SoundWorld

Overview

SoundWorld is a standard for naming and classifying sound timbres used by music programs, to replace the patchwork of numbering systems used by MIDI keyboards, sound modules and sound libraries. It aims to be to musical sounds what Unicode is to text characters.

A program which is SoundWorld-compatible will be able to list the sounds available to it using standardized, user-friendly names in hierarchical menus; it will be able to play music files which use these names, even if the files were created for different playback devices; it will intelligently substitute similar sounds for ones which are not available on the current computer; and it will be able to do all of this for sounds which it has never encountered before just as well as for existing ones.

Sound IDs

SoundWorld refers to sounds using *sound IDs*. A sound ID, or ID for short, is a structured name describing a sound timbre, such as **woodwind.flute** or **woodwind.flute.piccolo**. They are shown here in bold. Although IDs are often the names of instruments, this is incidental: they refer to sound timbres themselves, no matter how the sound is generated. So **woodwind.flute** refers to a flute-like sound whether it is played live, sampled, or synthesized.

A set of these IDs is called a *sound world*, meaning a complete repertoire of named sounds that can be referred to by a music program. The sounds in a sound world do not have to be installed on a particular computer: they are all the sounds that *could* be played. It is often useful for a user to be able to see a complete list of sounds, even if only some are currently playable.

A sound world is only usable if each ID can be mapped to its corresponding sound timbre (or *sound* for short). The sound is normally obvious from the ID's name, but to be used by software it has to be described digitally, e.g. in terms of the MIDI commands that will play the sound on a particular MIDI device.

S3W

In the same way that the MIDI standard does not mandate that everybody uses a particular set of program numbers to specify sounds, the SoundWorld standard does not mandate a particular sound world. But it does *encourage* programs to use a specific sound world called the Sibelius Standard Sound World (S3W), though others will sometimes be needed.

S3W makes music programs and hardware compatible in the same way that General MIDI does, but it is much more scalable: S3W lists most sounds currently available in music hardware and software, especially acoustic instruments; it is extensible by manufacturers and users who create new sounds; and programs encountering new sounds can deal with them in the same way as existing sounds. The rest of this section describes SoundWorld in general rather than the specifics of S3W.

An example: playing a note

When a SoundWorld-compatible music program plays a note, it needs the following information:

- The ID of the required sound (whether chosen by a user, or specified in a music file);
- The complete sound world that this ID comes from;
- The commands needed to play IDs on the required playback device, e.g. MIDI bank and program numbers;
- Optional parameters such as pitch, duration, volume, etc. required by this specific note. Some sounds, such as drum sounds, may not require a pitch or even a duration.

With this information, the program has to do the following:

- It must check whether the sound is available on the current computer. It may be nominally available, but not for the pitch (or other parameters) that this note requires; if the program wants to check this level of detail, it will need information about the capabilities of the appropriate playback device (which is not specified in SoundWorld). Another reason the sound may not be available is if the device limits the number of channels or slots available to play different timbres.
- If the sound is unavailable, a substitute sound has to be found. Programs can use the *basic substitution rule* to do this, described below, or a more sophisticated rule of their own. (If channels or slots are in limited supply, a variant on the basic substitution rule can be used to decide which sound already in use is closest to the sound required.)
- The program sends the sound commands to the playback device, and the desired note (or one similar to it) is played.

An example sound world

Typically a sound world is specified in a file containing an ordered list of IDs (though SoundWorld does not mandate any specific file formats). Music programs may be hard-wired to use a particular sound world which has known properties, or they may able to use any sound world file.

A simple sound world might contain the following IDs, in this order:

woodwind.flutes.flute

woodwind.flutes.flute.ensemble.2 players

woodwind.flutes.flute.ensemble.3 players

woodwind.flutes.flute.ensemble.flutter-tongue

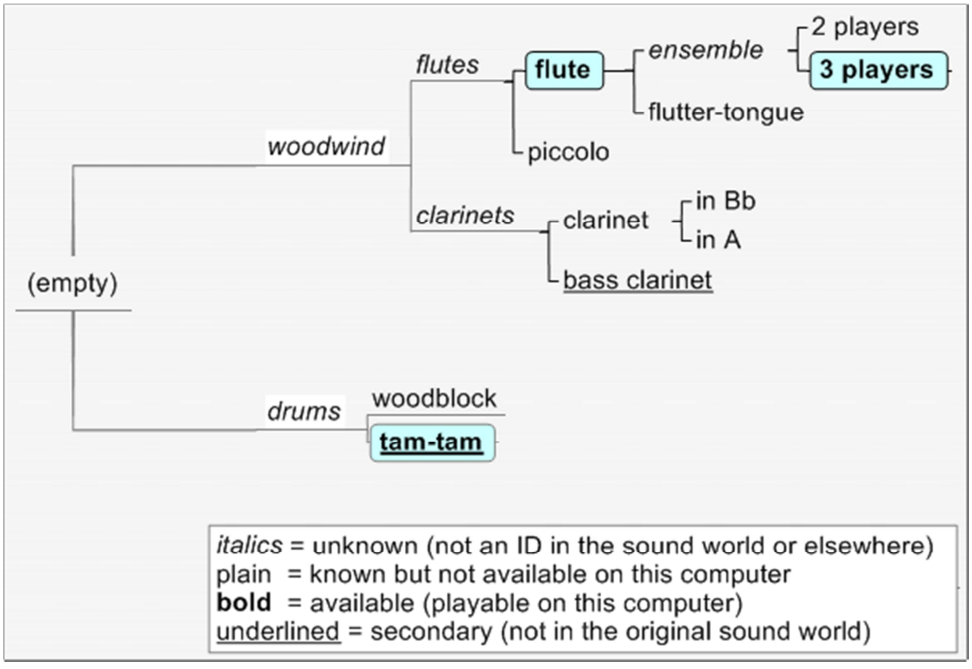
woodwind.flutes.piccolo

woodwind.clarinets.clarinet.in Bb

woodwind.clarinets.clarinet.in A

drums.woodblock

This sound world can also be displayed as a tree:



Note that the order of the IDs is preserved, which is essential whether they are stored in a list or tree format. The order is needed when finding the best substitute for a sound which is unavailable, and it's also useful when displaying the IDs in menus.

Primary and secondary IDs

Actually the tree shown here contains two extra IDs that are not listed in the sound world: **drums.tam-tam** and **woodwind.clarinets.bass clarinet**. This tree is a snapshot of all *known* IDs on a particular computer – meaning all those that a music program has come across. A program starts with *primary* IDs, the ones in the sound world file, but can supplement this with *secondary* IDs, which typically come from two sources:

- Sound sets, which are files listing the IDs provided by a playback device;
- Music files opened in the program which have been created on a different computer.

Here, **drums.tam-tam** came from a sound set: we know this because the tree shows that it's available to be played on the current computer, so it must be in the sound set of a playback device. But **woodwind.clarinets.bass clarinet** came from a music file: we know this because it's shown as unavailable, so it can't be in a device's sound set.

The fact that an ID is secondary makes no difference to its behaviour. For instance a sound world may include **keyboard.piano.grand**, but a specialized piano sound module may provide a secondary ID called **keyboard.piano.grand.steinway.model d**, created after the sound world was defined. Any music program that needs this sound will come across its ID in the sound module's sound set, or in a music file created on another computer using this sound set, and be able to add it temporarily to the sound world. The ID can then be displayed in menus, can be chosen by users, and so on.

Known and unknown IDs

IDs can be divided into three types: unknown, primary and secondary. A known ID just means a primary or secondary ID. An unknown ID such as **woodwind.flute.ocarina** could actually be known elsewhere – perhaps it's in a sound library – but has never been encountered on the current computer; so an unknown ID can become a secondary ID by being encountered.

Some nodes in the tree actually correspond to unknown IDs, because they aren't listed in the sound world in their own right. For instance **woodwind** and **woodwind.flutes** are unknown and shown in italics, because they're only the start of a known ID. Leaf nodes are always known, and some non-leaf nodes such as **woodwind.flutes.flute** can be known.

Whether or not a node in the tree is known, i.e. is listed in the sound world as a separate ID, makes no functional difference at all; it only makes a difference to a program's user interface. The main reason not to list a node as a separate ID is because that ID would not be very useful: for instance **woodwind** does not evoke a definite sound that users would want to choose, so it would just clutter up any menu of sounds. But in other respects unknown IDs can be used just like known ones: if a program attempts to play a note using **woodwind**, and it finds this sound is not available (which will normally be the case), the basic substitution rule will simply fall back to a related sound that is available such as **woodwind.flutes.flute**.

Sound sets

To find which IDs are available on the current computer, a typical mechanism (though not the only one) is simply to use pre-supplied files, called sound sets, which list the IDs for a specific playback device and the commands required to play them.

For instance, the computer which generated the above tree may have used a sound set like this, using a fictitious file format:

device ‘Tinny Sound Module’

‘drums.tam-tam’, bank 1, program 0, pitch 60

‘woodwind.flutes.flute’, bank 0, program 0

‘woodwind.flutes.flute.ensemble.2 players’, bank 0, program 1

This shows where the secondary sound **drums.tam-tam** came from. The Tinny Sound Module is a MIDI device, and to play a tam-tam the program needs to issue bank 1, program 0 then pitch 60 (middle C). Each unpitched percussion instrument is listed as a separate ID, so SoundWorld needs no concept of drum sets.

The silent sound

Sound worlds will often need an ID to represent silence. This cannot be done in a standard way in different sound worlds, such as with a special ID such as **none**, because sounds are meant to be specified separately from IDs, and SoundWorld does not mandate how sounds are specified. So it is left to each sound world how to indicate that an ID maps onto silence.

One use for the silent sound is as a fallback for very individual sounds such as a tam-tam, or General MIDI’s applause, if they are unavailable and any other fallback sound would be disturbing or comical. A tree structure for achieving this can be seen in S3W.

Designing a new sound world

Designers of new sound worlds are recommended to study the design decisions made for the Sibelius Standard Sound World: see below, especially the section S3W Design FAQs.

Definition of a sound world

This is a more formal definition of a valid sound world and related terms, for people designing SoundWorld-compatible programs or new sound worlds.

1. A *sound* means a timbre. (It could be defined as a set of notes which are perceived as being produced by the same instrument, person, object etc. using the same playing technique. So the notes can vary in other parameters such as pitch and duration.)
2. A *sound ID* (or simply an *ID*) is a string consisting of 0 or more *elements*, separated by ‘.’. It should be human-readable and should describe a sound.
3. An *element* is a string of 1 or more characters (excluding ‘.’).
4. A *sound world* is an ordered tree of IDs. Each node in the tree corresponds to an element of each ID, and the tree is ordered in the sense that the sub-nodes of each node are ordered. Each ID maps onto exactly one sound: specifically, the sound described by the name of the ID. (So IDs cannot have misleading names.)
5. A sound world has a name expressed as a unique non-empty string; this name can be used in music files to show which sound world any sound IDs come from. It should be human-readable and capitalized suitable for display directly by music programs, e.g. ‘Sibelius Standard Sound World’.
6. The IDs in the sound world are called *primary*. Any other ID added to the tree later is called *secondary*.
7. Each ID can only appear once within the same sound world. Similarly each sound can only appear once within the same sound world, with the exception of the silent sound. (Silence can be considered to be an infinitely quiet version of any sound, and therefore has some similarity with any sound.)
8. A sound world must support the basic substitution rule: the rule must choose as far as possible the best substitute for every ID, whatever combination of IDs is actually available. This places several constraints on the tree: for instance, each of a sound’s descendants must be a better substitute for it than any sound in the tree which is not a descendant.

9. The root of the tree corresponds to an empty ID (containing no elements), which behaves like any other ID. So it can be listed in a sound world as a primary ID (or not), and therefore can correspond to a sound (or not). If its sound is not available, then according to the basic substitution rule *any* missing sound which falls back to the empty ID, will then fall back from there to the highest priority sound in the tree which *is* available. So the only situation in which the empty ID can fail to fall back to a sound is when no sounds at all are available.

10. No file format or data format for sound worlds is mandated. In particular, it is not mandated how the sounds are specified: they could be specified implicitly (from the name of the ID itself), verbally (if this name is not clear enough), or via the data required to play them on specific playback devices. Normally this latter technique is used not to specify the sound itself, but to specify a version of the sound on a particular playback device. This is typically done using a *sound set*, which is a file listing a device's IDs and the commands to play the corresponding sounds.

The basic substitution rule

The basic substitution rule finds the best substitute for a sound which is unavailable. As its name implies, it *doesn't have to be used* by programs if they have a more sophisticated rule instead. But other rules will often rely on specific knowledge of the sounds involved, such as a custom-made list of the best substitute for each ID. The basic rule needs no information beyond the tree structure, so it can be used for secondary sounds that have never been seen before, and even for whole sound worlds that have never been seen before.

'Musical similarity'

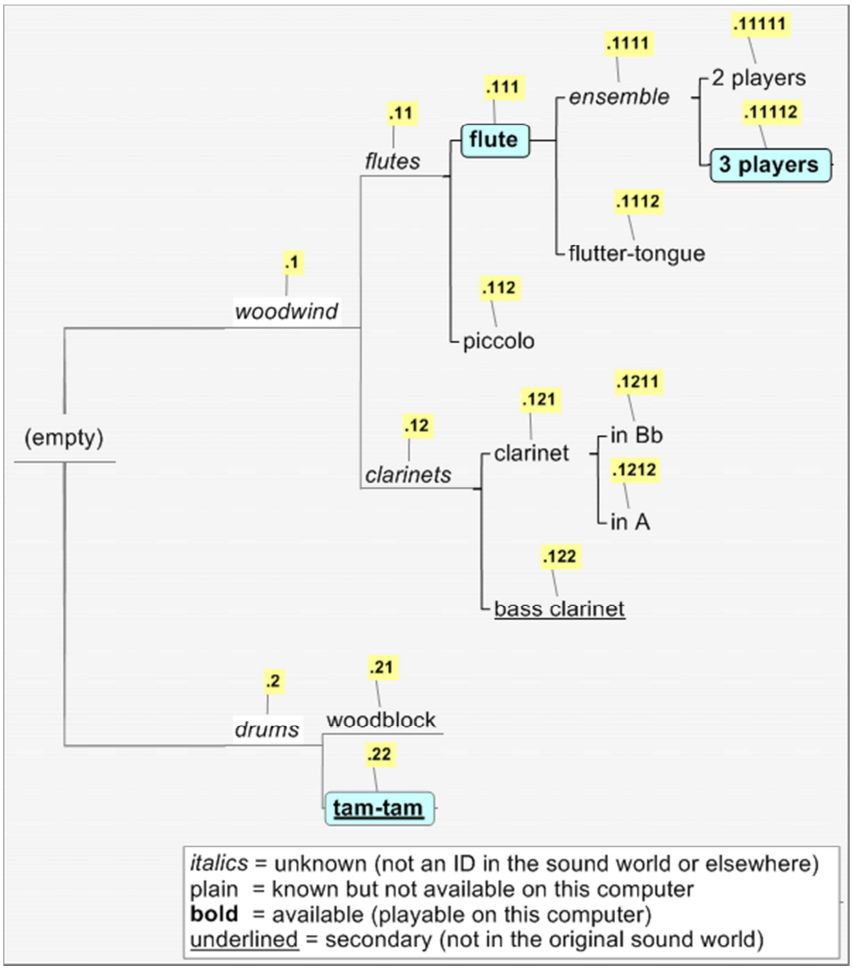
The rule maps any ID, whether known or unknown, on to the most *similar* available sound, which is the *closest* one in the tree (using a special definition of closeness).

For the rule to find good fallback sounds, the sounds in the tree should be arranged according to a special type of similarity which is not acoustic similarity, but what we might call *musical similarity*. For instance, if a violin pizzicato sound is unavailable, it is probably better for it to fall back to a violin arco sound, not to a guitar sound, even though the latter is closer in an acoustic sense. Musical similarity is a subtle combination of acoustic similarity and musical suitability, and this must be kept in mind when choosing the IDs that make up a sound world.

Algorithm to substitute a single ID

The rule uses a kind of depth-first search, which can be described informally as follows. If an ID is unavailable, the rule will always fall back to a descendant if one is available, starting with the first child (and its descendants). If none of the descendants are found to be available it will try falling back to descendants of the parent, then the grandparent, and ultimately the descendants of the root node (meaning available sounds anywhere in the whole tree).

The rule can be described more accurately by labelling each node with a decimal number between 0 and 1 (assuming, just to simplify this explanation, that nodes don't have more than 9 branches). The root is taken as 0, and each node further from the root adds a decimal digit: so we have .1 for the 1st branch, .2 for the 2nd, .12 for the 2nd subbranch of the 1st branch, and so on:



As an example, **woodwind.clarinets.clarinet** is labelled as .121 and it falls back as follows:

- Find the lowest-numbered available node in the range >= .121 but < .122 (i.e. the lowest number with the same 3rd digit). This means checking .121, .1211, then .1212. It turns out that none are available, so…
- Find the lowest-numbered available node in the range >= .12 but < .13. This means checking .12, then .121 and its descendants again (or just skipping them), then .122. It turns out that none are available, so…
- Find the lowest-numbered available node in the range >= .1 but < .2. Two are available: .111 and .11112. The lowest is .111 which is **woodwind.flutes.flute**.

So **woodwind.clarinets.clarinet** falls back to **woodwind.flutes.flute**.

Here is another example which involves the root node being reached. Pretend that **drums.tam-tam** is in fact unavailable (which it isn't):

- Find the lowest-numbered available node in the range >= .22 but < .23. None are available so…
- Find the lowest-numbered available node in the range >= .2 but < .3. None are available so…
- Find the lowest-numbered available node in the range >= 0 but < 1; in other words, in the whole tree. The lowest is .111 which is **woodwind.flutes.flute**.

This algorithm executes in time of less than O(n) for a single substitution, where n is the number of nodes. This is not very efficient when a music program will typically want to find substitutes for all the unavailable sounds, which could be many thousands. Later we give an algorithm which efficiently finds all these substitutes at once.

However, sometimes an algorithm like the above one is needed. For instance, a music program may want to use a more sophisticated rule than the basic substitution rule, to choose different substitutes depending on a note's pitch. If a sound (say, a piccolo) is nominally available, but a particular note can't be played because another parameter (say, its pitch of C5) is out of range, a program could do an 'on-the-fly' search to find the closest substitute (say, a flute sound). This search can't normally be done in advance, since the definition of 'available' is a one-off for the individual note: in this case 'available for playing C5'.

Consequences of the rule, and their justification

Any sound world by definition has to support the basic substitution rule as closely as possible. This means that its IDs cannot be arbitrary: the tree structure that they form has to organize the sounds in terms of similarity.

Here are some important consequences of the rule and why it was chosen to have them:

1. Because the rule searches the tree *below* the original sound before searching above it, **a sound's descendants must be more similar to it than any of its ancestors, siblings, etc**. This can be thought of in terms of sets: we can regard **brass.trumpet** as the set of all possible notes which sound like a trumpet. This set includes the subset **brass.trumpet.mute**, which is the set of all notes which sound like a muted trumpet. So, according to the basic substitution rule, a trumpet note must sound more like any other trumpet note than any non-trumpet note; for instance, a muted trumpet sounds more like any other trumpet than a flute.

Although this assumption is approximate, one justification for it is that in S3W it was possible to define a hierarchy of effects that can be added to IDs, like Ornaments (**trill** etc.), Macro quality (e.g. **pizzicato**), Micro quality (e.g. **mute**). In this hierarchy at least, it seems broadly true that if two IDs start with the same elements but then differ at an element which is an effect, this completely overrules any similarities in subsequent elements, which represent less important effects.

2. **When two *descendants* of a sound are available, and one is also a descendant of the other, the higher descendant (i.e. nearer the root) is preferred.** For instance, in the above example **woodwind.clarinets.clarinet** doesn't fall back to **woodwind.flutes.flute.ensemble.3 players**, because this has value .11112 which is more than the value of .111 for **woodwind.flutes.flute**. In effect, when searching for an available node inside the **woodwind** branch, it goes deeper and in increasing order via .1, then .11, then .111, at which point it stops.

One justification for this is that when looking for the fallback for a node, the rule should first check whether the node itself is available. So when looking for the fallback for **woodwind.flutes.flute**, it must first check .111 itself. Another justification is that if **woodwind.flutes.flute** is available, it is likely to be a better representative of flutey sounds than many of its descendants such as **woodwind.flutes.flute.flutter-tongue**. So, supposing that both of these two sounds are available when trying to find a fallback for **woodwind.flutes**, it's better to choose **woodwind.flutes.flute** than its descendant **woodwind.flutes.flute.flutter-tongue**.

3. The children of an unavailable node are searched in the order first to last. This means that ***all* the descendants of a node's first child must be more similar to that node than any of the node's other children**. In practice this often means that the first child contains the basic or most representative sound or sounds, and the other children contain variations (muted, accented etc.) which the basic sound is an approximation to. If a sound doesn't exist but its parent and its sibling('s descendant) do, the rule prefers the parent.

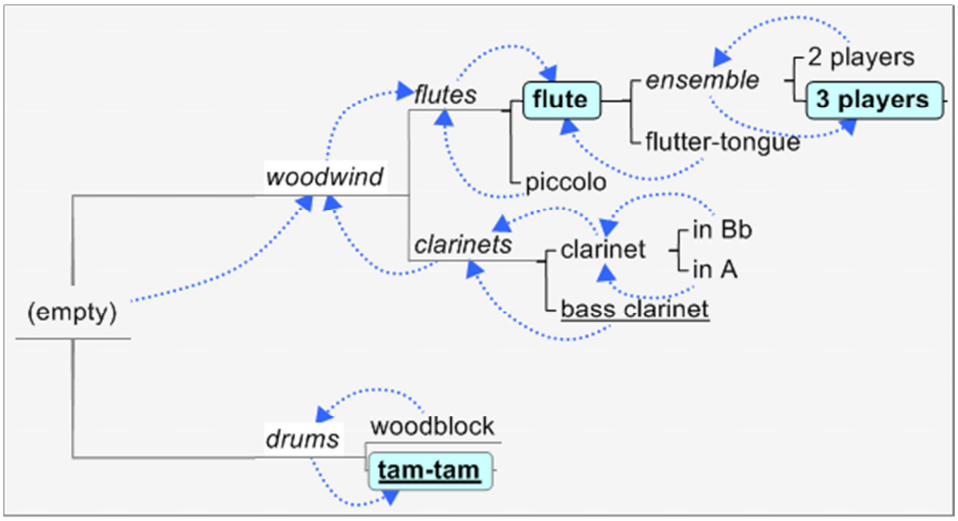
Algorithm to substitute the whole tree

The following algorithm is much more efficient than the previous one because it finds the best substitute for all the IDs in a sound world at once, taking exactly O(n) for the whole tree:

1. Construct a tree of all known IDs, making sure the order of branches is preserved.

2. For each node which has an available sound, mark it as having been *processed*, meaning it either has a sound, or an arrow leading to another node. (Programs should cope with the special case where there are no sounds available at all.)
3. Find a node which has not yet been processed, but all of whose children (if any) have been processed. This means starting with leaf nodes, which are eligible because they have no children.
4. This node's own sound is clearly not available (because it hasn't been processed in step 2), so we look for a child or parent node to fall back to. If all the children have arrows pointing to the current node, or if there are no children, draw an arrow from the current node to the parent. (If there is no parent, then this is the root node and there must be no sounds available at all, so nothing can be played anyway.)
5. Otherwise (if at least one child doesn't have an arrow pointing to the current node), draw an arrow to the *first* child which does not have an arrow pointing to the current node. This is the main reason why IDs in the tree have to be ordered.
6. Mark the current node as processed, and go to step 3.
7. When step 3 runs out of suitable nodes to process, all nodes will have been processed. The root node (the empty ID) doesn't have to be treated differently from the others: it can fall back to one of its children, e.g. piano, allowing nodes elsewhere in the tree to fall back to piano via the root.
8. Finally, we find the substituted sound for each node: if it has an available sound, then no substitution is required. Otherwise, there will be exactly one arrow pointing away from it: keep following the arrows until an available sound is reached. This process will always terminate, because there are no cycles of arrows.

Here's an example of this algorithm operating on the previous tree, when step 7 has been reached:



If we're trying to find the substitute for (say) **woodwind.clarinets.clarinet**, then according to step 7 we follow the arrows up to **woodwind** and then down to **woodwind.flutes.flute**. This is the same substitute as with the previous algorithm, but finding it is much faster because following arrows is much faster than searching the tree, once all the arrows have been found.

Sibelius Standard Sound World (S3W)

Sibelius Standard Sound World, known hereafter as S3W, is the sound world intended for normal use in most circumstances.

Features

S3W has several features over and above what sound worlds are generally required to support:

- It contains IDs for many sounds available in existing music software and hardware, especially of acoustic instruments.
- All IDs are in English, and use only Latin 1 characters (which can be stored in 1 byte).

Some features are actually ID naming conventions, which can be used by programs to analyze IDs or create new ones:

- Each ID indicates whether it is a pitched or unpitched sound. This is useful for programs which want to treat unpitched sounds specially, such as to give the user a list of drum sounds.
- Effects like mute, staccato, vibrato etc. are named in a consistent way across all the IDs, so that music programs can automatically add combinations of these effects to any ID, including new IDs.
- There are other explicit conventions for creating new IDs: partly for consistency, and partly so that programs can automatically generate IDs (such as organ registrations based on combinations of stops).

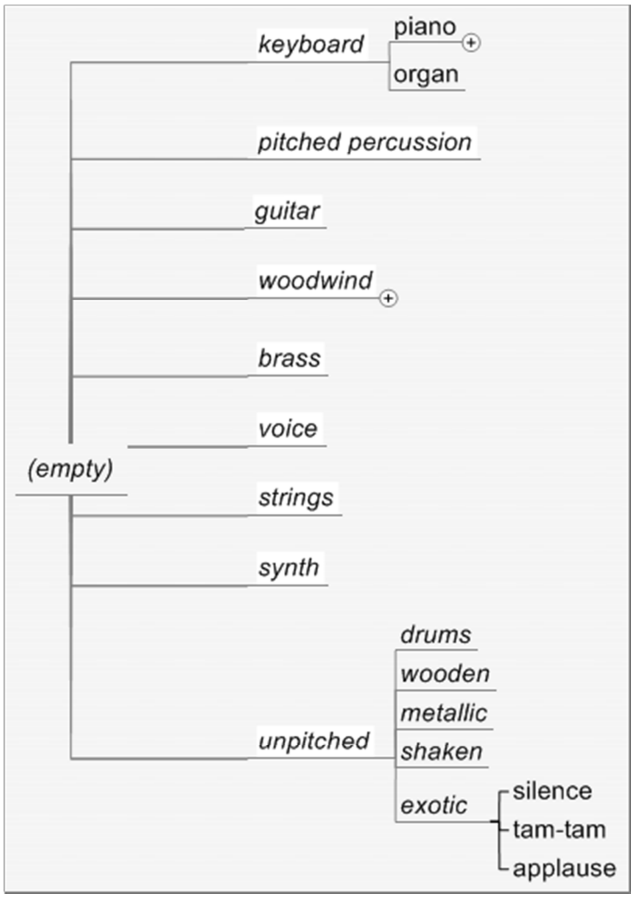
Syntax of IDs

S3W uses IDs in a single standard language, namely English. Foreign music terms used in English, such as ‘legato’, are also allowed. By using English, the character set can be conveniently limited to Latin 1, which includes accented characters needed for foreign terms like ‘pincé’. Also, each character only takes up 1 byte.

In S3W IDs are further restricted to lower case, to avoid case errors and for ease of typing (so the legal Unicode characters are 0020-0040, 005B-007E, 00A0-00BF, 00D7, 00DF-00FF). But in user interfaces such as menus, it will normally be best to display IDs in title case, with the first letter of each word capitalized. This also fixes problems with some lower case words: for instance **french horn** will be displayed as ‘French Horn’.

Tree structure

S3W starts with **keyboard.piano.grand** as the first ID, and the empty ID is unavailable. This is so that if a computer has (say) no guitar sounds available, any guitar ID will fall back to the empty ID, and thence to the grand piano if that is available instead.



Unpitched sounds

It is a rule that all unpitched sounds go into the **unpitched** branch, so that programs can easily identify them. This branch is at the bottom, so that pitched sounds will only fall back to an unpitched sound if there are absolutely no pitched sounds available.

The ID **unpitched** itself is unavailable, partly so that users can't choose it from menus, and partly because mapping it to (say) the silent sound would prevent unavailable unpitched sounds falling back to available unpitched sounds.

Silence as a fallback

unpitched.exotic contains sounds which, if unavailable, fall back to the silent sound, because they are too unusual for there to be a useful substitute. The **unpitched.exotic** branch is at the very bottom of the tree so that it is the lowest priority fallback of all.

Because **unpitched.exotic.silence** is the first sound in the **unpitched.exotic** branch, pitched and unpitched sounds will ultimately fall back to silence, instead of to the other exotic sounds. This also means that unpitched sounds will never fall back to pitched sounds.

It's convenient that the silent sound **unpitched.exotic.silence** is always available – even for a computer without a sound card. Even for a computer without speakers! So S3W-compatible programs don't have to treat as a special case situations where there are no playback devices at all: the silent sound is always available as a last-ditch fallback. (Programs may also find it useful internally to pretend there is always at least one device, a special 'Silent' device which is hidden from the user, and can only play the silent sound.)

Empty ID

The empty ID or root node is unavailable for similar reasons that the **unpitched** node is unavailable: to prevent it being listed in menus, and so that unavailable pitched sounds can fall back to available pitched sounds via the root.

Effects

There are many ID elements or combinations of elements that are used in more than one ID, so they can be treated by programs as 'well-known names'. These are called *effects*.

For instance, **mute.straight** is an effect that appears in several brass IDs. A program could speculatively add it to an ID, then check whether the resulting ID is known: if it is, it can assume this is a muted version of the original sound. It will have to try adding the effect before and after each existing element of the ID, since its position isn't always predictable.

The effects come in a number of types, found in the **SoundIDPriority.xml** file (available on request).

Number of players

Sounds are assumed to be played by 1 player by default. To indicate several players, use the **ensemble** element followed by the number of players e.g. **ensemble.2 players**.

Note that **ensemble** and **n players** are not consecutive in the hierarchy of effects, because the difference between one and many is much more important than the precise number. For instance, if the ID **brass.trumpet.ensemble.mute.2 players** isn't available, the basic substitution rule will fall back to **brass.trumpet.ensemble.mute.4 players** rather than **brass.trumpet.mute** or **brass.trumpet.ensemble.2 players**. So to stand in for 2 muted trumpets, any number (> 1) of muted trumpets is preferable to a solo one, or to 2 unmuted trumpets.

Numeric elements when adding secondary IDs

Some S3W IDs contain numbers, such as **brass.trumpet.ensemble.4 players**. Here **4 players** is called a *numeric element* because it contains a number. Typically, the numeric elements in primary IDs are just common examples, but other examples with different numbers are likely to appear in secondary IDs. So it would be no surprise to come across a sound set which contained **brass.trumpet.ensemble.2 players** or **brass.trumpet.ensemble.3 players**.

Programs which implement S3W should treat numeric elements specially when adding secondary IDs to the tree, so that they appear in correct order alongside similar elements which contain a different number. The rule is as follows:

- A *number* is a sequence of 1 or more characters **0** to **9**, where the sequence is either preceded by a space or is at the start of the element, and is either followed by a space or is at the end of the element.
- A *numeric element* is any element containing a number. If it contains more than one number, the first number is the relevant one.
- When adding a secondary ID to the tree, non-numeric elements are treated normally. In other words, all secondary elements below a particular node appear in alphabetical order *after* all the primary elements.
- When adding a secondary ID to the tree, numeric elements are added in numeric order between or beside any existing primary or secondary elements which differ only in the number.

See the diagram in the next section, in which secondary elements such as **3 players** (underlined) have been sorted between the others. In particular, the secondary element **10 players** has been ordered in the tree after **9 players**. According to a simple definition of alphabetical order, **10 players** would go before **2 players**, which is why alphabetical order is inadequate.

Singular/plural differences have to be avoided when naming numeric elements, because (say) **1 player** and **3 players** would not be recognized as differing only in the number. S3W does not use **1 player** because sounds implicitly use a single player unless otherwise specified.

Substituting numeric elements

It is not essential, but desirable, for S3W-compatible programs to treat numeric elements specially when substituting. The basic idea is that if a program finds that (say) **brass.trumpet.4 players** is unavailable, but **brass.trumpet.5 players** is available, then that should be its first fallback choice. It should even be preferred to **brass.trumpet.2 players** if the latter appears earlier in the tree. So numeric elements should fall back to their neighbour which is *numerically* closest before all others.

This requires some modifications to the **Algorithm to substitute the whole tree** described earlier, as follows.

First, we assume that numeric elements which are siblings and differ only in their number appear in numerical order and with no other nodes between them. This is guaranteed for secondary nodes by **Numeric elements when adding secondary IDs** above. It cannot be guaranteed for primary nodes: programs should just cope as best they can in the rare case that primary nodes are not in numeric order or have other nodes between them (without reordering them).

The simplest way to deal with groups of numeric elements is to separate them from their siblings by temporarily inserting a special node in their ID, which in this example is denoted by **#**:

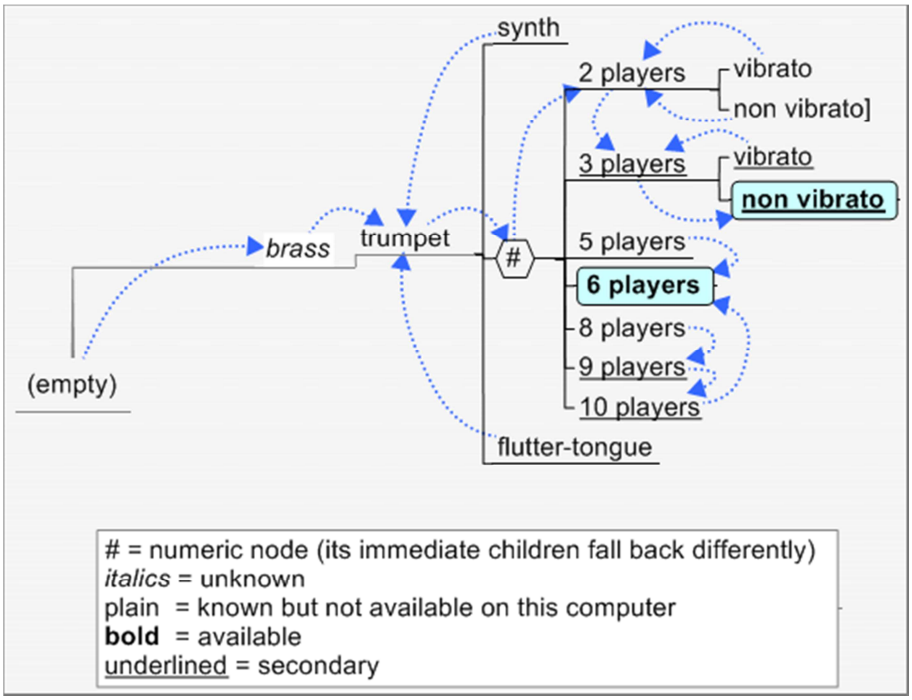
Before	*After*
brass.trumpet.synth	(unchanged)
brass.trumpet.2 players	**brass.trumpet.#.2 players**
brass.trumpet.3 players.vibrato	**brass.trumpet.#.3 players.vibrato**
brass.trumpet.flutter-tongue	(unchanged*)*

The fallback rules for **#** nodes and their *immediate* children are different, which affects steps 4 and 5 of the algorithm:

4. This node's own sound is clearly not available (because it hasn't been processed in step 2), so we look for a child or parent node to fall back to, as follows.
   4.1. If the node is a **#** node, and none of its children has an arrow pointing to it, draw an arrow to its first child. Otherwise draw an arrow to its parent. Then jump to step 6.
   4.2. If the node is unavailable and all its children have arrows pointing to it, or if there are no children, then:
   If the node has **#** as its parent, draw an arrow to its *nearest* sibling which does not have an arrow pointing to it, or a chain of arrows pointing to it. *Nearest* means the sibling with the nearest number, or the lower number in case of a tie. If there is no sibling that satisfies this, draw an arrow to the parent **#** node.
   If the node doesn't have **#** as its parent, draw an arrow from it to its parent. (If there is no parent, then this is the root node and there must be no sounds available at all, so nothing can be played anyway.)

Step 5 is unchanged, even for a node which has **#** as its parent.

This is actually simpler than it seems, when you look at an example:



Some points to note about this example:

- The numeric elements **2 players** and **5 players** don't have arrows to their parent like a normal node would. Instead the arrows go to their nearest sibling (and would go to the lower-numbered sibling if two siblings either side were equally near).
- The arrow from **10 players** doesn't go to **8 players** because there is a *chain* of arrows going in the opposite direction. Instead it goes to its nearest sibling which doesn't have such a chain, which is **6 players**.
- The nodes …**2 players.vibrato** and …**2 players.non vibrato** are normal nodes, so fall back to their parent in the normal way. It makes no difference that their parent is a numeric element.
- The **#** node has an arrow to its first child. This is always the case, unless the whole tree below it is empty.
- **brass.trumpet** has an arrow to the **#** node in the normal way. Only arrows *from* a **#** node, or a child of a **#** node, are treated differently.

Sound set files

Sound sets should always contain IDs which are as specific as possible. For instance, if a playback device has a trumpet ensemble sound, it's better to list it as **wind.trumpet.ensemble.3 players** than **wind.trumpet.ensemble**. This has several advantages, such as giving more information when users see a menu of trumpet sounds, and having better fallback behaviour.

Extending S3W

The primary IDs included in S3W are the most useful ones, so that menus in music programs don't get cluttered. They are based on the sounds currently available in sound libraries. Quite apart from adding completely new sounds, S3W could be massively extended just by adding new combinations of effects to existing IDs. For instance, for most acoustic instruments an accented version of its sound is possible and useful, whether or not it is currently listed in S3W. By varying the number used in numeric elements (such as players), the number of possibilities is almost infinite.

If other combinations of effects are required they can be added as secondary IDs in sound sets, or even be automatically generated by programs when the user requests a particular mixture of effects.

New versions of S3W may include more of these variations as primary IDs as they start to appear in sound libraries. It makes little difference that an ID which is currently secondary may become primary later. At worst, this can affect the order in which it appears in its branch, and therefore may affect whether the sound is used as a substitute.

S3W Design FAQs

Some important decisions were made when designing S3W: it's useful to understand these when adding secondary sounds to S3W. Designers of other sound worlds should consider using many of the same decisions.

How do I decide whether to classify the different timbres of two notes under the same ID?

It's best to refer back to the definition of a sound as a set of notes which are perceived as being *produced by the same instrument, person, object etc. using the same playing technique*. This allows for the sound of different notes to vary in other parameters such as pitch, volume, and duration. For example, the different registers of the clarinet sound so individual that they have their own nicknames, but they are still perceived as belonging to the same instrument, so should be classified under the same ID. Clearly, here our experience of listening to real instruments tends to make us regard 'the same timbre' as *almost* equivalent to 'played on the same instrument', even though sound worlds are classifications of sounds not physical instruments.

The above definition allows for broadly or narrowly defined sounds, based on broad or narrow interpretations of the phrases *same instrument* and *same playing technique*. This doesn't matter because it is always possible to subdivide an ID later, to refer to more specific sounds, without affecting the behaviour of programs. For instance, if it is unclear whether different sizes of grand piano produce different timbres, **keyboard.piano.grand** could be split into **keyboard.piano.grand.baby**, **keyboard.piano.grand.concert** and so on. If these baby and concert grand sounds are available, but a program is simply asked to play **keyboard.piano.grand**, then it will fall back to using one of these two sounds. A grand piano sound will still be produced, so subdividing the ID retains backwards compatibility.

If keyboard.piano.grand and keyboard.piano.upright are primary IDs, should keyboard.piano also be a primary ID? It seems strange because any actual piano, and therefore any piano sound, must be either a grand or an upright.

This question arises throughout any sound world. S3W has a **wind** subtree, but does this mean **wind** should be listed as a separate ID, even though there's no such thing as a generic wind instrument?

The simple answer is: it makes no difference to the operation of a program, though it makes a difference to the user interface. By listing **keyboard.piano** as a separate sound in your sound world, it will appear separately on menus, which may give users an easier choice when selecting sounds. But if no playback device provides a sound it calls simply **keyboard.piano**, the sound played when you choose this ID will typically be the first child of this ID, such as **keyboard.piano.grand**.

Unsophisticated playback devices may have a piano sound which is difficult to classify specifically as grand or upright, in which case they may claim to have simply **keyboard.piano**; but they can offer this sound even though **keyboard.piano** is not listed in the sound world. So the decision whether the sound world includes **keyboard.piano** still rests entirely on whether it's useful to show this ID to users.

Is it legitimate to specify duration (such as staccato) in an ID?

IDs only indicate timbre; other parameters must be supplied separately to play an actual note. But there are cases such as **wind.flutes.flute.staccato** where a short note played with this sound has a different timbre from a short note played using **wind.flutes.flute**. If there were no difference, the single sound **wind. flutes.flute**

would be enough for both long and short notes, despite any slight timbre change between the two. So here the duration word **staccato** is merely used to indicate a timbre change. On an instrument such as a piano there is no distinction between a short note and a staccato note so **keyboard.piano.staccato** should not exist.

The same goes for other parameters of sound such as volume: a volume word **accent** might be used to indicate a different timbre from a normal loud sound. So a loud note played using **brass.trumpet** must be different from a loud note played using **brass.trumpet.accent**, otherwise there would be no need for the latter ID.

How should drum sets and other multiple sounds be mapped onto IDs?

In MIDI, drum sets are usually represented as a single instrument for which different pitches product different drum sounds. Because these are not perceived as the same timbre at all, in SoundWorld each drum sound must be represented by a different ID. Therefore *there are no SoundWorld drum sets*, just individual sounds which can be played without specifying a pitch.

Should IDs indicate which pitches are available?

The sounds in S3W's **unpitched** branch are just a special case of the general problem of a program deciding whether a particular note is playable using a particular ID. A note may be unplayable because the ID is unavailable, the note's pitch is too high or low, or even because some other parameter (such as volume) is out of range. Unfortunately it is not practical to generalise **unpitched** so that each ID mentions the range of pitches for which it's playable, because this can vary from one playback device to another. So it is left to each music program to check whether a requested note can be played successfully using the available sounds, and if not to change some of the parameters (typically the ID) to produce a substitute note.

Should transposing instruments be treated differently?

Because SoundWorld classifies sounds not instruments, issues of transposing instruments do not arise. Implementations should always play sounds at the pitch requested: D5 played on **wind.flutes.piccolo** will sound at the same pitch as **wind.flutes.flute**, not an octave higher.

As a rare exception, it is convenient to represent individual organ stops as separate SoundWorld sounds even though some stops play at a different pitch from what is requested (such as an octave higher or lower). Organ stops are used in combination to produce registrations, and some sound libraries contain individual sampled stops to be used in combination in the same way. To support this usage in SoundWorld, we have to tolerate the fact that these sounds produce the wrong pitch, but there is a reasonable excuse: organ stops are often harmonics of the right pitch, so they can be thought of playing the right pitch but without the fundamental.

If a third party (such as a sound library publisher) invents a new secondary ID, isn't there a danger of a name clash?

Yes, but this isn't a problem as long as new IDs are self-describing. For instance, if two companies independently create a new sound called **wind.heckelphone**, both sounds will sound like a Heckelphone and thus will be similar.

So there is no need to give companies their own namespaces within IDs, which would result in monstrosities like **wind.acme sounds.heckelphone** and so on.

SoundWorld-compatible music programs

These are recommendations for the behaviour of SoundWorld-compatible programs.

Playback

A music program should play any ID (primary or secondary) if it is available. If it is not available the program should find the best substitute, using either the basic substitution rule or a more sophisticated rule. Programs should not 'refuse' to play a sound just because it's unavailable.

Substitute IDs should not overwrite the original IDs in the user's data: they should just be used on the fly for playback.

Music files

In music files saved by a program, sounds should be specified using IDs (not MIDI numbers etc.). This will reduce or eliminate the need to specify playback devices in files.

When saving an ID which was unavailable for playback, the original ID should still be saved rather than the substitute, in case the file doesn't need the fallback when played on another computer.

Music files should name the sound world that IDs come from, unless that file format only ever uses a particular sound world.

Displaying IDs in dialogs

Sounds should be primarily indicated using IDs, not the manufacturer's own sound names or MIDI numbers. This other information can be displayed in a subsidiary way if necessary, such as in parentheses. Similarly, when users have to specify sounds they should do this using IDs not numbers.

IDs should be capitalized when displayed, because their internal form may be lower case for convenience (as it is in S3W). If an ID is displayed in its entirety, rather than splitting the elements into submenus (say), the separation between elements should be retained. Elements can be separated with the '.' character that IDs use internally, or some other character such as an arrow, but not with a space as this is too ambiguous. For instance, it's better to display the ID **woodwind.english horn** as 'Woodwind.English Horn', than as 'Woodwind English Horn' which would be indistinguishable from **woodwind.english.horn**.

Even when a track or staff uses a fallback sound, the main ID displayed should be the *requested* ID; the fallback ID can be displayed in a subsidiary way, such as in parentheses.

Displaying IDs in menus

Menus (or similar controls) from which users can choose sounds should normally list all known sounds, not just primary sounds or available sounds. This makes it possible to create files using IDs which don't happen to be available. In some programs users may need to type in arbitrary IDs, in case they are not just unavailable but unknown.

Hierarchical menus make it difficult to display non-leaf IDs. For instance, if both **wind.flutes.flute** and **wind.flutes.flute.flutter-tongue** exist, there are two ways this could be shown. Firstly with the **flute** element twice:

Wind > Flutes > Flute

Flute > Flutter-tongue

Or secondly with **wind.flutes.flute** represented by a placeholder such as '(Normal)' in the submenu:

Wind > Flutes > Flute > (Normal)

Flutter-tongue

The second method is recommended.

Adding secondary IDs

Secondary IDs can be added to a sound world to produce (in effect) a new version of it. For instance, a company might invent a secondary ID for a new sound in a sound library, and it will have to be given an order relative to the existing IDs in the sound world. A music program encountering a secondary ID can easily add it to its internal representation of the sound world tree, except that usually there is no information about the ID's order relative to other IDs. For instance, a secondary ID might be found in a music file, specifying the timbre of a track: this gives no ordering information. Equally sound sets, another place where secondary IDs could be found, don't specify the order of IDs. (They could, but merging several sound sets containing different sets of secondary IDs and dealing with potential order conflicts would create big problems.)

So it seems unfeasible for a music program to add secondary IDs to the tree in the order that they were intended. The negative effect of this is fairly small: the fallback behaviour of secondary IDs will not be ideal in some fairly obscure situations. A music program could simply add secondary IDs in the order they're encountered, so each new ID is treated as lower priority than its siblings. This is unsatisfactory because the order is non-deterministic. Instead programs are encouraged to add secondary IDs in a deterministic order, which is simply *alphabetical order* after all the primary IDs. So below each node there will in general be a list of primary IDs in correct order, followed by secondary IDs in alphabetical order. To maintain this, a program could store the index of the first secondary ID in each node, and to do an insertion sort when a new secondary ID is encountered.

Running out of channels or slots

Playback devices typically have a limit on the number of different sounds (as opposed to notes) that can play at once. For instance, if a MIDI device has 16 channels it can only play 16 different sounds at once (leaving aside drum sets). When a program needs to play a 17th sound, it can use a variant of the basic substitution rule to decide which existing channel has the closest sound to the one required, and use this sound rather than the one requested. Conceptually, it could pretend that the 16 sounds already in use are the *only* ones available; then work out which one the required sound would fall back to in the normal way. This could simply use the 'Algorithm to substitute a single ID' described earlier, though a less simplistic algorithm would be more efficient.

Internal representation of IDs

It should be more efficient for programs to store each ID internally as an array of integers instead of a string. Each ID element could be mapped onto an arbitrary (say) 16-bit number the first time it is encountered; so sequences of elements are mapped to arrays of numbers. This will work as long as a sound world contains no more than (say) 65536 different elements.

IDs available on multiple devices

It is very convenient if, for a particular music program on a particular computer, each ID refers to a specific sound on a specific playback device, even if another device can play a different version of the sound. Then a user can specify the sound of a track with an ID alone: there's no need to specify the device as well.

Below are two different ways a program can cope when several devices have sounds for the same ID, let's say the ID **keyboards.piano**.

Simple implementation

The simplest solution is if the user can just choose which device has the preferred piano sound. Thereafter **keyboards.piano** will refer to that sound, and the only way to use a piano sound on another device will be to change globally which device **keyboards.piano** refers to.

A general problem with this solution is that the user no longer has access to all the sounds available – or rather, not access to all of them at once.

Another disadvantage is that it makes it impossible to mix two versions of the same sound in the same piece of music. For instance, it will be impossible to play music for two pianos using two different instances of a VST instrument, which is sometimes needed.

Advanced implementation using device-specific IDs

The following strategy not only gives the user access to different versions of the same sound, but also makes it easy to judge the relative quality of the alternatives.

If a sound is available on several devices, we assume there is a way to list the versions of the sound in decreasing order of quality. The order could be based on a quality rating stored in a device's sound set (either a rating for the whole device, for individual sounds); or it could be user-definable; or both in combination. In general the device order may be different for different sounds.

The first version of a sound (which we treat as the 'best') is given the basic ID name, and subsequent versions are given the ID name plus a device name. For instance, suppose a particular computer has 3 versions of the ID **keyboard.piano**, available on devices called Quality Sound Module, Average Sound Library, and Tinny Sound Card, in decreasing order of quality. Then the corresponding IDs might be **keyboard.piano**, **keyboard.piano.(average sound library)** and **keyboard.piano.(tinny sound card)**, in that order. The device name must always be added to the end of the ID.

Exactly how the devices are named is implementation-dependent. It is probably best to use a 'tidy' name which could be taken from the device's sound set, rather than the name reported by the system (which might just be the name of a MIDI port). Here the names are put in parentheses, as this looks good in a menu structure where '(Normal)' is used when there is no element, like this:

Keyboard >	Piano >	(Normal)
		(Average Sound Library)
		(Tinny Sound Card)
		Grand
		Electric

It is recommended that these two device-specific IDs should be put immediately after the basic ID, as shown, and before any other IDs in the same branch (an exception to the advice given above in 'Adding secondary IDs'). This will not affect substitution behaviour, because the basic ID **keyboard.piano** is available by definition, and this will always be used as a fallback in preference to the IDs in the branch below it.

Sometimes there may be multiple instances of the same device, such as multiple instances of a plug-in instrument, or multiple identical sound modules. It is recommended that these are given the device name plus **.1**, **.2** and so on. For instance, if there were three Quality Sound Modules in the above example, they would have the IDs **keyboard.piano**, **keyboard.piano.(quality sound module).2** and **keyboard.piano.(quality sound module).3**. Note that the first is, in effect, **keyboard.piano.(quality sound module).1**. Putting the instance number in a subbranch produces good fallback behaviour if the number of instances changes.

Saving device-specific IDs

When a music program saves an ID in a file, it is usually a good idea for it to save the whole ID including the device name. This will automatically produce the following behaviour, which seems correct:

- If an ID doesn't include a device name, e.g. **keyboard.piano**, it implicitly uses the best sound available on the current computer; when the file is played back on another computer, the program will choose the best sound available there, which may be on a different device. This seems the best choice.
- However, if an ID is device-specific, e.g. **keyboard.piano.(tinny sound card)**, it shows that the user has specifically chosen a lower-quality device – perhaps to get some special effect. When the file is played back on another computer, the program will first try to use the *same* lower-quality device. This seems the best choice since it will produce the same special effect. If this sound is not available, the basic substitution rule will use the parent ID **keyboard.piano** which is typically the highest quality sound, which again seems the best choice.

Music programs must be careful that device names used in IDs one computer will be understood on another. This is a reason to use standard device names: a standard form of the name could be specified in a device's sound set, for instance. If a non-standard device name is used in an ID, perhaps because the user has renamed it, then it should be replaced with the standard name when saving the ID to a file.