, to illustrate how the packet protocol may be used.

The Packet Level

The following description assumes a bidirectional link (closed loop) between the two parties. There is a brief discussion at the end of this section on how the protocol behaves in an open loop situation.

Message Format

All messages are transmitted as standard Kurzweil system exclusive messages of the form:

```
$F0 $07 <dev-ID> <mesg-ID> <data ...> $F7
```

Normally, in Kurzweil system exclusives, the <mesg-ID> byte is actually a product ID. Since this protocol is intended to apply to a variety of products, we use a certain range of the number space for message ids:

$78-$7B	Sync Messages (levels 0 thru 3)
$7C	Data Packet
$7E	Data Packet acknowledged (ACK)
$7F	Data Packet not acknowledged (NAK)

The Hand Shake

To exchange data packets (in either direction), the parties at either ends of the MIDI cable must be in sync. Not physical sync, since we are transmitted data asynchronously, but logical sync, in which each party begins in the

same state. The synchronization process consists of exchanging handshake messages which specify the desired transmission parameters (speed, packet size, etc.). These messages have the form

```
$F0 $07     <dst-ID> <syncN> <src-ID>
            <xSpeed> <nPacks> <pSizeH> <pSizeL> $F7
```

The elements of this message are

dst-ID The device ID of the destination. SYNC0 ($78) messages may be sent with a device ID of 127 as a general request for sync.

syncN The message ID ($78 thru $7B), which includes the party's current synchronization state.

src-ID The device ID of the transmitter (0 thru 126).

xSpeed The desired transmission speed (1x, 2x, 4x, etc.).

nPacks The maximum number of outstanding packets (1 thru 127).

pSize The maximum number of eight bit bytes that will be transmitted in a packet (a 14 bit value sent as two seven bit values).

All sync sequences begin at level 0 with both parties transmitting at standard MIDI speed. A party can transmit SYNC0 messages to actively make a connection or it can wait passively for the arrival of a SYNC0 message. When either party receives a SYNC0 message, it should respond by transmitting a SYNC1 message. Here, the transmission parameter negotiation takes place: each party begins by declaring its maximum capability and then lowering its parameters in response to SYNC1 messages from the other party, until a common denominator has been achieved. Then each party switches to level 2. If a speed change is required because the parties have agreed on a higher transmission rate, they switch speeds before transmitting their level 2 messages. Once the parties exchange identical SYNC2 messages, they switch to level three. Once each party has received a valid SYNC3 message, synchronization has been achieved and the exchange stops. Now the parties are able to exchange data packets.

Data Packets

Each data packet message has the form:

```
$F0 $07 <dst-ID> $7C <src-ID> <pktNum> <pSize> <data ...> <chkSum> $F7
```

The elements are

dst-ID The destination device ID (0 thru 126).

src-ID The source device ID (0 thru 126).

pktNum The packet number. Packet numbers will sequence from 0 to 127.

pSize The number of eight bit bytes contained in the packet (two seven bit values concatenated, MSB sent first). Must be between zero and the agreed upon maximum. The actual number of data bytes in the message will be larger (see below).

data The binary data, transmitted in a seven bit format (see below).

chkSum A fourteen bit check sum accumulated over the data bytes only. The check sum is calculated by rotating the 16 bit sum left one bit and then adding each seven bit value. Only the low seven bits of each byte are transmitted in the packet (i.e., the sign bits are dropped), MSB first.

Binary Data Transmission Format

The simplest transmission scheme is "nibble-izing," sending each data byte as two four bit values. This is the easy to read but it doubles the transmission time. The most efficient packing format (short of some exotic compression scheme like Huffman encoding) is to send the low seven bits of each byte, with every seven data bytes followed by one byte containing the seven sign bits. Thus, the seven binary data bytes

```
AAAAaaaa BBBBbbbb CCCCcccc DDDDdddd EEEEeeee FFFFffff GGGGgggg
```

are transmitted as

```
0AAAaaaa 0BBBbbbb 0CCCcccc 0DDDdddd 0EEEeeee 0FFFffff 0GGGgggg 0ABCDEFG
```

If the number of data bytes is not a multiple of seven, a short chunk is transmitted. E.g., if three bytes are left over

```
AAAAaaaa BBBBbbbb CCCCcccc
```

they are transmitted as

```
0AAAaaaa 0BBBbbbb 0CCCcccc 00000ABC
```

This scheme requires that the receiver know the length of the stream, either in logical size (i.e., the number of eight bit bytes) or physical size (the number of seven bit bytes). In general, to transmit N eight-bit bytes requires N + (N + 6)/7 seven bit bytes (assuming truncating integer division). Conversely, receiving M seven bit bytes yields 7 * (M / 8) + (M mod 8 - 1) eight bit bytes.

Since the packets are delimited by the system exclusive message, their length is self defined. Thus, within the packet itself, we include the size as the number of eight bit bytes contained in the data. While this may seem counter-intuitive, we believe it to be a more convenient number since it represents the size of the buffer into which the data is unpacked (see the attached code samples).

Response to Data Packet

```
$F0 $07 <dst-ID> $7E <src-ID> <pktNum> $F7      (ACK)
$F0 $07 <dst-ID> $7F <src-ID> <pktNum> $F7      (NAK)
```

Successfully received packets are ACKed. If an error or timeout occurs during packet reception, the receiving party will transmit a NAK message. Upon receiving a NAK, the transmitting party will re-transmit the incorrect packet.

Open Loop Behavior

Timeouts during a sync sequence will cause the transmitting party to reduce its requirements to a minimum (lowest speed, one packet, 128 byte packet size) and begin data transmission. Since it knows that the other party did not respond to sync, it can transmit packets at a default rate, without waiting for timeouts on ACKs.

Data packets received by party which is not in level three sync are ignored. Thus, a receiver which may operate in either mode needs some kind of front panel control to enable the reception.

Bypassing Sync

It is also possible to allow a receiver to power up in level 3 sync state (given default parameters of normal MIDI speed, 1 outstanding packet and a reasonable buffer size, say, 128 bytes). This would allow it to transmit and receive data packet messages without going thru the handshake sequence.

Message Summary

State	Message	Condition	Response	Newstate
SYNC0				
	SYNC0		SYNC1	SYNC1
	other		none	
SYNC1				
	SYNC0		SYNC1	
	SYNC1	mismatch	SYNC1	
	SYNC1	match	SYNC2	SYNC2
	other		none	SYNC0
		timeout		SYNC0
SYNC2				
	SYNC2	match	SYNC3	SYNC3
	SYNC2	mismatch	SYNC0	SYNC0
	SYNC3	match	SYNC3	SYNC3
	SYNC3	mismatch	SYNC0	SYNC0
	other		none	
		timeout		SYNC0
SYNC3				
	SYNC2	match	SYNC3	
	SYNC3		none	
	DATA	correct	ACK	
	DATA	error	NAK	
		rcv timeout	NAK	
		NAK limit exceeded		SYNC0
	ACK		transmit next	
	NAK		re-transmit current	
		xmt timeout	re-transmit	
		retry limit exceeded		SYNC0
	other		SYNC0	SYNC0

K1000 Data Transfer Protocol

The higher level protocol consists of a stream of messages which exist at a level above the packet protocol, which simply acts as a delivery vehicle. In the K1000 series expanders, this higher level protocol implements a simple remote file system which allows a remote computer to manipulate the contents of the K1000's object heap memory.

Message Format

```
100 message-id data...
```

Each message begins with its protocol ID byte (in this case, 100, the product ID for the K1000 series) and a message ID. The format of the remaining data is determined by the message type. Messages are divided into two categories, requests and responses. For convenience, request message types are even numbers and the response type for any message is its request type + 1.

In the following, all messages are in eight-bit binary! They must be formatted, then transmitted using the packet protocol described above. Also, the notation [n] indicates the size of the item in bytes. If omitted, the size is one byte.

Directory Request/Response

```
100  0  type idno
100  1  type idno size[4] zone[4] name[N]
```

Used by the remote device to request information about objects contained in the K1000's memory. An `idno` of zero means all objects of a given type. `Size` gives the size of the object in bytes, `zone` identifies the heap space in which the object resides, and `name` is the name of the object (if it has one) as a NUL-terminated string.

The when a request for info about all objects of a given type is received, the K1000 responds with a series of messages. The last of these is a null message (`idno` of zero) to indicate the end of the list.

Dump Request/Response

```
100  2  type idno size[4] offset[4]
100  3  type idno size[4] offset[4] data[N]
```

Used to read objects (or portions of objects) from the K1000's memory. A single read request will elicit multiple responses if request size is larger than the available data size in the message.

Create New Object

```
100  4  type idno size[4]
100  5  type idno size[4] zone[4]
```

Used to create new objects in the K1000's memory. If the `size` item in the response message is zero, the object could not be created and the `zone` item is an error code. Otherwise, `zone` indicates the heap zone name in which the object will reside. New objects may have the same ID as ROM objects; in this case the RAM object will "hide" the ROM object.

Write Request/Response

```
100  6  type idno size[4] offset[4] data[N]
100  7  type idno size[4] offset[4]
```

Used to write data into objects (or portions of objects). When writing into a newly created object, the series of write requests should be terminated by a zero length write message (i.e., an EOF signal).

Delete Object

```
100  8  type idno
100  9  type idno
```

Used to delete objects from the K1000's memory. Deleting a RAM object which was overriding a ROM object of the same ID will "uncover" the ROM object. Thus, if one is maintaining a directory of K1000 objects, the delete request should be followed by an info request to determine if a ROM version of the object exists.

Kurzweil 1000 Series
Object Overview

Kurzweil Music Systems, Inc.

1st Edition: February'88

Objects In General

The ROM and RAM memory of the 1000 series is organized as a collection of objects which are identified by a unique type and ID number. Any object may be up or downloaded by using the binary data transfer protocol (described in a separate document). Besides programs (aka patches) other object types exist (e.g., LFO shape tables) and any ROM object may be replaced by a RAM version.

Object Format

The first two bytes of each object contain the type and ID number of the object. E.g., in a program, the type is `progType` and the ID number is the the program number (typically one greater than the MIDI program number). While some objects have a fixed size (in which the format of the remaining data is determined solely by the type of the object) most objects are variable sized; the second word of the object is the size of the object in bytes. The size may be odd but the software assumes that all objects begin on even boundaries and pads the size to an even number of bytes.

Each variable sized object consists of 1) a header whose size is determined by the object type, 2) an optional name (presence determined by type), NUL-terminated and padded to an even number of bytes and 3) the extension data whose size is just the size of the object less the size of the fixed header and the name (if present). The contents of the extension data is arbitrary; for certain objects (programs and layers) the extension data contains more objects (which may be of fixed or variable size). Thus, editor/librarian software must be prepared to deal with a hierarchy of nested objects.

Programs, Layers, Etc.

The most visible user object is the **program**, which is basically a shell which contains the layer objects (up to four in the current 1000 series) as well as objects which define global control sources (LFOs and ASRS). The information in the program header is small, consisting of some voice allocation parameters (stealing and polyphonicity limit) and an alternate MIDI program number.

The majority of the information needed for starting notes is contained in the **layer** object. Besides the stuff in the layer header, the layer's extension data contains objects which define the local control sources (LFOs, ASRs and envelopes).

Many of the items in objects are ID numbers of other objects; others are enumerations where the actual value of a parameter is obtained by using the parameter as an index into a table. As an example, lets look at the description of the LFO parameter block:

```
/*
 * LFO parameter
 */
typedef struct {
       uByte   lfob_type;            /* = lfoType */
       uByte   lfob_idno;            /* = 1 or 2, 8 or 9 for globals */
       uByte   lfob_rfu;
       uByte   lfob_flags;           /* initial phase, etc */
       uByte   lfob_shape;           /* ID# of shapeType object */
       uByte   lfob_rtCtl;           /* rate control (enum table #1) */
       uByte   lfob_rtMin;           /* min rate (enum table #4) */
       uByte   lfob_rtMax;           /* max rate (enum table #4) */
} LFOB;
```

In this structure the waveshape of the LFO is determined by the `lfob_shape` parameter, which is the ID# of an LFO shape table (i.e., a `shapeType` object). Thus to display the name of the waveshape (e.g., 'Sine' or 'Rising Saw') one needs a list of `shapeType` ID's and names. Similarly, the LFO's rate control and range are obtained by indexing into the appropriate master tables (tables #1 and #4).

Most stuff within the layer block is fixed size, with two exceptions: the envelopes and two tables (ID numbers 17 and 18) used by the high level effects (absent if the program uses modular effects). The envelope header just contains the number of segments in each section (attack and release) with the segments themselves immediately following the header.

Note: although the on-board editor allows only eight segments per section, the note control software supports up to 128 segments per section. However, trying to edit (in the 1000) an envelope with more than eight segments in a section will probably cause the editor to crash!

Objects within the program or layer are parsed in sequential order. Scanning stops at the end of the data or when an invalid object is detected. Valid objects which are not recognized by the note control software are ignored (e.g., the two tables mentioned above are used only by the editor). If two objects have the same ID number, the last one seen is the one that is used.

The ID numbers of the various objects are significant. E.g., each layer may have an LFO #1 and/or #2. LFOs with IDs other than these are ignored (although future instruments may support more LFOs per voice). Global LFOs, which are contained in the program data, have ID numbers beginning with nine (e.g., gLFO 1 is really LFO #9).

Note: Early on in the development, we decided that any local object (i.e., with an ID number of 1 thru 8) appearing in the program data would apply to all layers (unless overridden by a specific object appearing in the layer data). We haven't actually done this yet, but will probably do so in a future release of the software to allow limited form of data compression. Be prepared!

Within programs and layers, table ID numbers 1 thru 63 are reserved (e.g., in the next release, a special table ID may be designated to contain MIDI data to be transmitted on program change). Other numbers may be used, e.g., to add comments, copyright info, etc.

Keymaps and Sound Blocks

Each layer object contains the ID of a **keymap**; this object is a variable size mapping table used to convert MIDI key numbers and attack velocities into pitch and loudness and sample selection. Each keymap in turn refers to one or more **sound blocks**, which are collections of sound file headers (which contain the low level information needed to play back the samples). Since these are variable-sized structures with several optional sections, a diagram of the overall layout is included.

The variable data portion of a keymap consists of one or more byte per key arrays. These include one or two bytes of optional tuning information, optional volume adjust, optional sound block IDs and sound file IDs. Keymaps are also organized into multiple timbre levels which are automatically selected by key velocity.

Note: Currently, a keymap always contains a byte per key array of sound file header numbers. A future optimization will allow for special keymaps which reference a single soundfile (e.g., an attack noise) by storing the sound file ID in the headers

The sound blocks contain lists of sound file headers. These are the actual descriptions of the samples, along with the so-called "natural envelopes" needed for proper playback.

Other Objects

The actual waveshapes generated by the LFOs are stored as `shapeType` objects. All incoming MIDI program numbers are converted to internal program numbers using MIDI program lists (`mlistType`). Intonation tables (`itblType`) allow tuning of the scale away from equal temperment. Velocity maps (`vmapType`), in combination

with several of tables described below, determine the mapping of MIDI key velocities into loudness and control sources.

Demo Song Objects

(excerpts from the V5 Software addendum manual)

A RAM based SONG (Demo) object is basically a MIDI File, Type 0, with a 1000 Series object header prepended to it. Demo songs may be loaded, named, renumbered, or removed with ObjectMover. A separate program is needed to turn MIDI files into object files.

There may be more than one Demo song in an object file, in which case, there would be multiple song headers. All song objects must be word aligned. The Demo player will play them in the order of their resource IDs, in a repeating cycle.

It is recommended that your program remove, or truncate, text meta-events from the data stream (to save RAM space). Only the tempo meta-event is interpreted by the 1000.

The block size in the object header is the size of all the objects in file, negated. The blockSize must include everything from itself to the end of file, so you will calculate `DO_blockSize` like this:

```
demoFile->DO_blockSize = -(fileSize - 32);
```

The object size itself is calculated in a similar way. Object size should include all data from the type and id fields, to the end of your song. Because this is a 16 bit field, there is a 64K maximum size for demo objects.

The tempo field (`DS_tempo`) is a precalculated clock increment providing the initial tempo of the song. This is a 32 bit field which is added to the Demo clock every 2 milliseconds. The Demo clock counts off 1/480 of a quarter note times, with a 16 bit fraction. Your program can calculate the clock increment like this:

```
myDemo->DS_tempo = (long)bps * 1048L;
```

Where ’bpm’ is the desired "beats per minute" for your song. If you are working from the MIDI file’s "microseconds per beat" value, the formula becomes:

```
myDemo->DS_tempo = (60000000 / uspb) * 1048L;
```

Following the song header, comes the name field. This name may be any length (up to 64 chars) and must be null terminated. The MIDI File song data starts on the next even address after that.

Tables

Table objects are actually miscellaneous, one-of-a-kind objects. All tables are variable sized, unnamed objects; the format of the extension data is determined by the particular table’s ID number. The following is a list of some of the more interesting tables organized by ID number:

#	Description
1	Control Source Enumeration Table gives name, type and ID# for both MIDI and internal control sources.
2	Envelope/ASR Time Enumeration. Determines the actual times for envelope and ASR segments. Non-linear.
3	Enumeration of note start delays. Non-linear.
4	Enumeration of LFO rates. Non-linear. Contains offsets into table #27.
5	Enumeration of Envelope Scale Factors. Contains offsets into table #6. Uses 5% resistor value scale.
6	Logarithmic Multiplier Table used to scale envelope rates. Table #5 contains offsets into this table.
11	MIDI control number map. Incoming MIDI control change messages have their control numbers mapped thru this table. Allows re-assignment of controls.
16	Master parameter data. Contains most everything that is editable in the master menu. Snapping a copy of this object captures the entire state of the machine (MIDI channel and mode, program assignments, etc).

17 Object mapping table (contained in program only). Used by the high-level effects editor to map pseudo parameters to real parameters.
18 Pseudo parameter table (program only). Contains the actual values of the high-level pseudo parameters.
21 Key velocity to loudness curve (combined with velocity map).
22 Key velocity to control source value curve (combined with velocity map).
23 Mono/Poly pressure to control value curve. Default is linear.
24 Volume control table (MIDI control value to loudness). Non-linear.
25 Balance control table. Equal power crossfade curve.
27 LFO phase increments. Table #4 contains offsets into this table.
28 Enumeration of bi-polar envelope segment values. Converts percentages into output levels. Linear.
29 Enumeration of amplitude envelope segment values. Converts percentages into loudness levels. Non-linear.

Note: The master parameter block, which is currently table #16, is soon to become a distinct, named object. Thus, the instrument will be able to store multiple sets of master parameters. We haven't decided exactly how to do this, yet, so stay tuned for more info...

K1000 Object Types

Type#	Hex	Description	Named?	
66	$42	Master Table		
68	$44	LFO Shape	yes	
69	$45	Sound Block	yes	
70	$46	Keyboard Map	yes	
71	$47	MIDI Program List	yes	
72	$48	Edit Menu Definition		
73	$49	Edit Menu Group		
74	$4A	Edit Menu Entry List		
75	$4B	Intonation Table	yes	
76	$4C	Compiled Effects Descriptor	yes	
77	$4D	(Rcv) Velocity Map	yes	
78	$4E	Rcv PressMap	?	v5
79	$4F	Editor Descriptor		
80	$50	Program Data Block	yes	
81	$51	Layer Data Block	yes	
82	$52	ASR Parameter Block		
83	$53	LFO Parameter Block		
84	$54	ENV Parameter Block		
85	$55	EFX Parameter Block		
86	$56	INVNEG Parameter Block		
87	$57	MXR Parameter Block		
91	$5B	Demo Song	yes	v5
94	$5E	Program List	?	v5
95	$5F	Bin Map	?	v5

K1000 Database Master Tables

ID#	Description	Where	
1	Control Source Enumeration	EROM	
2	ENV/ASR Time Enumeration		SROM
3	Delay Time Enumeration		SROM
4	LFO Rate Enumeration		SROM
5	ENV Ctl Scale Enumeration		SROM
6	ENV Ctl Log Multiplier		SROM
7	Playback Rate Compensation Table (2)		SROM
8	Edit Parameter ID Table	EROM	
9	Edit Menu Position Table	EROM	
10	Editor Default Object Table	EROM	
11	MIDI Control Mapping Table	EROM	
12	Front Panel Button Table (4,5)		SROM
13	Diagnostic Tables (4)		SROM
14	Product Configuration Table (3,4,6)	EROM	SROM
15	Arnold Configuration Table (1)		SROM
16	Prototype Master Data Block (5)		SROM
21	Velocity to Loudness Curve		SROM
22	Velocity to Control Source Curve		SROM
23	Pressure to Control Source Curve		SROM
24	MIDI Volume Control Table	EROM	
25	Equal Power Attenuation Curve	EROM	
26	Playback Rate Table	EROM	
27	LFO Phase Increment Table	EROM	
28	ENV Value Enumeration	EROM	
29	Amp ENV Value Enumeration	EROM	
30	Amp ENV Attack Shape Curve	EROM	

Notes:

1. Determines sound engine configuration and bandwidth.
2. Specific to sound engine bandwidth.
3. Determines UART and front panel type.
4. Specific to product.
5. Specific to front panel.
6. Determines initial settings after hard reset.
7. Contains product and version ID numbers.

```
/*
 *    Types.h           Tuesday, February 23, 1988 7:04 PM
 */


/*
 * object type numbers
 */
enum {
      blockType=64,     /*  64  $40   - ignore */
      indexType,        /*  65  $41   - internal  index */
      tableType,        /*  66  $42   - Master Table */
      shapeType=68,     /*  68  $44   - LFO Shape Table (named) */
      soundType,        /*  69  $45   - Sound Block (named) */
      keymapType,       /*  7a  $46   - Keyboard Map (named) */
      mlistType,        /*  71  $47   - MIDI Program List (named) */
      menuType,         /*  72  $48   - Edit Menu */
      mngType,          /*  73  $49   - Edit Menu Group */
      melType,          /*  74  $4A   - Edit Menu Element */
      itblType,         /*  75  $4B   - Intonation Table (named) */
      fxType,           /*  76  $4C   - Compiled Effects */
      vmapType,         /*  77  $4D   - (Rcv) Velocity Map (named) */
      pmapType,         /*  78  $4E   - Rcv Pressure Map (named), V5 */
      editType=79,      /*  79  $4F   - Editor */
      progType,         /*  80  $50   - Program Data (named) */
      layerType,        /*  81  $51   - Layer Data (named) */
      asrType,          /*  82  $52   - ASR Parameter Block */
      lfoType,          /*  83  $53   - LFO Parameter Block */
      envType,          /*  84  $54   - ENV Parameter Block */
      efxType,          /*  85  $55   - EFX Parameter Block */
      invType,          /*  86  $56   - INV/NEG Parameter Block */
      mxrType,          /*  87  $57   - MXR Parameter Block */
      songType=91,      /*  91  $5B   - Demo Song (named), V5 */
      plistType=94,     /*  94  $5E   - Program List, V5 */
      bmapType,         /*  95  $5F   - Bin Map, V5 */
      lastType          /*  keep last! */
};

#define baseType  blockType

#define globidno  8           /* base id # for globals */
```

```
/*
 *    Objects.h               Tuesday, February 23, 1988 7:35 PM
 */


/*
 * Database definitions for K1000 Database Objects
 * To insure future compatability, all RFU items must be zero!
 */

typedef char            sByte;      /* signed byte */
typedef unsigned char   uByte;      /* unsigned byte */
typedef int             sWord;      /* signed word   (a 'short') */
typedef unsigned int    uWord;      /* unsigned word (a 'short') */

#define     nDYNAM      8           /* # dynamic range marks */

/*
 * generalized data block header
 */
typedef struct {
      uByte gdb_type;
      uByte gdb_idno;
      sWord gdb_size;
} GDB;

/*
 * extended data block header
 */
typedef struct {
      uByte xdb_type;
      uByte xdb_idno;
      sWord xdb_rfu;
      long  xdb_size;
} XDB;

/*
 * Master data block
 */
typedef struct {
      uByte mdb_type;               /* tabletype */
      uByte mdb_idno;               /* 16 */
      sWord mdb_size;
      uByte mdb_mode;               /* MIDI mode */
      uByte mdb_chan;               /* basic MIDI channel # 1...16) */
      uByte mdb_devI0;              /* sys-ex device ID 0...126 */
      uByte mdb_dchan;              /* displayed channel # 1...16 */
      uByte mdb_velMap;             /* ID of vmapType object */
      uByte mdb_ctlMap;
      uByte mdb_flags;
      uByte mdb_bflags;
      sByte mdb_tune;               /* master tune (± cents) */
      sByte mdb_trans;              /* master transpose (± semi-tones) */
      uByte mdb_intTab;             /* ID of itabType object */
      uByte mdb_refKey;             /* intonation reference key */
      uByte mdb_txPList;            /* transmit program change map */
      uByte mdb_rxPList;            /* receive program change map */
      uByte mdb_bbPList;            /* bin bank program map (K1000 only) */
      uByte mdb_bbentry;            /* current entry */
      uByte mdb_editPos[4];
      sByte mdb_pwRange;            /* global pitch wheel range (± QT) */
      sByte mdb_spRange;            /* global soft pedal range (± dB) */
      sByte mdb_dynam;              /* additive dynamic range adjust (±dB) */
      uByte mdb_rfu1;
      uByte mdb_version[4];         /* software version stamp */
      uByte mdb_cprogs[16];         /* programs/channel */
```

```
	/*
	 * local keyboard control assignments (K1000 only)
	 */
	uByte mdb_pedal1;		/* sustain pedal */
	uByte mdb_pedal2;		/* other pedal    */
	uByte mdb_wheelUp;		/* mod wheel - up */
	uByte mdb_wheelDn;		/* mod wheel -    down */
	uByte mdb_slider;		/* data slider */
	uByte mdb_rfu2;
	uByte mdb_rfu3[10];
	/*
	 * more per channel stuff
	 */
	uByte mdb_cflags[16];		/* flags */
	sByte mdb_volume[16];		/* volume adjust (±dB) */
	sByte mdb_stereo[16];		/* stereo position */
	uByte mdb_plimit(16];		/* poly limit */
	uByte mdb_range[16][2];		/* low/hi MIDI keys */
} MDB;

/* Flag definitions for mdb_flags */
#define mdb_monoOut	0x01		/* force monophonic output */
#define mdb_ignNOff	0x02		/* ignore MIDI all notes off */
#define mdb_mRefKey	0x04		/* MIDI intonation ref key */
#define mdb_parEdit	0x08		/* MIDI parameter editing */
#define mdb_confirm	0x10		/* ~confirm deletes */
#define mdb_echoPChg	0x20		/* echo program changes */
#define mdb_stealSame	0x40		/* steal same note */

/*
 * Flag definitions for mdb_bflags
 */
#define mdb_btnRept	0x01		/* button repeats */
#define mdb_btnRate	0x02		/* button repeat rate (slow/fast) */
#define mdb_btnAccl	0x04		/* button acceleration */

/*
 * Flag definitions for mdb_cflags
 */
#define mdb_volDis	0x01		/* volume disabled */
#define mdb_panOver	0x02		/* stereo pan override */
#define mdb_rngOver	0x04		/* MIDI kbd range override

/*
 * program data block
 * just a shell to enclose the layer definitions
 */
typedef struct {
	uByte pdb_type;
	uByte pdb_idno;
	sWord pdb_size;
	uByte pdb_rfu1;
	uByte pdb_midiProg;		/* output program */
	uByte pdb_stealOpt;		/* assignment algorithm */
	uByte pdb_phLimit;		/* polyphonicity limit */
	uByte pdb_rfu3[8];
} PDB;

/*
 * layer data block (named)
 * contained in program data
 */
typedef struct {
	uByte ldb_type;			/* = layerType */
	uByte ldb_idno;
	sWord ldb_size;
	uByte ldb_keymap;		/* ID of keymapType object */
	uByte ldb_lokey;		/* lowest MIDI key # */
```

```
      uByte ldb_hikey;              /* highest MIDI key # */
      sByte ldb_trans;              /* transpose (± semi-tones) */
      sByte ldb_dtune;              /* detune (± cents) */ */
      uByte ldb_delay;              /* delay (enum table 23) */
      sByte ldb_volume;             /* volume adjust (±dB) */
      sByte ldb_stereo;             /* stereo position */
      uByte ldb_effect;             /* compiled effect ID */
      uByte ldb_nLink;              /* compiled effect linkage */
      uByte ldb_kflags;             /* key state flags */
      uByte ldb_xflags;             /* effects flags */
      uByte ldb_enable;             /* enable switch (enum table #1) */
      uByte ldb_legato;             /* alt attack switch (enum table #1) */
      uByte ldb_vflags;             /* velocity trigger stuff */
      uByte ldb_dynam;              /* dynamic range (dB) */
      sByte ldb_keyTilt;            /* amplitude tilt (±dB) */
      uByte ldb_balCtl;             /* balance control (enum table #1) */
      sByte ldb_spRange;            /* soft pedal range (±dB) */
      sByte ldb_pwRange;            /* pitch wheel range (± quarter-tones) */
      uByte ldb_lastOut;            /* used by compiled effects */
      uByte ldb_rfu[7];
} LDB;

/*
 * ldb_kflags
 */
#define ldb_ignRels     0x01        /* ignore key release */
#define ldb_ignSust     0x02        /* ignore sustain pedal */
#define ldb_ignSost     0x04        /* ignore sostenuto pedal */
#define ldb_ignSusp     0x08        /* ignore suspend pedal */

/*
 * ldb_xflags
 */
#define ldb_pwlDis      0x01        /* pitch wheel disabled */
#define ldb_pwlKey      0x02        /* pitch bend only key down */
#define ldb_volDis      0x08        /* volume control disabled */
#define ldb_touchDis    0x10        /* touch sense disabled */ */
#define ldb_balRev      0x20        /* balance control reversed */

/*
 * ldb_vflags
 */
#define ldb_vt1Level    0x07        /* vtrig 1 level (fff -> ppp) */
#define ldb_vt1Sense    0x08        /* vtrig 1 sense */
#define ldb_vt2Level    0x70        /* vtrig 2 level */
#define ldb_vt2Sense    0x80        /* vtrig 2 sense */

/*
 * Amplitude control patch block
 */
typedef struct {
      uByte type, idno;             /* = efxtype,1 */
      uByte rfu1[2];
      uByte amInput;                /* mod source (enum table #1) */
      uByte amDepth;                /* mod depth (enum table #1) */
      sByte amMin ,                 /* min depth (dB) */
      sByte amMax;                  /* max depth (dB) */
      uByte rfu2[8];
} AFXB;

/*
 * Pitch control patch block
 */
typedef struct PFXB {
      uByte type, idno;             /* = efxType,2 */
      uByte rfu1[2];
      uByte fmInput;                /* mod source (enum table #1 */
      uByte fmDepth;                /* mod depth (enum table @1) */
```

```
      sByte fmMin;                  /* min depth (cents) */
      sByte fmMax;                  /* max depth (cents) */
      uByte rfu3;
      uByte dtCtl;                  /* detune control (enum table #1 */
      sByte dtMin,                  /* min value (cents) */
      sByte dtMax;                  /* max value (cents) */
      uByte rfu2[4];
} PFXB;

/*
 * Envelope control patch block
 */
typedef struct ENCB {
      uByte type, idno;             /* = efxType,3 */
      uByte rfu1[2];
      struct {
            uByte rfu;
            uByte rateCtl;          /* control (enum table #1) */
            uByte minScale          /* min rate (enum table #5) */
            uByte maxScale;         /* max scale (enum table #5) */
      }
            atCtl,                  /* attack rate */
            dtCtl,                  /* decay rate */
            rtCtl;                  /* release rate */
} ENCB;

/*
 * ASR parameter block
 */
typedef struct ASRB {
      uByte asrb_type;              /* = asrType */
      uByte asrb_idno;
      uByte asrb_flags;
      uByte asrb_trig;              /* trigger input (enum table #1) */
      uByte asrb_dtime;             /* delay time (enum table #2) */
      uByte asrb_atime;             /* attack time (enum table #2) */
      uByte asrb_stime;             /* sustain time (unused) */
      uByte asrb_rtime;             /* release time (enum table #2) */
} ASRB;

/*
 * asrb_flags
 */
#define asrb_hold       0x01        /* hold while trigger on */
#define asrb_rept       0x02        /* repeat while trigger on */
#define asrb_effect     0x80

/*
 * LFO parameter block
 */
typedef struct {
      uByte lfob_type;              /* = lfoType */
      uByte lfob_idno;
      uByte lfob_rfu;
      uByte lfob_flags;
      uByte lfob_shape;             /* ID of shapeType object */
      uByte lfob_rtCtl;             /* rate control (enum table #1) */
      uByte lfob_rtMin;             /* min rate (enum table #4) */
      uByte lfob_rtMax;             /* max rate (enum table #4) */
} LFOB;

/*
 * lfob_flags
 */
#define lfob_phase      0x03        /* initial phase (0, 90, 180, 270) */
#define lfob_effect     0x80
```

```
/*
 * envelope parameter block
 */
typedef struct {
      uByte envb_type;              /* = envType */
      uByte envb_idno;
      sWord envb_size;
      uByte envb_naSegs;            /* # attack segments */
      uByte envb_nrSegs;            /* # release segments */
      uByte envb_rfu[2];
} ENVB;

/*
 * envelope segments immediately follow the header
 *
 *    for all segments but last
 *          level and time are enumerated
 *
 *    last segment of each section is special:
 *
 *    if (level > 0)
 *          jump back and repeat going forward
 *          level is segment # to jump to
 *          time is repeat count (0 means infinite)
 *
 *    else if (level < 0)
 *          loop back and repeat (ie, backward, then forward)
 *          -level is segment # to stop at and change direction
 *          time is repeat count (0 means infinite)
 *
 *    else (level == 0)
 *          if (time > 0)
 *                time is decay/release time
 *          else if (attack section)
 *                infinite sustain
 *          else
 *                instant release
 *
 */


typedef struct {
      sByte eseg_level;             /* level (enum table #28 or #29) */
      uByte eseg_time;              /* time (enum table #2) */
} ESEG;

/*
 * INV/NEG data block
 */
typedef struct {
      uByte invb_type;              /* = invType */
      uByte invb_idno;
      uByte invb_rfu[2];
      uByte inub_inv1in;            /* input (enum table #1) */
      uByte invb_inv2in;
      uByte invb_neg1in;
      uByte invb_neg2in;
} IHVB,

/*
 * MXR data block
 */
typedef struct {
      uByte mxrb_type;              /* = mxrType */
      uByte mxrb_ldno;
      uByte mxrb_rfu[2];
      uByte mxrb_in1A;              /* input (enum table #1) */
      uByte mxrb_in1B;
```

```
      uByte mxrb_in2A;
      uByte mxrb_in2B;
} MXRB;

/*
 * keymap (named)
 * converts key number and velocity to sound file, pitch, and amplitude
 */
typedef struct {
      uByte kmap_type;              /* = keymapType */
      uByte kmap_idno;
      sWord kmap_size;
      sWord kmap_loKey;             /* low MIDI key # */
      sWord kmap_nKeys;             /* # keys - 1 */
      sWord kmap_keyOff;            /* offset to key data */
      sWord kmap_nOctv;             /* notes/octove */
      sWord kmap_pitch;             /* pitch of lowest key */
      sWord kmap_cents;             /* cents/key */
      uByte kmap_flags;
      uByte kmap_sound;             /* ID of soundType object */
      uByte kmap_rfu1;
      uByte kmap_level;             /* # timbre levels */
      uByte kmap_timbre[nDYNAM];    /* timbre level map */
      uByte kmap_rfu2[4];
} KMAP;

/*
 * kmap_flags
 */
#define kmap_atten      0x01        /* 1 means separate atten/key */
#define kmap_tune       0x0C        /* tuning type */

/*
 * the keymap data consists of byte/key arrays
 *
 * switch (kmap_flags & kmap_tune)
 *   case 0x00:   no tuning adjust
 *   case 0x04:   relative byte (LSB only)
 *   case 0x08:   relative word (MSB/LSB)
 *   case 0x0C:   absolute word (MSB/LSB)
 *
 * for (each timbre level) {
 *    if (kmap_flags & kmap_atten)
 *          amplitude adjust (1 byte/key) (8 * dB / 6)
 *    if (kmap_sound == 0)
 *          sound block ID (1 byte/key)
 *    sound file header ID (1 byte/key)
 */

/*
 * sound block (named)
 */
typedef struct sblk {
      uByte sblk_type;              /* = soundType */
      uByte sblk_idno;
      sWord sblk_size;
      sWord sblk_hBase;             /* base sound file ID # */
      sWord sblk_nHdrs;             /* # sound file headers - 1 */
      sWord sblk_hdrOff;            /* offset to 1st header */
      uByte sblk_rfu[6];
} SBLK;

/*
 * block header is followed by NUL-terminated name
 * then by array of sound file headers
 */
typedef struct sfh {
      uByte sfh_rootKey;            /* root MIDI key # */
```

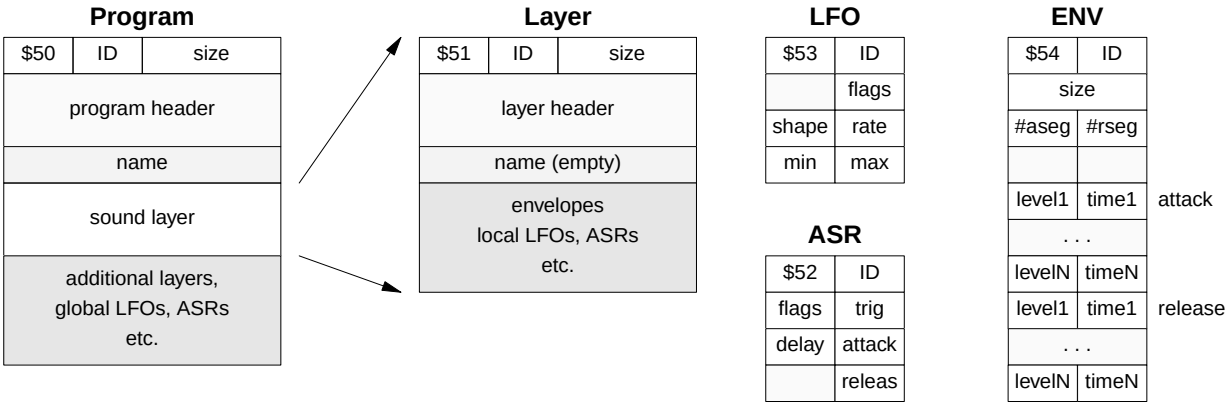
```
      uByte sfh_flags;
      sWord sfh_atten;              /* amp adjust (dons) */
      sWord sfh_pitch;              /* pitch at highest play rate */
      uByte sfh_rfu1[2];
      long  sfh_start1;             /* normal start sample address */
      long  sfh_start2;             /* alternate start sample address */
      long  sfh_loop;               /* loop point sample address */
      long  sfh_end;                /* last sample address + 2 */
      sWord sfh_env1;               /* offset to normal envelope */
      sWord sfh_env2;               /* offset to alternate envelope */
      uByte sfh_rfu2[4];
} SFH;

/*
 * header array is followed by natural envelopes
 * each segment is two words:
 *
 *    delta-a,delta-s
 *
 * where delta-s is the segment length in samples
 * and delta-a is the attack/decay rate computed as
 *
 *    delta-a = (2048 * delta-dB) / (6 * delta-s)
 *
 * if the delta-a of the 1st segment is > 0,
 * a starting amplitude of 0 is assumed
 * otherwise, the amplitude starts at the maximum value
 *
 * the last two segments are the decay and release segments
 * the delta-s for these must be 0
 * the delta-a should be computed using a delta-s which corresponds
 * to the # of samples in 10 msec at the highest playback rate
 * for the decay segment, delta-a may be zero (for infinite sustain)
 */



/*
 * LFO shape table (named)
 *
 * header followed by name, then waveshape (256 words)
 * signed indexing is used so the offset is to the 0th entry
 * (ie, the middle of the table)
 */
typedef struct {
      char  shapeTYPE;              /* shapeType */
      char  shapeID;                /* ID # */
      sWord shapeSize;              /* size (bytes) */
      sWord shapeOffs;              /* offset to center point (0th entry) */
} SHAPE;



/*
 * MIDI List (named)
 * used to map MIDI Program change numbers to real program numbers
 */
typedef struct {
      uByte mlistTYPE;
      uByte mlistID;
      sWord mlistSize;
      sWord mlistBase;              /* base MIDI program *  */
      sWord mlistN;                 /* # table entries - 1  */
      sWord mlistOff;               /* offset to 1st entry (byte) */
} MLIST;
```

```
/*
 * Intonation table (named)
 */
typedef struct {
      uByte itblTYPE;
      uByte itblID;
      sWord itblSize;
      sWord itblTbl[12];            /* [0] always 0 */
                                    /* entries in cents rel to equal temp */
} ITBL;


/*
 * Velocity map (named)
 */
typedef struct {
      uByte vmapTYPE;
      uByte vmapID;
      sWord vmapSize;
      uByte vmapVels[nDYNAM];       /* fff -> ppp */
} VMAP;


/*
 * Data structure definitions for 1200 Demo objects
 * (from the V5 manual addendum)
 */

/*
 * file type, first longword in file
 */
#define SROM            0x53524F4D

/*
 * Demo object type and id declarations
 */
#define Demo1200        0x5B
#define DemoIDBase      1

/*
 * object file header, one per file
 */
typedef struct {
      long DO_type;                 /* Demo (SROM) */
      word DO_rfu[14];              /* must be 0 */
      long DO_blockSize;            /* negative block size */
} dFHdr;

/*
 * song header, one per song
 */
typedef struct {
      char DS_objType;              /* 0x5B is a Demo song */
      char DS_objID;                /* is (1..255) */
      word DS_objSize;              /* size of object */
      long DS_tempo;                /* actual initial beat increment */
} demoSong;

/* name of the object goes here, before the song itself */
/* (must be null terminated, even number of bytes) */
```

```
/*
 *    Tables.h                Tuesday, February 23, 1988 7:15 PM
 */


/*
 * standard table header
 */
typedef struct TBLHDR {
      uByte tbl_type;         /* = tableType */
      uByte tbl_idno;
      sWord tbl_size;
} TBLHDR;


/*
 * time enumeration entry
 */
struct enumTime {
      uWord ticks;            /* msec / 10 */
      uWord delta;            /* 32767 / ticks */
      uWord slop;             /* 32767 % ticks */
      uWord rfu;
};

/*
 * lfo rate enumeration
 */
struct enumRate {
      uWord roffs;            /* offset into rate table entry */
      uWord hertz;            /* hertz * 100 */
};

/*
 * control source enumeration
 */
struct enumCtls {
      uByte type;
      uByte idno;
      sWord offs;             /* offset to name string
};

/*
 * envelope scale factor table header
*/
struct envScaleHdr {
      sWord base;             /* offset to 0th entry */
      sWord low, high;        /* valid index range */
      sWord dispNom;          /* display value denominator */
}

/*
 * table entry
 */
struct envScaleEnt {
      sWord multOff;          /* offset into log multiplier table */
      sWord dispNum;          /* display value numerator */
}
```

```
/*
 * table IDs
 */
enum {
      enumCtlsID=1,     /*   1  - control source enum */
      enumTimeID,       /*   2  - ENV/ASR time enum */
      enumDelayID,      /*   3  - layer delay enum */
      enumRateID,       /*   4  - LFO rate enum */
      envScaleID,       /*   5  - EMU rate scale factor anum */
      logScaleID,       /*   6  - log multiplier table */
      prcTableID,       /*   7  - playback rate compensation */
      pidTableID,       /*   8  - parameter ID table */
      mnpTableID,       /*   9  - menu position table ID */
      defTableID,       /*  10  - default object table */
      ctlMapID,         /*  11  - MIDI control mapping table */
      btnTableID,       /*  12  - front panel button table */
      diagTblID,        /*  13  - diagnostic table */
      productID,        /*  14  - product  ID/configuration */
      arcTableID,       /*  15  - Arnold configuration */
      masterID,         /*  16  - initial master parameters */
      omtID,            /*  17  - object mapping table (program only) */
      pseudoID,         /*  18  - Pseudo parameters (program only) */
      attenMapID=21,    /*  21  - velocity -> attenuation table */
      velocMapID,       /*  22  - velocity -> control value table */
      pressMapID,       /*  23  - pressure -> control value table */
      volTableID,       /*  24  - volume control table */
      balTableID,       /*  25  - balance control table */
      playRateID,       /*  26  - playback rate table */
      lfoDeltaID,       /*  27  - LFO phase increment table */
      envValueID,       /*  28  - bi-polar EMU level enumeration */
      ampValueID,       /*  29  - amplitude EMU level enumeration */
      atkTableID,       /*  30  - amplitude EMU attack table */
      typeTableID
};
```

1000 Series Program/Layer Structures



Program Header

	0	1	2	3
0	$50	ID #	size (bytes)	
4		output prog #	stealing option	poly limit
8				
12				

Layer Header

	0	1	2	3
0	$51	ID #	size (bytes)	
4	keymap ID #	low key #	high key #	transpose
8	detune	delay	volume	stereo
12	compiled EFX ID#	compiled EFX link	k-flags	x-flags
16	enable switch	alt start switch	v-flags	dynamic range
20	keyboard tilt	balance control	soft pedal range	p-wheel range
24	compiled EFX			

Layer Flags

k-flags				ignore suspend	ignore sostenuto	ignore sustain	ignore release
x-flags		balance sense	touch disable	volume disable		p-wheel key only	p-wheel disable
v-flags	v-trig #2 sense	velocity trigger #2 level		v-trig #1 sense	velocity trigger #1 level		

1000 Series Master Parameter Block

	0	1	2	3	4	5	6	7
0	$42	16	size		MIDI Mode	Channel	SysEx ID	Display Channel
8	velocity map ID		flags	bflags	master tuning	master transpose	intonation table ID	intonation key number
16	transmit prog map	receive prog map	bin bank prog list	bin bank selection				
24	p-wheel range	soft-pdl range	dynamic range adj		software version stamp			
32	program		program assignment per channel					
48	pedal 1 assign	pedal 2 assign	wheel up assign	wheel dn assign	slider assign			
64	flags		miscellaneous flags per channel					
80	+/-dB		volume adjust per channel					
96	+/-9		stereo position per channel					
112	off,1..N		polyphonicity limit per channel					
128	low	high	MIDI key number range per channel					

1000 Series LFO Tables

LFO Rate Enumeration

	type=$42	ID=4
	size (bytes)	
0	offset to phase incr	
	hertz * 100	
	. . .	
255	offset to phase incr	
	hertz * 100	

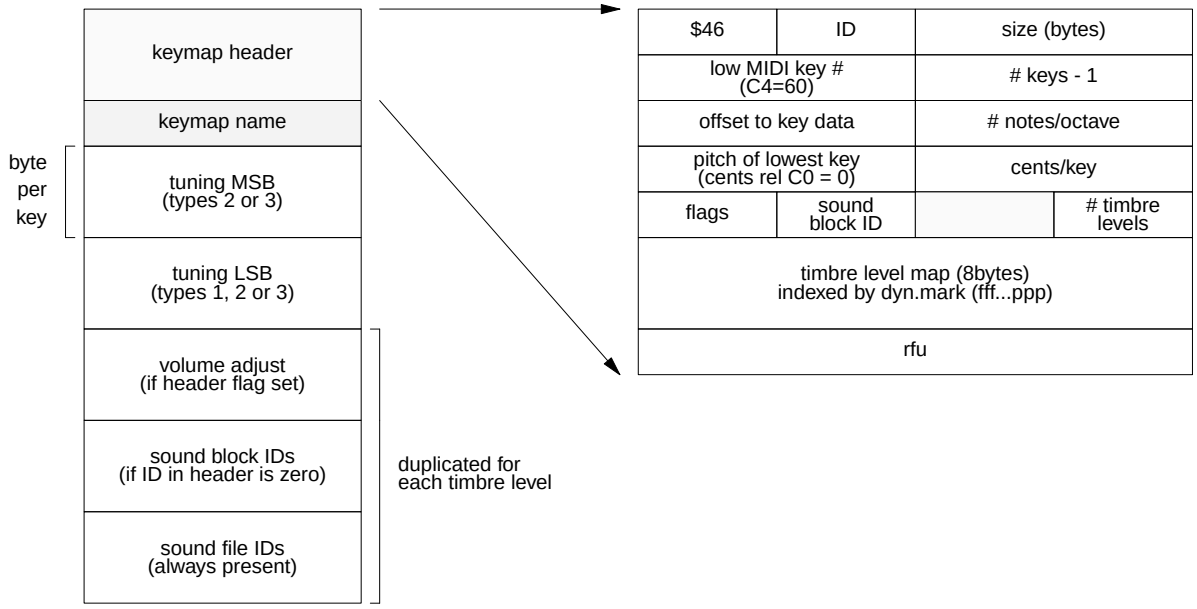
LFO Phase Increment

	type=$42	ID=27
	size (bytes)	
	phase incr[0]	
	non-linear table of phase increments	
	phase incr[2047]	

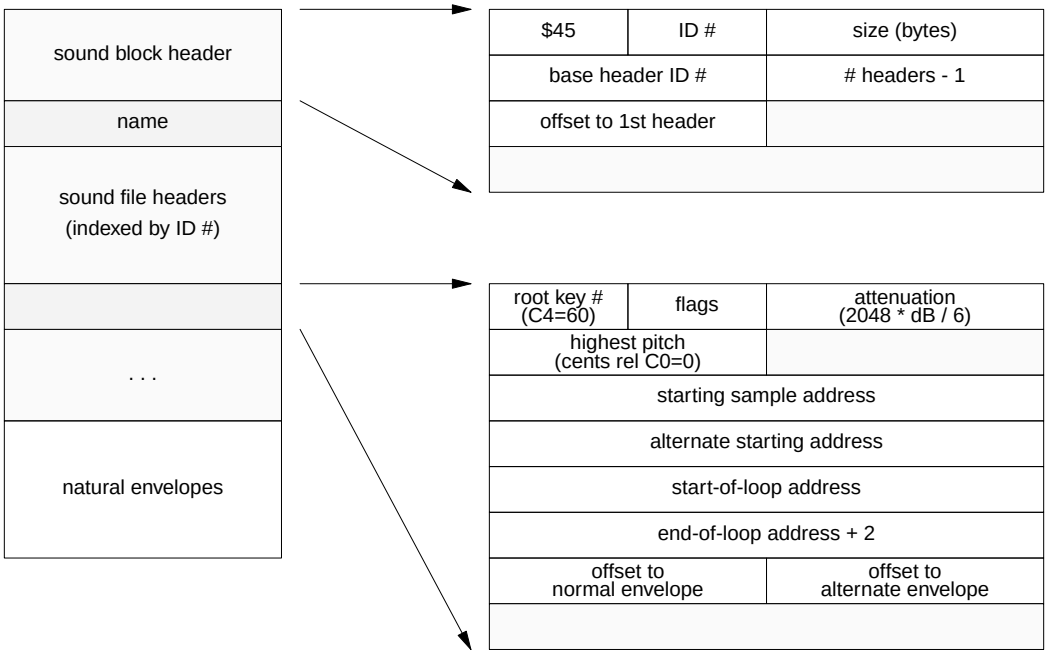
LFO Shape Table

	type=$44	ID#
	size (bytes)	
	offset to zeroth entry	
	name (nul-terminated)	
	value[-128]	
	. . .	
	value[0]	
	. . .	
	value[127]	

1000 Series Keymap Structure



1000 Series Sound Block Structure



1000 Series Velocity Mapping